EK-KA820-TM-003

KA820/KA825 Processor Technical Manual

digital™

EK–KA820–TM–003

KA820/KA825 Processor Technical Manual

April 1987

This manual is written for people who install and replace the KA825 module in the field and for people who incorporate KA825 modules into their own products or systems. The manual gives detailed information about maintenance functions. It also tells how to customize the processor and write system software, including exception handlers, interrupt handlers, and device drivers for dedicated devices.

digital equipment corporation • maynard, massachusetts

First Edition, December 1985
Second Edition, May 1986
Third Edition, April 1987

M474000

Contents

Chapter 5 Handling Exceptions and Interrupts

Chapter 6 Dedicated I/O and Memory Devices

Tables

Preface

This manual tells what you need to know about the KA820 processor to use console commands for maintenance functions and to customize the processor to suit your needs. It also gives information you need to write system software tailored to the KA820 module, including device drivers for dedicated devices, exception handlers, and interrupt handlers.

Intended Audience

This manual is for:

- People who install and replace the KA820 module in the field.
- Engineers and system programmers who incorporate KA820 modules into their own products or systems.

Before You Use This Manual

You should be familiar with the basic concepts and features of VAX computers, including the:

1. VAX instruction set and data types
2. VAX addressing modes
3. VAX memory management system
4. VAX process structure

You can find this information in the *VAX Architecture Handbook.*

Structure of This Manual

Although the organization of this manual is tutorial, you can use the manual as a reference as well. It consists of seven chapters and a set of appendixes:

Chapter 1 Describes the major functions, characteristics, and components of the KA820 module. Together with the detailed description in Chapter 2, it gives you a context for assimilating the programming and operating information in Chapters 3 through 7.

Chapter 2	Completes the overview begun in Chapter 1. It describes the operation of the KA820 module as a whole and explains the function of each part.
Chapter 3	Defines the power-up functions and options available to you on the KA820 module.
Chapter 4	Describes the console functions for the KA820. Console commands such as Examine and Boot help you to maintain the system.
Chapter 5	Gives information you need to write exception and interrupt handling software for the KA820.
Chapter 6	Provides programming information for the dedicated devices on the KA820 module: serial line units, boot RAM, EEPROM, watch chip, and RCX50 diskette controller.
Chapter 7	Gives an overview of the diagnostic software that tests the KA820 and tells what you need to know to run the VAX cluster exerciser programs and the serial-line unit diagnostic program.

Appendixes A through K contain lists and tables too lengthy to include in the text of the manual. A glossary and an index follow the appendixes.

Related Manuals

The KA820 module is one of a family of processors, memories, and adapters that use the 32-bit VAXBI bus. For a technical summary of all VAXBI modules, system components, and integrated circuits see the *VAXBI Options Handbook*.

Other related technical manuals are the:

VAX 8200 Owner's Manual

MS8200 Memory Technical Description

DWBUA VAXBI-TO-UNIBUS Adapter Technical Description

VAX Architecture Handbook

VAX Diagnostic Supervisor User's Guide

VAX Diagnostic System User's Guide

Conventions Used

CTRL/x	You form a control character by pressing the CTRL key and typing another key at the same time.
Number bases	Most numbers in the text are expressed in decimal form. Addresses are expressed in hexadecimal form. Bit patterns are expressed in binary form. Numerals for which the context does not make the number base clear are labeled decimal, hex, or binary.
Your input	Examples show your input in red.
Uppercase	Command formats show literals in uppercase.
Lowercase	Command formats show variables in lowercase.
[ ]	Command formats show optional qualifiers and parameters enclosed in square brackets.
⟨CR⟩	Carriage return.
⟨LF⟩	Line feed.
RET	Return key. Pressing RET on the terminal sends ⟨CR⟩ ⟨LF⟩ to the system. Most console commands terminate with RET. The RET key is not shown in command examples, but you should assume that you must press RET after typing a command, unless the text indicates otherwise.
`Special font`	Text in this font indicates what you see on the console terminal.

Chapter 1
Introduction to the KA825 Module

The KA825 module is a single-board VAX processor used in the VAX 8250 and 8350 computer systems. It performs at approximately 1.2 times the speed of the VAX-11/780. A set of very large scale integrated circuits on the module implements the VAX instructions. The KA825 module communicates with other nodes in the system on the 32-bit VAXBI bus.

An earlier version of this module, the KA820, ran at approximately VAX-11/780 speed. The KA825 is distinguished by bit 23 of the System Identification Register being set. For the rest of this manual, the term KA820 refers to the KA825 module; the terms VAX 8200 and VAX 8300 refer to VAX 8250 and VAX 8350, respectively.

This introduction describes major features of the KA820 module and identifies the options available to you. Chapter 2 completes this overview and provides a background for the operating and programming details that follow.

In multiprocessor systems, the first processor is called the primary processor; it is installed in VAXBI slot K1J1. Additional processors are called attached processors, and they can be installed in any slot in the VAXBI backplane. The multiprocessor system can be symmetrical, where all processors share the workload equally, or asymmetrical, where each processor is dedicated to specific functions such as I/O control or computation.

The KA820 module is designed to support three operating systems: VMS, ULTRIX-32, and VAXELN. VMS and ULTRIX-32 are general-purpose, time-sharing systems. VAXELN systems are dedicated, real-time systems developed under VMS. As a fourth alternative, you may prefer to develop your own system software for the KA820 module.

The KA820 module protects the integrity of data and processes by checking extensively for:

- Parity errors
- VAXBI transaction errors
- Unforeseen microcode conditions
- Interrupts that occur at unexpected levels

The machine-check function, which is invoked following detection of a hardware error, passes control to appropriate exception-handling software. This lets the software evaluate the situation and respond as required. In addition, the KA820 module uses a microcode-based ASCII console and provides a serial-line unit for the console terminal.

The KA820 module implements a self-test program in microcode. Self-test runs automatically on power-up and in response to a console command, and it checks the KA820 hardware thoroughly.

1.1 KA820 Functional Sections

The KA820 module consists of three major sections:

1. CPU section
2. VAXBI interface section
3. Port controller and port controller interface (PCI) bus devices section

1.1.1 CPU Section

Three processor chips carry out processor functions according to 40-bit microinstructions in control store.

- I/E chip (instruction decoding and execution)
- M chip (memory management, processor registers, and four serial-line units)
- F chip (floating-point accelerator)

Control store consists of a 15K ROM and a 1K RAM implemented in five ROM/RAM chips and protected by parity. The RAM stores microcode patches, making microcode changes as simple as software changes: DIGITAL distributes microcode patches along with software updates on RX50 diskettes. The MIB bus, which connects control store and the processor chip set, is also protected by parity.

An on-board translation buffer (BTB) stores virtual-to-physical address translation information (page-table entries) for 512 pages of memory. The BTB backs up a mini-translation buffer (MTB) in the I/E chip that stores five page-table entries. And an on-board 8K-byte data cache stores data from the most recently accessed locations in memory. The processor chips communicate with the BTB and the cache on the parity-protected 32-bit DAL bus.

1.1.2 VAXBI Interface Section

The VAXBI interface forms a second section of the KA820 module. It consists of the 32-bit, parity-protected BCI bus and the bus interconnect interface chip (BIIC). The BIIC implements the VAXBI bus protocol, including the distributed arbitration scheme and bus error checking facilities.

1.1.3 Port Controller and PCI Bus Devices Section

The port controller and dedicated PCI bus devices make up a third section of the KA820 module. The port controller buffers the transfer of addresses and data between the CPU and the PCI bus devices, as well as between the CPU and the VAXBI interface.

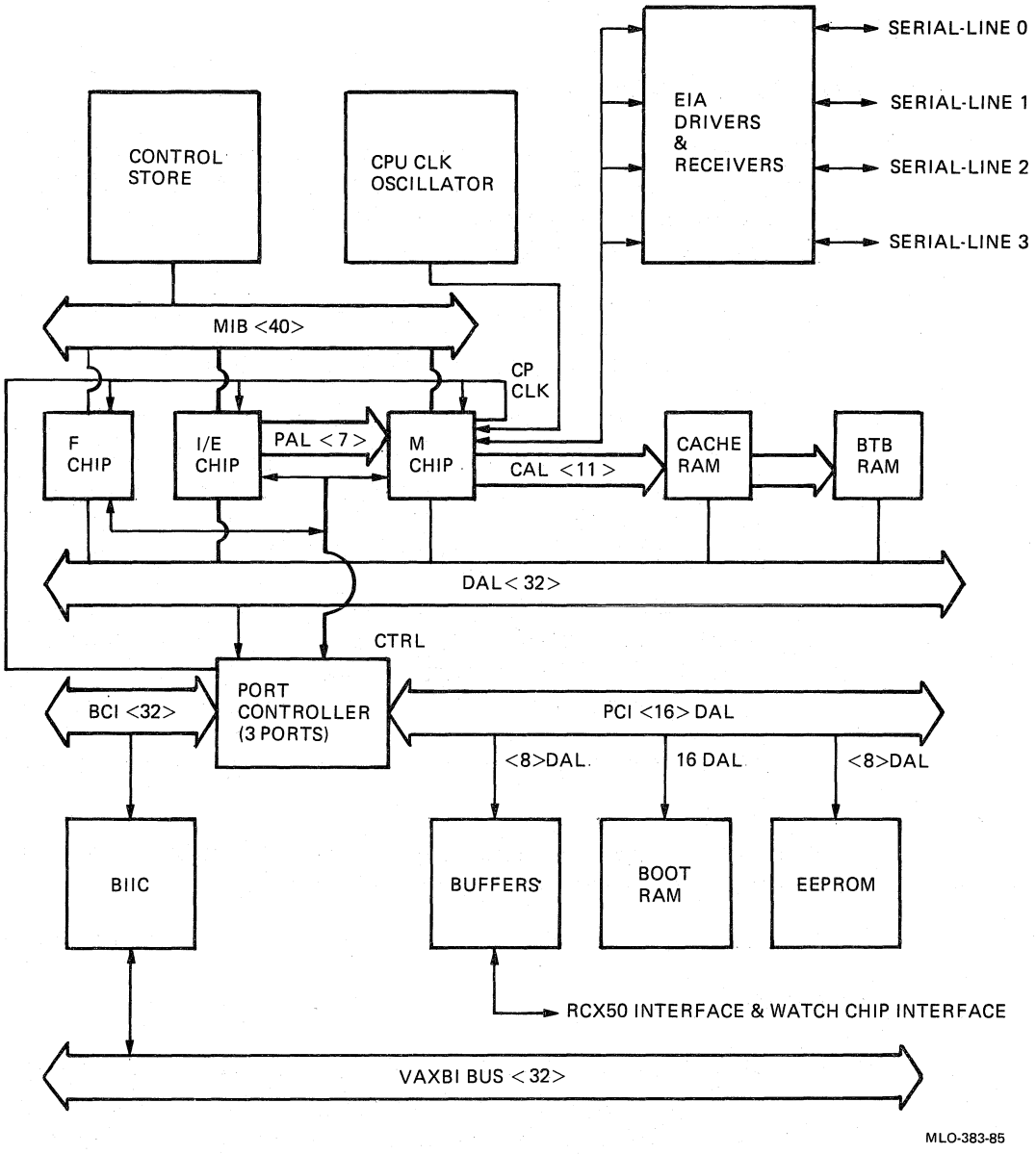


Figure 1–1: KA820 Block Diagram

The EEPROM is a 16K-byte nonvolatile memory on the PCI bus. It stores choices for KA820 options, VAX bootstrap macrocode, and a set of patches for control-store microcode. A write-protection circuit keeps you from changing EEPROM data by mistake. Microcode copies the bootstrap macrocode to an 8K-byte boot RAM on the PCI bus at the beginning of the boot process.

The PCI bus also runs off the KA820 module to connect to the battery-backed-up watch chip and the RCX50 controller for the RX5O diskette drive. The watch chip keeps the time of year for up to 100 hours without system power, and the diskette controller and drive let you install software and microcode updates.

1.2 Customer Options

The KA820 module offers a variety of options that let you customize your computer system. Data stored in the EEPROM and signals that run off the module to switches on the control panel define the action and characteristics of the KA820 module on power-up:

- Perform the slow self-test or the fast self-test on power-up.
- Restart or halt on power-up following self-test.

  Restart — If battery-backed-up memory has retained valid data during the preceding power outage, and no VAXBI node is faulty, try a warm start to continue the process that was executing when power failed. If memory does not contain valid data or a VAXBI node fails its self-test, boot the system, loading and starting a fresh copy of the operating system.

  Halt — Enter the console mode instead of beginning program execution. In this mode, the KA820 module accepts console commands from the console terminal. Console commands are useful when you want to perform system maintenance functions such as installing software or running diagnostics.

- Set the default console baud rate to one of eight values ranging from 150 to 19200.
- Select the processor node that serves as a logical console for an attached KA820 processor in a multiprocessor system.
- Enable or disable self-test on the RCX50 diskette drive controller.

1.3 VAXBI Overview

The VAXBI bus is a 32-bit synchronous bus. It joins the KA820 module to the rest of the computer system. The important characteristics of the VAXBI bus are its low cost, high bandwidth, moderate number of logical connections, large address space, and high data integrity.

All devices in the VAXBI system can arbitrate for control of the bus, so no processor is dedicated specifically to controlling bus use. Distributing the arbitration maximizes the multiprocessing capability of the bus and lets you configure systems to meet a variety of needs.

Each device on the VAXBI bus is called a node. A single VAXBI bus can have 16 nodes, which can be processors, memories, and adapters. An adapter is a node that connects other buses, communication lines, and peripheral devices to the VAXBI bus. Each of the 16 nodes can control the VAXBI bus, and slot

placement has no effect on the relative priority of the node. A node receives its priority and node ID, a number from 0 to 15 (decimal), from a plug on the VAXBI backplane slot where the node is inserted.

1.3.1 VAXBI Addressing

The VAXBI bus supports 30-bit addressing, giving 2**30 addresses (1 gigabyte of physical address space). This address space is split equally between memory and I/O space (512 megabytes each).

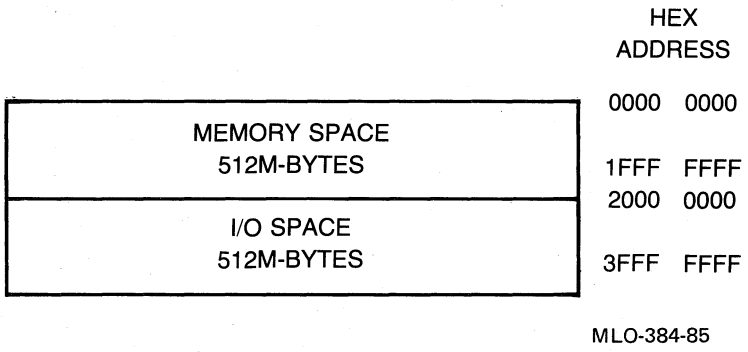


Figure 1–2: VAXBI Physical Address Space

In I/O space, each node has an 8K-byte block of addresses known as its nodespace. The first 256 bytes of each nodespace are allocated to registers on the BIIC. In addition, each node is allocated 3.75 megabytes of node private space. The dedicated devices on the KA820 module PCI bus use addresses in the node private space.

1.3.2 VAXBI Timing and Arbitration

Events occur on the VAXBI bus at fixed intervals. Data is clocked onto the bus at the leading edge of a transmit clock signal and received and latched with a receive clock signal at the end of a bus cycle. Information processing occurs during the cycle following the one in which data is transmitted and latched.

Bus arbitration and address and data transmissions are time multiplexed over 32 data lines. Interrupt sequences use command transactions and can be directed to a single processor or to several processors. Arbitration logic is distributed among all the nodes and follows a dual round-robin priority scheme.

1.4 KA820 Module Layout

Figure 1–3 shows the major components of the KA820 module. The five segments along the bottom edge of the module provide a total of 300 external I/O pins to connect with the rest of the computer system. Segments A and B carry the VAXBI bus signals.

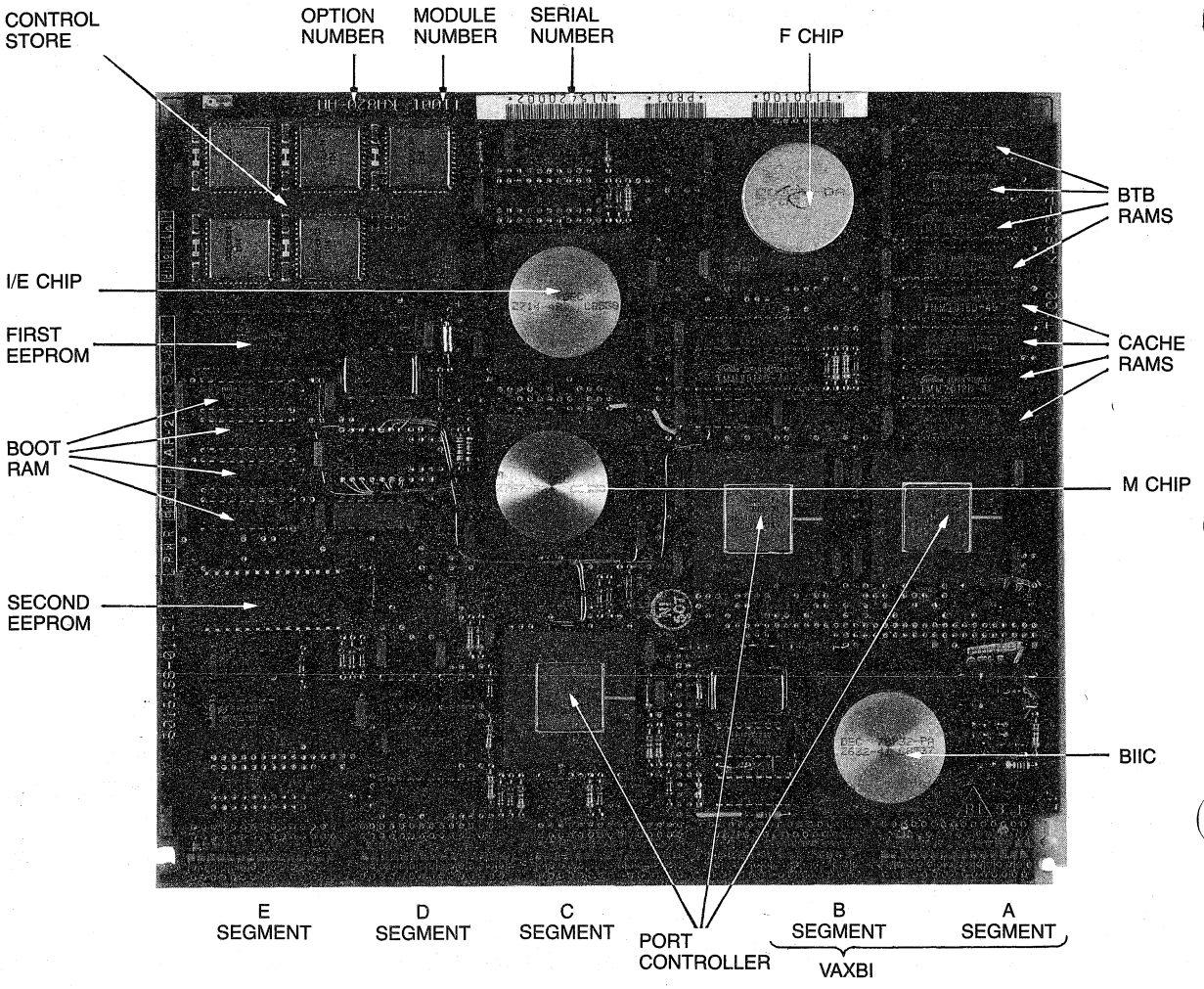


Figure 1–3: KA820 Module Layout

1.5 Power Requirements

The KA820 module requires three voltages with the voltage regulation shown in Table 1–1. The current and power columns indicate conditions at 70 degrees C (worst case). The maximum voltage ripple is 400 millivolts peak to peak.

Table 1–1: KA820 Module Power Requirements

Voltage	Regulation	Current	Power
+5.0	+/– 5%	9.0 amps	45.5 watts
+12.0	+/– 10%	36 milliamps	0.4 watts
–12.0	+/– 10%	40 milliamps	0.5 watts
			46.4 watts total

1.6 Environmental Requirements

The KA820 module requires air movement of at least 200 linear feet per minute, at a maximum ambient temperature of 50 degrees C. Temperature, relative humidity, and altitude requirements depend on use of the RX50 diskette drive with the KA820 module.

Table 1–2: Environmental Requirements

Configuration	Temperature	Relative Humidity	Altitude
Operation without the RX50	5–50 degrees C (10–122 degrees F)	10%–95%	0–2400 meters (0–8000 feet)
Operation with the RX50 installed and in use	15–32 degrees C (59–90 degrees F)	20%–80%	0–2400 meters (0–8000 feet)
Operation with the RX50 installed but not in use	10–40 degrees C (50–104 degrees F)	10%–90%	0–2400 meters (0–8000 feet)
Allowable storage conditions	–40 to 66 degrees C (–40 to 151 degrees F)	10–95%	0–9000 meters (0–30000 feet)

Chapter 2
KA820 Module Detailed Description

The KA820 central processor executes the full set of VAX instructions. Two address translation buffers, a data cache, and a floating-point accelerator help to make this processor efficient. The KA820 module can translate a virtual address to its physical equivalent and access cached data in 160 nanoseconds when there is a hit in the mini-translation buffer (MTB). When there is a miss in the MTB but a hit in the backup translation buffer (BTB), access to cached data requires 320 nanoseconds. In comparison, a read memory access with misses in both translation buffers and the cache requires at least 3.84 microseconds (24 CPU bus cycles).

Three functionally distinct sections make up the KA820 module:

- CPU section
- Interface to the VAXBI bus
- Port controller and related memory and I/O devices

The CPU section consists of a three-chip processor unit, control store, the backup translation buffer, and a cache. A 160-nanosecond CPU clock cycle, divided into eight phases, controls timing for the CPU section.

The BIIC (bus interconnect interface chip) coordinates the transfer of information between the KA820 module and the VAXBI bus. A separate 200-nanosecond clock cycle controls timing for the VAXBI bus.

The port controller (PCntl) acts as a buffer and traffic director, routing addresses, control signals, and data between the BIIC and the CPU section. The port controller communicates with the processor chips, the BTB, and the cache on the 32-bit DAL (data and address lines) bus. It communicates with the BIIC on the BCI bus. And it drives the 16-bit PCI, an asynchronous bus that connects the port controller to external devices as well as devices on the KA820 module:

- 8K-byte on-board boot RAM used to store bootstrap code during processor initialization
- 16K-byte on-board EEPROM that provides permanent storage for bootstrap code, control store patches, and configuration data
- RCX50 diskette controller external to the KA820 module
- Battery-backed-up watch chip external to the KA820 module

2.1 CPU Section

The KA820 module implements the VAX architecture in the CPU section, and the heart of this section is the processor chip set. The chip set consists of three custom-made, integrated circuits:

1. I/E chip (instruction/execution)
2. M chip (memory interface)
3. F chip (floating-point accelerator)

These chips carry out the VAX instruction set. Each VAX instruction involves a sequence of steps to access and manipulate data. Microcode for each VAX instruction controls the coordinating and sequencing of the steps. The control-store hardware contains the microcode in an array of 40-bit words arranged as 15360 words of read-only memory (ROM) and 1024 words of random-access memory (RAM). The RAM locations contain patches that update the microcode.

The two translation buffers (BTB and MTB) and the cache supplement the processor chip set with on-board storage that speeds execution time. The translation buffers contain address translation information copied from main memory. The cache holds program instructions and data, also copied from main memory.

Figure 2–1 shows the CPU section of the KA820 module block diagram given in Chapter 1.

Four buses connect the CPU components: MIB, DAL, PAL, and CAL. The parity-protected MIB (microinstruction bus) carries microinstructions, control signals, and addresses between control store and the processor chip set. Like the microword, it is 40 bits wide. The 32-bit DAL bus is also parity protected; it carries data and addresses among the processor chip set, the cache, and the backup translation buffer RAMs, and from them to the rest of the KA820 module through the port controller. The PAL bus carries address signals for the backup translation buffer tags and cache tags in the M chip. And the CAL bus carries address signals from the M chip to the backup translation buffer and cache RAMs.

2.1.1 I/E Chip Functions

Four logically distinct areas make up the I/E (instruction/execution) chip:

- Instruction buffer
- Microsequencer
- Execution unit
- MTB (mini-translation buffer)

The instruction buffer is a silo that holds up to two longwords of prefetched VAX instructions. This lets the CPU execute sequences of instructions rapidly, without waiting for memory read cycles to fetch instructions. The hard-

ware attempts to keep the instruction buffer full. When it is not full and there is no other activity on the DAL bus that takes precedence, the instruction buffer initiates a read function. In addition, the instruction buffer sends information it gathers from decoding VAX instructions to the execution unit and the F chip.

The microsequencer in the I/E chip determines the address of the next microinstruction to be executed, except at the beginning of a VAX instruction and when the first part done (FPD) flag is set following an interrupt or exception. When a new VAX instruction is being decoded, the microaddress generator determines the entry point of the microroutine to be executed, based on the VAX opcode, the operand specifiers, and the current microinstruction. The I/E chip drives the microaddress over the MIB bus during the first half of the CPU clock cycle, and control store sends the addressed microinstruction word on the MIB bus to the execution unit during the second half of the cycle. This prefetch function assures that the execution unit never waits for a microinstruction. A new microinstruction is available at the beginning of each 160-nanosecond CPU clock cycle.

The execution unit contains the general purpose registers (R0 through R15), the arithmetic and logic units, the shifter, and the data paths. It executes the microinstructions needed to implement the macroinstructions in the instruction buffer, moving data and addresses to and from the registers and on the DAL bus to the:

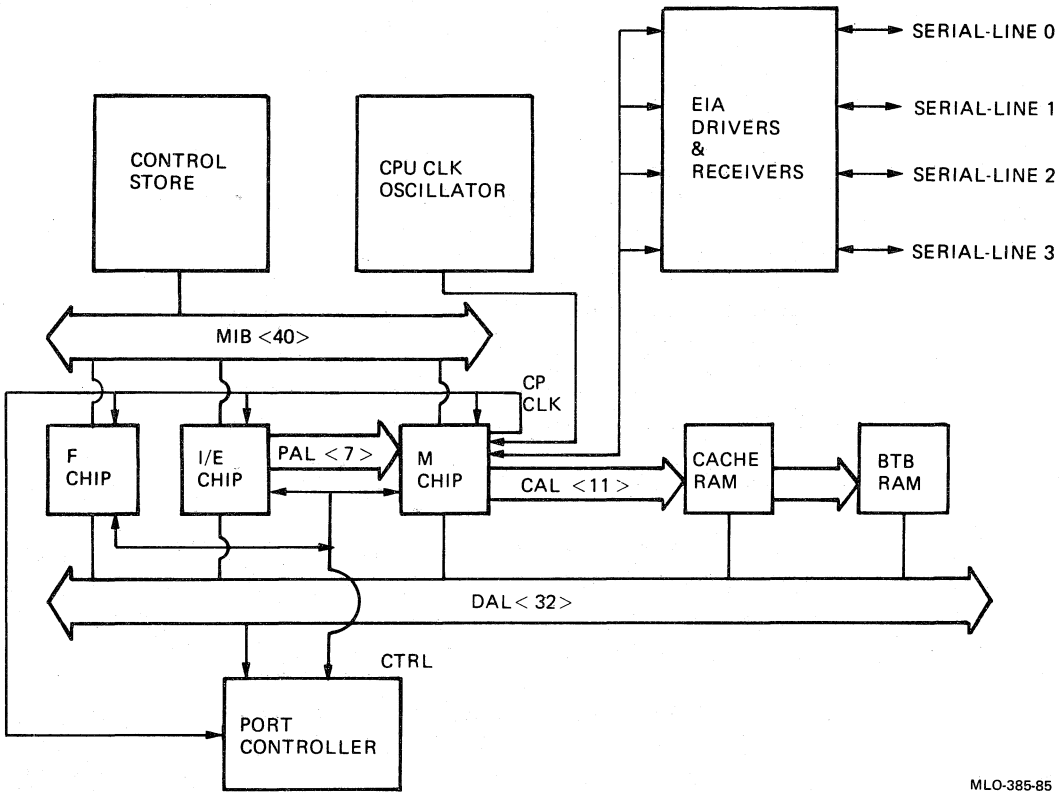


Figure 2–1: KA820 CPU Section, Block Diagram

- F chip
- M chip
- BTB (backup translation buffer)
- Cache
- Port controller

When the execution unit derives a virtual address to be accessed, it sends that address to the MTB for translation to a physical address.

The MTB stores physical address translations for five pages of virtual memory: four data-stream pages and one instruction-stream page. The information for each address translation is called a page table entry (PTE). Each MTB location contains a tag and a page table entry. The tag tells whether there is a valid page table entry (a hit) in the MTB for a given virtual address. The page table entry includes a 21-bit page frame number identifying the 512-byte page of physical memory to be used on references to the virtual address. On a hit, the MTB generates the physical address from the page table entry and makes the required physical address available on the DAL bus without any delay. See Section 2.1.2 for more information on address translation.

2.1.2 M Chip, BTB, and Cache Functions

The M (memory interface) chip functions complement the functions performed by the I/E chip; they include:

- BTB tag store
- Cache tag store
- Internal processor register (IPR) implementation
- Interrupt handling
- Memory management
- CPU clock generation
- Serial-line unit implementation
- Port controller interface

2.1.2.1 BTB (Backup Translation Buffer) — The BTB backs up the MTB with 512 page table entries (PTEs). The BTB has two sets of storage locations: tag storage in the M chip and PTE storage in the BTB RAMs. Each BTB tag tells whether four corresponding PTEs are valid.

The BTB RAM contains 256 PTEs for system-space pages and 256 PTEs for process-space pages. This RAM is arranged as 128 blocks of four longwords. Each longword contains one PTE. The M chip contains 128 BTB tags, one for each block of four PTEs.

When there is a miss in the MTB, the I/E chip puts a 32-bit virtual address on the DAL bus. At the same time, the I/E chip sends virtual address bits ⟨31, 16:11⟩ on the 7 PAL lines to identify one of the 128 BTB tags in the M chip. Each tag consists of 14 virtual address bits ⟨30:17⟩ and 4 valid bits, one for each PTE. The M chip compares the BTB tag entry with the address asserted on the DAL bus. If the bits in the tag match bits ⟨30:17⟩ in the virtual address, and the valid bit is set, there is a hit in the BTB. On the same CPU clock cycle the M chip puts the address of the appropriate BTB RAM location (virtual address bits ⟨31, 16:9⟩) on the 11-bit CAL bus, even if there is a BTB miss. Virtual address bits ⟨10:9⟩ identify one of the four longwords in the four-longword block. The BTB RAM responds by sending the PTE to the MTB, and the MTB updates the corresponding location. On the next cycle the MTB gets a hit on the same virtual address, converts it to a physical address, and asserts it on the DAL bus. Therefore with an MTB miss and a BTB hit, the address translation is delayed by one cycle. Figure 2–2 shows the relation of the tags to the BTB.

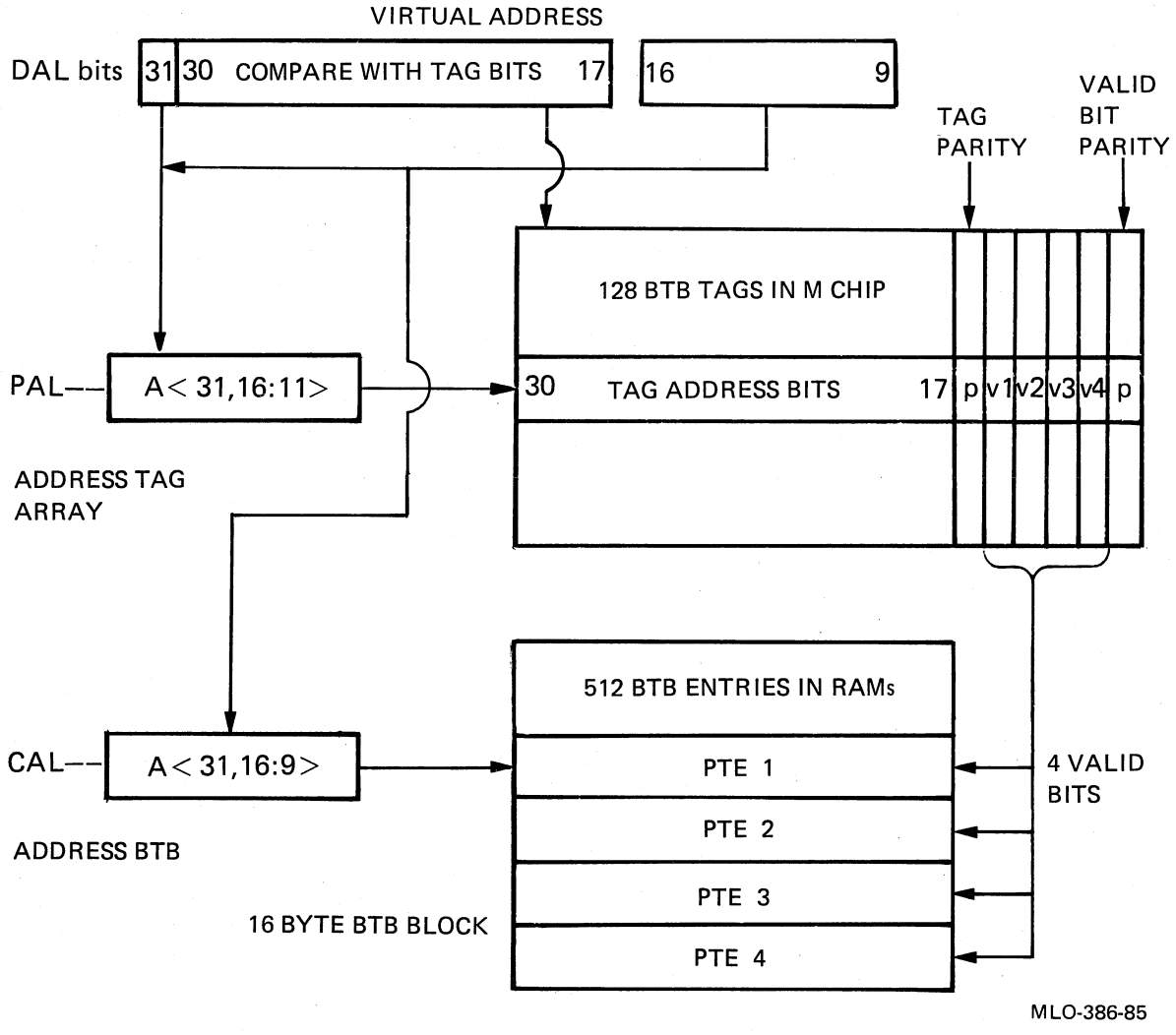


Figure 2–2: BTB and BTB Tag Addressing

If a reference to a virtual address misses in the MTB, the BTB may or may not yield a hit. The MTB will be updated when there is a hit in the BTB. If there is a miss in the BTB as well, the KA820 module begins a BTB fill cycle, performing a longword read transaction to the appropriate page table in VAXBI memory. The CPU then updates both the BTB and the MTB with a new page table entry. Since the four PTEs associated with one BTB tag identify contiguous pages in virtual memory, a change in the 14-bit address stored in the tag invalidates the other three PTEs, in the same block, that were not accessed.

Figure 2–3 shows the format for each page table entry.

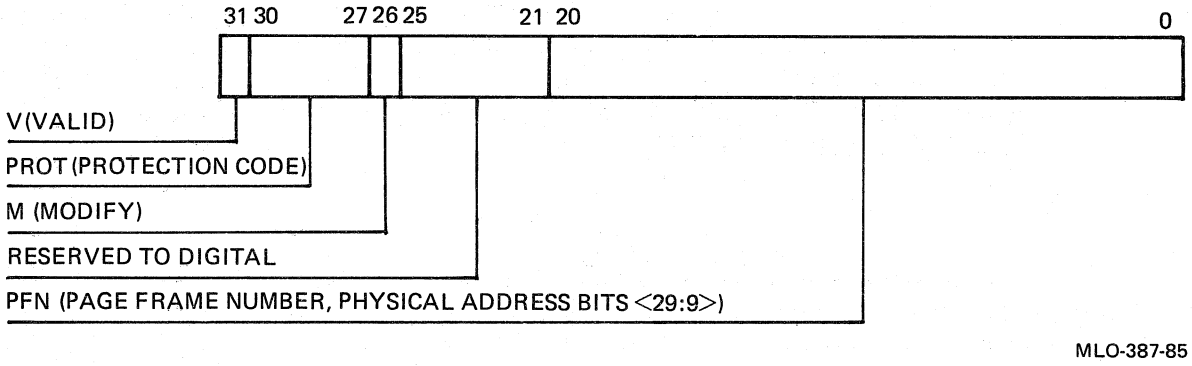


Figure 2–3: Page Table Entry Format

See the *VAX Architecture Handbook* for an explanation of memory management.

2.1.2.2 Cache — The cache RAM stores copies of data from main memory and, like main memory, uses physical addresses to access the data. The cache operation resembles the BTB operation, and the cache tag store is also implemented in the M chip.

The cache consists of an 8K-byte array, divided into 128 blocks of 4 octawords (16 longwords or 64 bytes) each. The cache tag store contains 128 tags: one for each 4-octaword block of data. Each tag includes:

- 16 bits of a physical address (bits ⟨28:13⟩) with a parity bit
- 4 valid bits with a parity bit; the four valid bits apply to the four octawords within a block of cache data

Figure 2–4 shows the cache and cache tag addressing scheme.

When the I/E chip begins a reference to a physical memory location, it puts the entire physical address on the DAL bus. At the same time it asserts 7 address bits ⟨12:6⟩ on the PAL bus to identify one of the 128 cache tag locations in the M chip. The M chip compares the tag entry with the address asserted on the DAL bus. If the addresses match, and the valid bit is set, and physical address bit ⟨29⟩ is 0 (not I/O space), there is a hit in the cache. This means that the data in the corresponding location in the cache is valid.

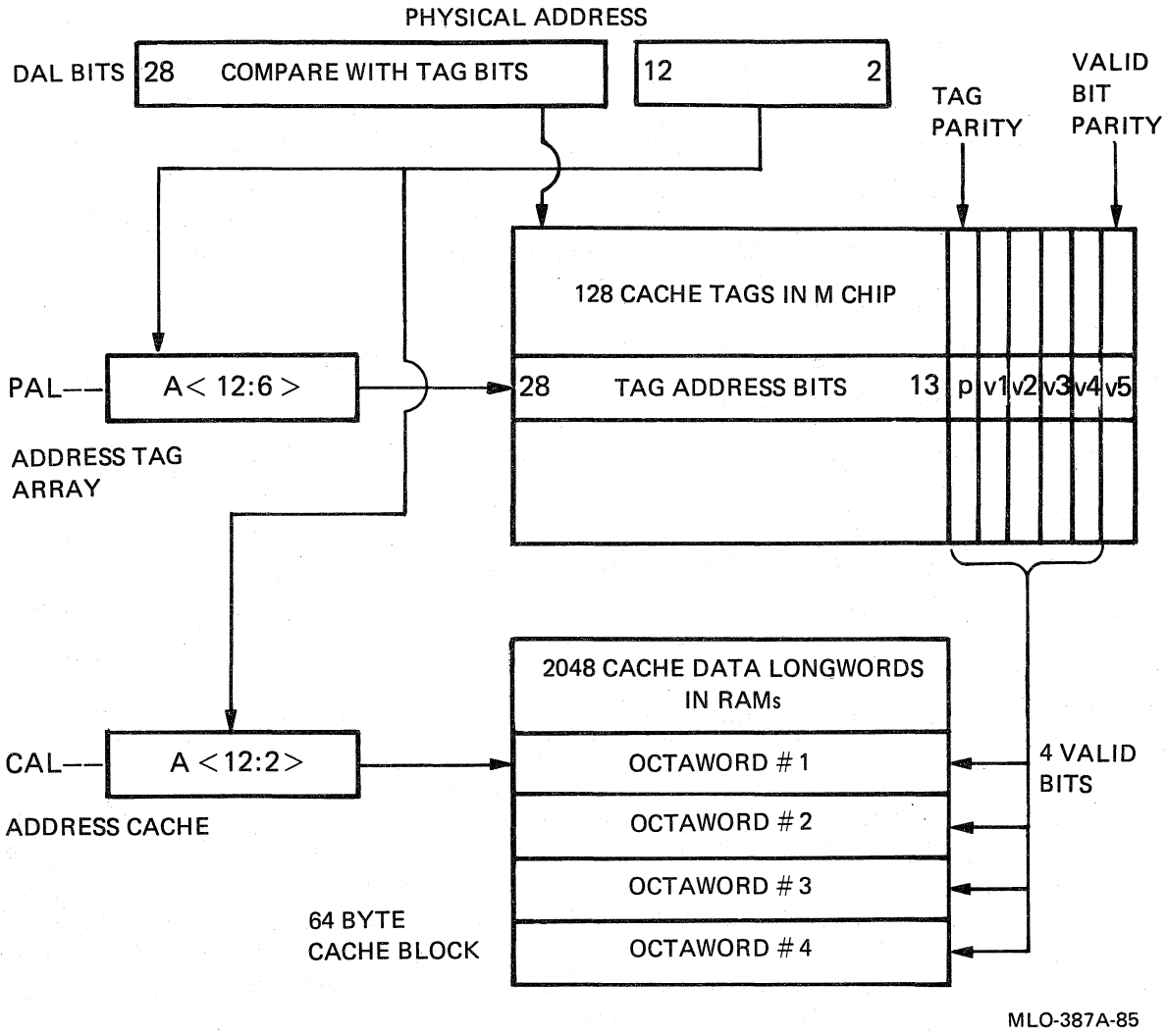


Figure 2–4: Cache and Cache Tag Addressing

On the same CPU clock cycle the M chip puts the address of the appropriate cache RAM location (physical address bits ⟨12:2⟩) on the 11-bit CAL bus, even if there is a miss. On a read transaction, the cache data is asserted on the DAL bus for use by the I/E chip. On a write transaction, new data is written both to the cache and to the corresponding location in main memory.

On a cache miss (read transaction only) the KA820 module begins a cache fill cycle, reading an octaword from main memory to update the cache. The CPU first reads the longword that was missed in the cache, completing the reference requested by the I/E chip. The CPU continues the cache fill cycle by reading the remaining three longwords in the given octaword block. The address in the tag entry will change to reflect the address of the new data, and the valid bits will show that the other three octawords in the 64-byte cache block are now invalid.

Note that a write transaction with a cache miss does not involve updating the cache.

Table 2–1 shows the number of CPU clock cycles and the time required for data access with hits and misses in the MTB, the BTB, and the cache, when there is no other activity on the VAXBI bus.

Table 2–1: Time Required for Read and Write Data Transactions

Read or Write	MTB	BTB	Cache	CPU Clock Cycles to Free DAL	Transaction Time Longword	Octaword
Read	Hit	—	Hit	1	160 ns	—
Read	Miss	Hit	Hit	2	320 ns	—
Read	Miss	Miss	Hit	12+	1.92 μs+	—
Read	Hit	—	Miss	13+	2.08 μs+	—
Read	Miss	Hit	Miss	14+	2.24 μs+	—
Read	Miss	Miss	Miss	24+	3.84 μs+	—
Write	Hit	—	—	1 or 4	160 ns	640 ns
Write	Miss	Hit	—	2 or 5	320 ns	800 ns
Write	Miss	Miss	—	12+ or 15+	1.92 μs+	2.4 μs+

If another device on the VAXBI bus writes data to a memory location that is cached by the KA820 module, the cached data becomes obsolete. To deal with this condition, the BIIC monitors the VAXBI bus for write transactions initiated by other devices. When a write transaction occurs, the BIIC forwards the address to the port controller, which in turn forwards it to the M chip. If the cache contains data for the address in question, the M chip changes the tag to mark all four octawords invalid.

2.1.2.3 Internal Processor Registers — In addition to tag storage, the M chip contains 26 of the internal processor registers (IPRs). The microcode uses these registers for temporary storage during the execution of long and complex VAX instructions. Appendix F lists the IPRs, their addresses, and their bit configurations. Note that privileged user software can read and write the IPRs with MFPR and MTPR instructions.

2.1.2.4 Serial-Line Units — The four RS423-compatible serial-line units on the M chip connect terminals and modems directly to the KA820 processor. Signal lines from the serial-line units are converted to the standard RS232 format off the module. You can set the baud rate of each serial-line unit separately by writing to the appropriate TXCS Register. In addition, you can change the baud rate on serial-line unit 0, when the primary processor is in the console mode, by pressing the BREAK key on the console terminal. Avail-

able baud rates range from 150 to 19200 (see Chapters 4 and 6 for details). You can change the default baud rate for serial-line unit 0 by writing a location in the EEPROM (see Table 4–4).

The serial-line units are not buffered. Each serial-line unit interrupts the CPU each time it sends or receives a character.

Serial-line units are available only on the primary processor.

2.1.3 F Chip Functions

The F chip, a floating-point accelerator, increases the arithmetic efficiency of the KA820 module by speeding up execution of the integer and floating-point arithmetic instructions:

- ADD(F,D,G)
- CMP(F,D,G)
- CVTL(F,D,G)
- DIV(F,D,G,L)
- EDIV
- EMOD(F,D,G,H)
- EMUL
- INDEX
- MUL(2,3)(F,D,G,H,L)
- POLY(F,D,G,H)
- SUB(F,D,G)

The F chip operates in parallel with the I/E chip. The I/E chip makes decisions concerning the instruction being executed, while the F chip makes calculations.

The MIB bus feeds the F chip with VAX opcodes from the I/E chip and microinstructions from control store. The DAL bus carries operand data for the F chip to and from memory or to and from the general purpose registers (GPRs) on the I/E chip. Operation of the F chip is transparent to users.

2.1.4 Communication Between the Processor Chip Set and the VAXBI Bus

The processor chip set communicates with the VAXBI bus through the port controller and the backplane interconnect interface chip (BIIC). When the KA820 module accesses a memory location or an I/O location, the I/E chip sends the address to the port controller on the DAL bus. Then the M chip asserts the CMISS (cache miss) signal to indicate that a memory reference is required. Four command lines transmit the required transaction type from the M chip to the port controller.

On a read transaction with a cache miss, the port controller initiates a read on the VAXBI bus. At the same time it stalls the DAL bus until it retrieves the data and can send the data to the I/E chip and the cache. The port controller includes a 4-longword data silo that buffers data during the transfer process. When the VAXBI memory responds to the read command by sending four longwords in four consecutive VAXBI cycles, the port controller stores them in the data silo until it can transfer them. The requested longword goes to the I/E chip and the entire octaword goes to the cache.

On a write transaction to memory, the port controller stores the data to be written in the silo until the BIIC can send it on the VAXBI bus, freeing the processor chip set to continue processing.

2.1.5 Control-Store Operation

Control store on the KA820 module provides microcode for three sets of functions:

- VAX instruction execution control
- KA820 module initialization, bootstrapping, and console functions
- KA820 self-test

Five custom-made ROM/RAM chips make up the control-store hardware. Altogether they contain 15360 40-bit locations in ROM, 1024 40-bit locations in RAM, and 160 14-bit locations in contents-addressable memory (CAM). The 15K ROM contains an initial version of the microcode. The 1K RAM lets you add patches that change the microcode without replacing the control-store chips. The CAM locations contain the addresses of ROM words that begin sequences of microword patches in RAM.

During the first half of each CPU clock cycle the I/E chip puts a 14-bit address on the microinstruction bus (MIB). If the current microword comes from the control-store ROM, the address is asserted on MIB ⟨13:0⟩. In the second half of each CPU clock cycle, control store checks for a match in the CAM and puts the addressed microword on the MIB bus. If there is no match in the CAM, this microword is used for the next microinstruction. However, if the CAM does find a match, it signals the I/E chip to abort the current microinstruction fetch cycle. The I/E chip then readdresses control store to read from the patch RAM, leaving address bits ⟨9:0⟩ unchanged and asserting ones on bits ⟨14:10⟩. The I/E chip continues to read microinstructions from the RAM until it reaches the end of the patch sequence. The last instruction in a patch sequence causes a jump back to a ROM location for the next microinstruction.

The KA820 module loads patches into the patch RAM in two sets: primary patches and secondary patches. Microcode in the control-store ROM loads the primary patches from the EEPROM at the beginning of the power-up initialization sequence. The primary patches modify basic microroutines that are necessary to continue the processor initialization sequence. They may involve such functions as self-test, console implementation, and loading the bootstrap code. Secondary patches are not essential to the initialization process. The macrocode in the primary bootstrap program loads these patches.

The 1024 patch RAM locations are mapped against all 15360 ROM locations. Each RAM location can patch any of 15 ROM locations, all of which use the same ten low-order address bits (⟨9:0⟩) on the MIB bus. The RAM cannot contain patches for two ROM locations for which these ten address bits match. For example, if ROM location 35C (hex) has been patched, another patch cannot be written for ROM location B5C (hex) without overwriting the patch for location 35C.

2.2 VAXBI Interface

The BIIC (backplane interconnect interface chip) mediates all VAXBI transactions in which the KA820 module participates by implementing the VAXBI protocol and sending and receiving commands, addresses, and data. It communicates with the port controller on the BCI bus, allowing the port controller to act as a buffer between the BIIC and the processor chip set. Figure 2–5 shows the VAXBI interface portion of the KA820 module block diagram given in Chapter 1.

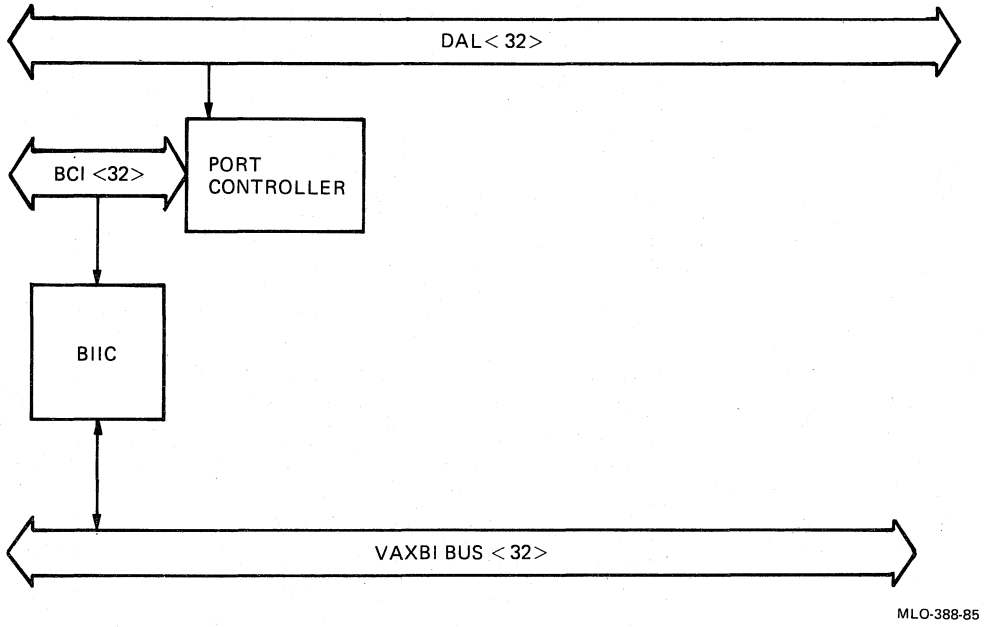


Figure 2–5: KA820 VAXBI Interface, Block Diagram

The BIIC also contains a set of control and status registers. The software uses them to handle device interrupts, interprocessor interrupts, and error conditions.

2.2.1 VAXBI Address Space

VAXBI address space is divided into two major sections: memory space and I/O space. Physical addresses with bit ⟨29⟩ set refer to I/O space; addresses with bit ⟨29⟩ clear refer to memory space. The following map shows the allocation of I/O address space among the VAXBI nodes.

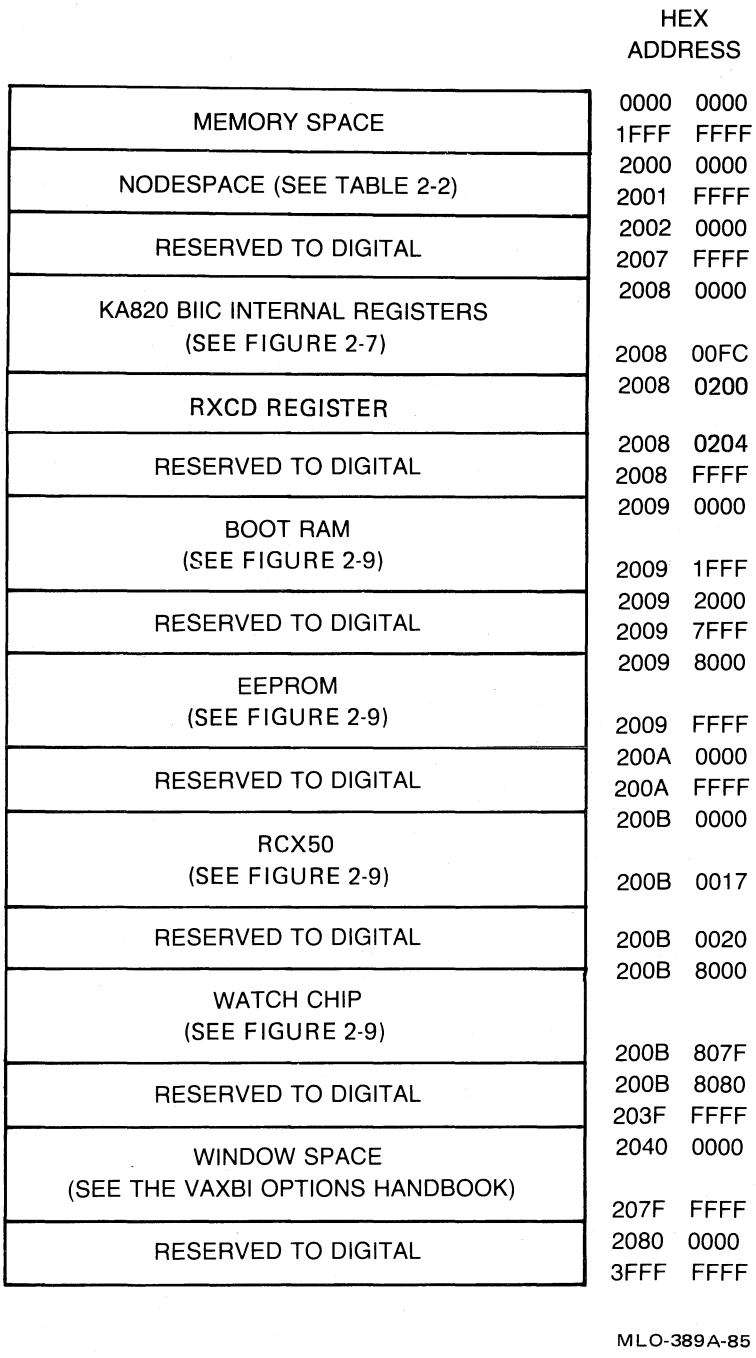


Figure 2–6: I/O Address Space on the VAXBI Bus

The first 128K-bytes are allocated to the 16 VAXBI nodes, so that 8K-bytes are available to each node. The nodespace for the KA820 module depends on the node ID plug inserted in the backplane at its slot. The primary processor uses slot K1J1. In addition, each node is allocated 3.75 megabytes for node private space, ranging from address 2004 0000 to 203F FFFF (hex). Each node can use this space to address its own registers and devices. The KA820 module implements PCI bus device addresses and one set of BIIC register

addresses (not accessible to software)in the portion of this space ranging from address 2008 0000 to 200B 807E. When a node generates a reference to an address in the node private space, it is a local reference (loopback transaction), confined to that node. No corresponding VAXBI transaction takes place. No node can access another node's private space.

2.2.2 KA820 Registers Accessible to Other VAXBI Nodes

The KA820 module implements the addresses for the RXCD Register and the BIIC internal registers in the VAXBI nodespace and in the node private space. However, software should access these registers only through the VAXBI nodespace. Software access to the RXCD Register and the BIIC internal registers through node private space will produce errors.

Note that when you use console commands, you can access the RXCD Register and the BIIC internal registers on the primary processor through their node private space addresses as well as through their VAXBI nodespace addresses. Figure 2–7 shows the BIIC internal registers and the two sets of corresponding addresses. Under the heading VAXBI nodespace address, bb stands for the nodespace base address of the VAXBI slot used. Table 2–2 shows the nodespace base address for each node.

Table 2–2: Node Space Base Address Assignments

Node	Address
0	2000 0000
1	2000 2000
2	2000 4000
3	2000 6000
4	2000 8000
5	2000 A000
6	2000 C000
7	2000 E000
8	2001 0000
9	2001 2000
A	2001 4000
B	2001 6000
C	2001 8000
D	2001 A000
E	2001 C000
F	2001 E000

Table 2–3 gives a brief description of the BIIC registers as they are used on the KA820 module. See Appendix D for the bit functions and configurations of relevant registers.

As the map in Figure 2–7 shows, the KA820 module leaves many of the registers and addresses in the BIIC unused. The RXCD Register, at address bb + 200 and 2008 0200 is implemented in the port controller, not the BIIC (see Chapter 4 and Appendix D for more information). Note that software should not access address 2008 0200.

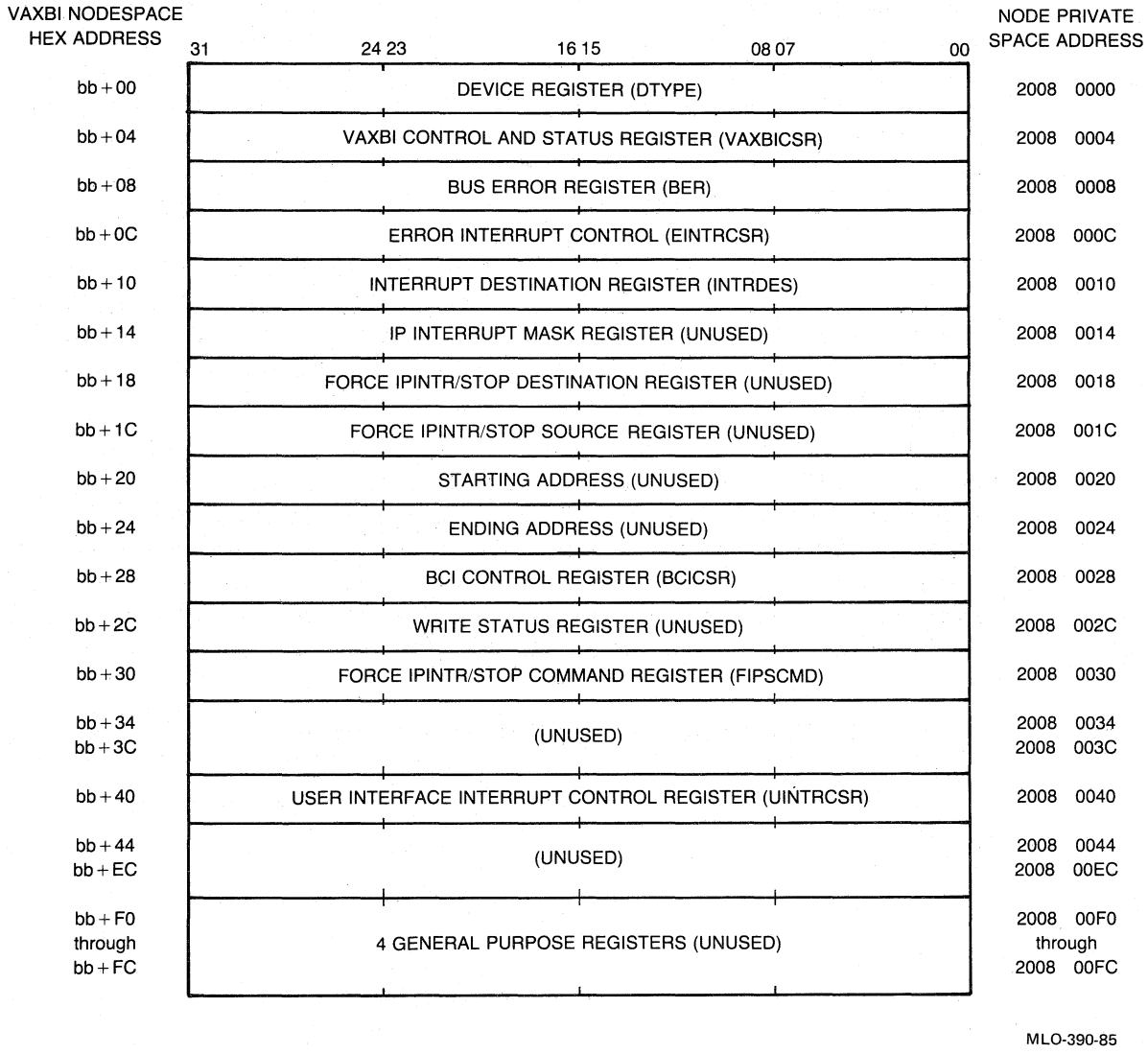


Figure 2–7: BIIC Internal Register Addresses Used on the KA820 Module (2008 0000 to 2008 0200 not accessible to software)

2.2.3 VAXBI Transactions

KA820 module microcode initiates transactions on the VAXBI bus to implement steps in VAX instructions and to respond to conditions and events on the bus. When the KA820 module responds as a slave to VAXBI transactions initiated by other nodes, VAX instructions and microcode are not involved; circuits in the BIIC and the port controller generate slave responses.

Each VAXBI transaction involves a sequence of three kinds of cycles:

1. Command/Address cycle
2. Embedded arbitration cycle
3. Data cycle

Table 2–3: BIIC Register Functions on the KA820 Module

Node Private Space Address	VAXBI Nodespace Address	Register Name	Function
2008 0000	bb+0	DTYPE	DTYPE defines this node as a KA820. Software can read it, but should not write it except when loading secondary patches (as described in Chapter 3).
2008 0004 2008 000C 2008 0010	bb+4 bb+C bb+10	VAXBICSR EINTRCSR INTRDES	Initialization software should load these registers to enable the KA820's BIIC to interrupt the KA820 when it detects VAXBI errors.
2008 0008	bb+8	BER	When the BIIC on the KA820 interrupts the KA820, the BER contains error information. Software should read and write this register whenever a VAXBI error interrupt or machine check occurs.
2008 0014 2008 0018 2008 001C 2008 0030	bb+14 bb+18 bb+1C bb+30	IPINTRMSK IPDES IPINTRSRC FIPSCMD	These four registers are designed for transmission of VAXBI IPINTR and VAXBI STOP transactions. However, system software can ignore these registers because the KA820 module supplies internal processor registers (IPIR and BISTOP) for these functions.
2008 0020 2008 0024	bb+20 bb+24	SADR EADR	System software should ignore these registers.
2008 0028	bb+28	BCICSR	Microcode loads this register and system software should not change it.
2008 0040	bb+40	UINTRCSR	System software should normally ignore this register, since the KA820 is not normally an I/O device.
2008 002C 2008 00F0 2008 00F4 2008 00F8 2008 00FC	bb+2C bb+F0 bb+F4 bb+F8 bb+FC	WSTAT GPR0 GPR1 GPR2 GPR3	These five registers are unused and system software should ignore them.

During the first cycle of a VAXBI transaction the master node sends a command code on VAXBI lines I ⟨3:0⟩ and a slave address on VAXBI lines D ⟨31:0⟩. If the command specifies a read or write function, the 32-bit field contains a physical address. On an interrupt, the master sends its interrupt priority level and a destination mask to the slave during the command/address cycle.

During the embedded arbitration cycle (the second cycle in all VAXBI transactions), other nodes contend for control of the bus following the current transaction.

One or more data cycles complete the transaction. During a data cycle the function of the 32-bit field depends on the transaction type. For example, the master sends data to the slave on a write. The slave sends data to the master on a read.

In all VAXBI transactions the slave returns one of four confirmation codes on VAXBI lines CNF ⟨2:0⟩:

ACK	Transaction acknowledged.
NO ACK	No node has been selected.
RETRY	Busy, try later.
STALL	Need more time, wait till data is ready.

The port controller monitors activity on the VAXBI bus. When an error occurs, the port controller interrupts the CPU. Microcode then initiates a trap and jumps to an error-handling microroutine. It stores machine-state information on the machine-check stack in memory, making the information available to exception-handling software. The software can evaluate the data on the stack and take appropriate action.

2.2.3.1 KA820-Initiated Transactions — Table 2–4 shows what transactions the KA820 module can initiate, together with corresponding data lengths.

Table 2–4: KA820-Initiated Transactions

VAXBI Transaction Initiated		Data Length	
Command	**Function**	**Longword**	**Octaword**
RCI	Read with cache intent	I/O space or BTB fill	Cache fill
IRCI	Interlock read with cache intent	Memory and/or I/O space	Cache fill
WCI	Write with cache intent	Memory only	Memory only
UWMCI	Unlock write mask with cache intent	Memory and/or I/O space	None
WMCI	Write mask with cache intent	I/O space or memory	None
STOP	Stop	Not applicable	Not applicable
IPINTR	Interprocessor interrupt	Not applicable	Not applicable
INTR	Interrupt	Not applicable	Not applicable
IDENT	Identify interrupting node	Longword	None

Write mask transactions involve a read-modify-write sequence, unless the mask is all ones, because a portion of the data in the target location must remain unchanged. The port controller changes a WMCI transaction with a mask of all ones to a WCI transaction.

When the KA820 module performs an IRCI transaction, the slave normally sets a lock that remains set until the KA820 module issues a UWMCI command to the same address. The KA820 module issues an IRCI/UWMCI transaction pair as part of each of the seven VAX interlocked instructions:

ADAWI

BBCCI

BBSSI

INSQHI

INSQTI

REMQHI

REMQTI

The KA820 module can initiate a STOP transaction but cannot respond to one. STOP is generally used for diagnostic functions. See the *VAXBI Options Handbook* for more information.

As the VAXBI master, the KA820 module can interrupt another processor with either an INTR or IPINTR transaction. However, KA820 modules in a multiprocessor system normally communicate with each other by writing data in the RXCD Registers.

2.2.3.2 KA820 Slave Responses — Table 2–5 shows VAXBI transactions and the confirmation codes with which the KA820 module responds to each.

As a safety feature the RXCD lock is not set unless the interlock read transaction is completed successfully. But the RXCD is unlocked by an unlock write transaction, even if the transaction is not completed successfully.

The KA820 module can respond to both the interrupt and interprocessor interrupt transactions.

2.2.3.3 Device Interrupt Sequence — When a VAXBI node (typically an I/O device) needs to interrupt the KA820 module, it arbitrates for control of the VAXBI bus according to its node number. When the node gains control of the VAXBI bus, it initiates an INTR transaction, sending its interrupt priority level on VAXBI lines D ⟨19:16⟩ and the decoded ID of the target processor on VAXBI lines D ⟨15:0⟩.

The KA820 module compares its own decoded ID with the one sent by the interrupting node and returns an ACK confirmation two cycles later if there is a match (NO ACK otherwise).

Table 2–5: KA820 Slave Responses

Command	VAXBI Transaction Received Description	KA820 Response on BI CNF ⟨2:0⟩ L
READ or RCI	Read or read with cache intent to a BIIC internal register or RXCD	ACK
WRITE or WCI	Write or write with cache intent to a BIIC internal register or RXCD	ACK
IRCI	Interlock read with cache intent to RXCD (with RXCD unlocked)	ACK
IRCI	Interlock read with cache intent to RXCD (with RXCD locked)	RETRY
UWMCI	Unlock write mask with cache intent to RXCD (with RXCD unlocked)	ACK
UWMCI	Unlock write mask with cache intent to RXCD (with RXCD locked)	ACK
INTR	Interrupt	ACK
IDENT	Identify interrupting node	ACK
IPINTR	Interprocessor interrupt	ACK
STOP	Stop	NO ACK
INVAL	Invalidate	ACK
BDCST	Broadcast	NO ACK

On a match, the KA820 module also sets one of four interrupt pending flags in the port controller, corresponding to the interrupt level of the transaction. The port controller then forwards the states of these flags (note that more than one may be set) to the M chip. The M chip compares the interrupt level of the highest flag with the priority of the process the KA820 module is currently executing. When the priority of the current process drops below that of the highest-priority pending interrupt, microcode initiates an IDENT transaction to identify the interrupting node. The BIIC on the KA820 module then arbitrates for control of the VAXBI bus, and eventually the KA820 module becomes bus master.

As bus master, the KA820 module asserts the IDENT command code on VAXBI lines I ⟨3:0⟩ and the level of the highest pending interrupt on VAXBI lines D ⟨19:16⟩ during the command/address cycle. The second cycle of IDENT allows arbitration for control of the VAXBI bus for the next transaction. The master asserts its decoded ID on the bus during the third cycle.

Then, on the fourth cycle, IDENT arbitration takes place. If several nodes have interrupts pending at the level indicated, each asserts its decoded ID (used here as an interrupt sublevel) on VAXBI lines D ⟨31:16⟩. The arbitrating node with the lowest ID (highest priority) wins the arbitration. This node returns its vector on the next cycle. If the winning node cannot respond imme-

diately, it sends a STALL confirmation on VAXBI lines CNF ⟨2:0⟩ until it can send its vector as read data. KA820 microcode uses this vector to find the interrupt service routine for the interrupting node.

Interrupting nodes that lose an arbitration must start again, regenerating interrupt transactions. The winning node must also interrupt again if it fails to receive an ACK confirmation from the KA820 module in response to its vector.

If the port controller detects a VAXBI error condition, it interrupts the M chip. The M chip then initiates a microtrap to jump to error-handling microcode (see Chapter 5 for details).

2.3 Port Controller and PCI Devices

The three ports of the port controller (BCI port, DAL port, and PCI port) make it central to the operation of the KA820 module. The BCI port connects the port controller to the VAXBI bus through the BIIC. The DAL port connects the port controller to the CPU. And the PCI port connects the port controller to the 8K-byte boot RAM, a 16K-byte EEPROM, a watch chip, and an RCX50 diskette controller. The timing for each of the three ports is separate: the BCI bus runs synchronously with the VAXBI clock cycle; the DAL bus runs on the CPU clock cycle; and the PCI bus is asynchronous. The port controller keeps the timing independent by buffering control signals, addresses, and data.

Three gate array chips make up the port controller: two data path chips and a control chip. Figure 2–8 is a portion of the KA820 block diagram given in Chapter 1; it shows the port controller and the devices that sit on the PCI bus.

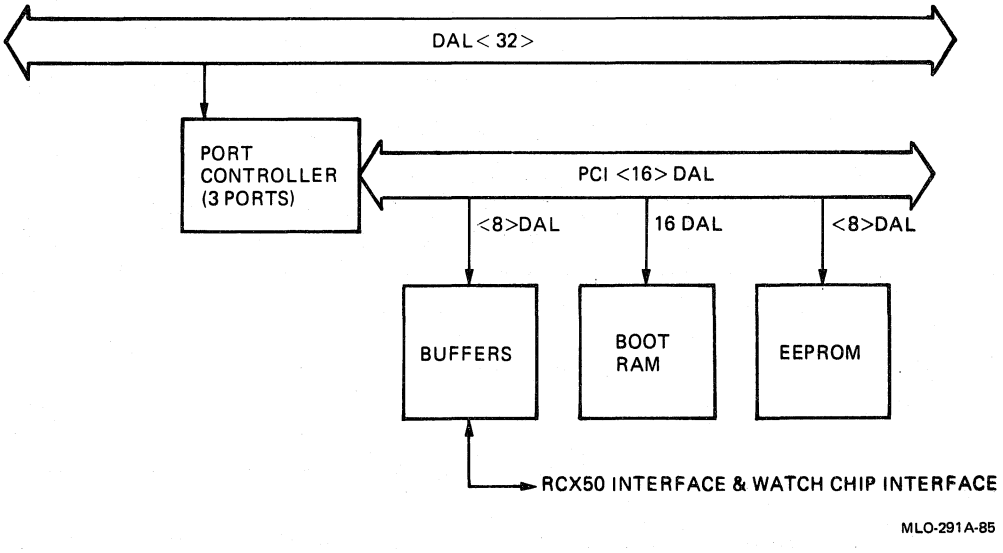


Figure 2–8: Port Controller and PCI Devices, Block Diagram

A 32-bit control and status register (PCntl CSR) and the 4-longword data silo contribute to the central function of the port controller. The data silo buffers octaword transfers on the VAXBI bus. The PCntl CSR contains status information for use by the microcode and software, control bits accessible to the CPU, and control signals from the control panel. The status bits tell when an interrupt is pending, when VAXBI transaction errors occur, and whether the KA820 module has passed self-test. See Appendix E for details. The PCntl CSR is located in the KA820 node private space at address 2008 8000 (hex); it is inaccessible to other nodes on the VAXBI bus.

2.3.1 PCI Bus Addressing

Devices attached to the PCI bus have I/O addresses in the KA820 node private space. They are accessible only to the KA820 CPU, through the DAL port on the port controller. Other nodes on the VAXBI cannot read or write to these devices. Figure 2–9 is a map showing PCI device addresses.

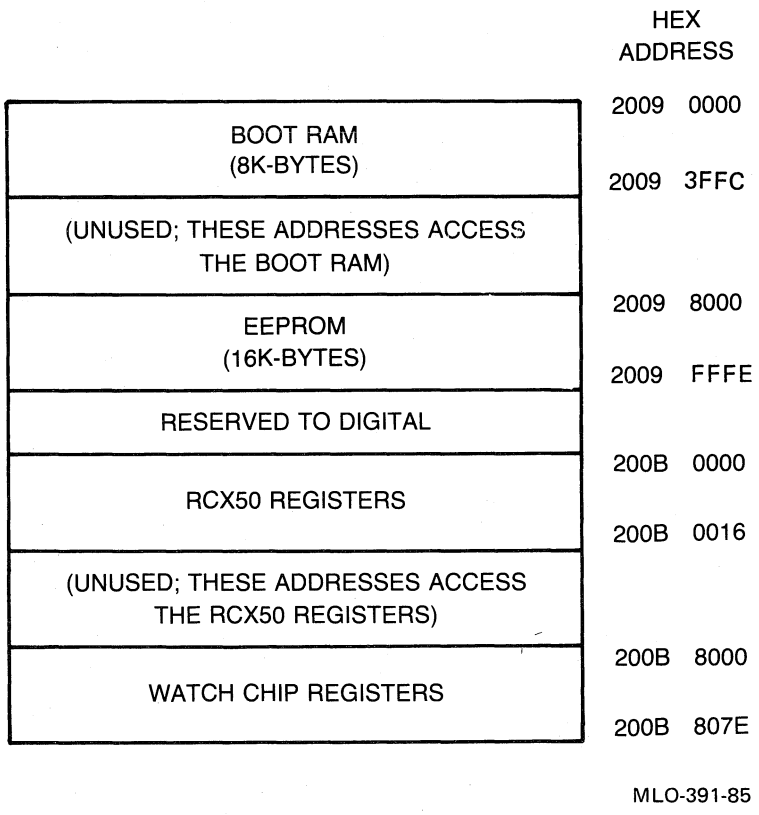


Figure 2–9: PCI Device Address Map

The 16 bits of the PCI bus are multiplexed for addresses and data, but address bit 0 is unused. This means that all PCI devices are accessed on word boundaries. However, software can access the packet buffer with all data types. The EEPROM, RCX50 controller, and watch chip are byte-oriented devices. They are accessible only with byte-length or word-length instructions on word boundaries. When word-length instructions are used, data in the high-order byte is undefined.

2.3.2 EEPROM Functions

The EEPROM (electrically erasable programmable read-only memory) on the PCI bus contains 8K-bytes of information defining options, the physical configuration, primary microcode patches, and VAX boot code. The data in the EEPROM remains valid when power goes down, so that on power-up the microcode can initialize the KA820 and boot the system according to the requirements of the configuration and the user.

DIGITAL distributes updates for the EEPROM together with software and microcode patches on diskettes. You can use the EEPROM Utility to load the update data into the EEPROM (see the *VAX 8200 Owner's Manual* for details). You can read and write 24 EEPROM locations with console D/E and E/E commands (see Chapter 4). In addition, you can use kernel mode software instructions to access any EEPROM location by using the EEPROM node private space address. You can also read and write EEPROM data with the EEPROM Utility (see the *VAX 8200 Owner's Manual* for details. The KA820 module protects the EEPROM, so that you cannot inadvertently write in it.

2.3.3 Watch Chip Interface

The eight low-order PCI bus lines and several control lines run from the KA820 module, through the backplane, to the watch chip. The watch chip lets the KA820 module keep time through a power outage or system shutdown that lasts up to 100 hours. The KA820 module stores a 32-bit time value in the TODR (Time of Day Register) in the M chip, but the TODR loses this value when the power goes down. The watch chip maintains the time after power is removed because a battery keeps it going for up to 100 hours.

Three 8-bit control and status registers on the watch chip are accessible to software, providing control that lets software write the time in the watch chip registers during installation. Then, in normal operation, software reads the watch chip to set the time in the TODR following a bootstrap operation. Because the watch chip stores time information in units of seconds, minutes, hours, days, and months, the operating system must convert this data to a 32-bit format before updating the TODR Register.

2.3.4 RCX50 Controller Interface

Like the watch chip, the RCX50 diskette controller is located off the KA820 module, and the low-order eight lines in the PCI bus run off the module to connect to the controller through the backplane. This device is normally used to load software updates and microcode patch distributions from DIGITAL. However, some system configurations do not use the RCX50 controller.

The RCX50 controller contains control and status registers and a buffer capable of storing data for one 512-byte sector on the diskette. Software can transfer commands, addresses, and data to or from the RCX50 controller in byte-length register-to-register moves (see Chapter 6 for more information).

Chapter 3
Sequences and Options on Power-Up

On power-up the KA820 processor tests and initializes itself and then initializes the rest of the computer system. Initialization leaves the hardware in a well-defined state (see Appendix G), ready for any of several courses of action. You can define in advance what action the processor takes during self-test and after initialization by choosing from a variety of options. Each option corresponds to a combination of inputs to the module I/O pins. In the VAX 8200 system the inputs come from switch settings on the control panel and jumper connections on an auxiliary module.

The power-up options include:

- Slow self-test — Test the functions of the KA820 module. In case of failure, report faults on the console terminal and enter console mode (if enabled). Complete the test in 10 seconds.
- Fast self-test — Quickly test the critical functions of the KA820 module. Enter console mode (if enabled) on failure. Complete the test in 0.25 second.
- Auto Start — Attempt to restart (warm start) the processor so that it continues executing the process that was running when the power failed. In systems with battery backup for memory, memory retains power for up to 10 minutes. If the contents of memory remain valid, the warm start succeeds. If the contents of memory are no longer valid, the warm start fails. Bootstrap (cold start) the system.
- Halt after initialization and enter console mode.
- Enable or disable the console.
- Use the physical console (serial-line unit 0) or the logical console (another VAXBI node).
- Enable or disable writing to the EEPROM.

The KA820 microcode follows a sequence of steps for each of these options.

One other option you can specify in the EEPROM affects the efficiency and behavior of the KA820 module after power-up, when it is running macro-level software or console functions. You can preset the default baud rate for the console serial-line unit. See Chapters 4 and 6 of this manual and the *VAX 8200 Owner's Manual* for information on selecting the console baud rate and writing to the EEPROM.

A typical system with a single KA820 processor might be set up to take the steps listed here on power-up:

1. Perform a slow self-test.
2. Initialize the KA820 processor and the rest of the computer system.
3. Attempt to restart the software that was running before the power failure.
4. Bootstrap the system if the restart attempt fails, loading the operating system from a disk drive.

3.1 Power-Up Sequence and Related Signals and Jumpers

The KA820 power-up sequence occurs when the BI DC LO L and BI AC LO L signals are cycled from true to false. The deassertion of BI DC LO L indicates that power to the VAXBI backplane is steady. BI AC LO L is deasserted next to indicate that dc power will remain steady for at least 4.2 milliseconds. Figure 3–1 shows the sequential relation of these two signals.

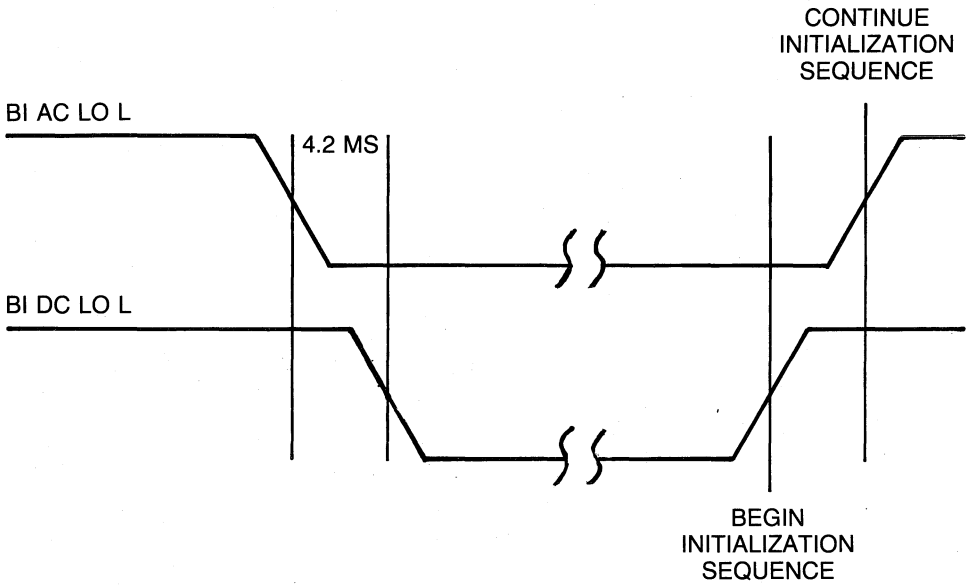


Figure 3–1: BI AC LO L and BI DC LO L Sequencing

A Restart push button on the control panel lets you simulate the power-down/power-up sequence to boot the system without actually removing power. See the *VAXBI Options Handbook* for more complete power sequence information.

Table 3–1 shows the external signals that implement the power-up options.

Table 3–2 lists the PCM module jumper configurations that affect the EEPROM update function. These jumpers control the external signal PNL ENB WT EEPROM H, listed in Table 3–1.

Table 3–1: External Signals Affecting the Power-Up Sequence

External Signal	Module I/O Pin	Accessible to Software As	Standard Sources on the VAX 8200 System	If True	If False
BI STF L — fast self-test	D55	PCntl CSR bit ⟨27⟩	Jumpers W1, W2, W3 on the PCM module	Perform self-test on power-up — jumper W2 to W3	Perform slow self-test on power-up — jumper W2 to W1
PNL RSTRT HLT H — RESTART	D51	PCntl CSR bit ⟨31⟩	Control-panel lower key switch — Update/Halt/ Auto Start	Enable an automatic restart or bootstrap on power-up — lower key switch is in the Auto Start position	Halt on power-up — lower key switch is in the Halt position; this signal is false by default on attached processors
PNL CNSL ENB H — console enable	D53	PCntl CSR bit ⟨29⟩	Control-panel upper key switch — Standby/ Enable/ Secure	Enable the console — upper key switch is in the Enable position	Disable the console — upper key switch is in the Secure position; this signal is false by default on attached processors
PN ENB WT EEPROM H — EEPROM write enable	D50	—	Control-panel lower key switch — Update/ Halt/ Auto Start and jumpers W4, W5, W6, WC on the PCM module	Enable writes to the EEPROM — lower key switch is in the Update position	Disable writes to the EEPROM — lower key switch is in the Auto Start or Halt position; this signal is false by default on attached processors
BI RESET L — reset system	B54	PCntl CSR bit ⟨28⟩	PRIM circuit on the PCM module, control-panel Restart push button	Imitate power-up and perform self-test and initialization sequence	—
BI DC LO L — dc power low	B9	—	PRIM circuit on the PCM module	Power has failed; stop processing	dc power to the backplane is steady; begin self-test when BI DC LO L becomes false
BI AC LO L — ac power low	B40	—	PRIM circuit on the PCM module	Power is failing, save state	Incoming power is steady; proceed with self-test and initialization when BI AC LO L becomes false

Table 3–2: PCM Module Jumper Configurations Affecting the EEPROM Update Function

Jumper	Use	Action
W4 to W6	Normal position	Drive PNL ENB WT EEPROM H high or low according to the lower key switch on the control panel
W4 to W5	Enable position	Enable writes to the EEPROM, overriding the control panel
W4 to W7	Secure position	Disable writes to the EEPROM, overriding the control panel

Note that in computer systems where two or more KA820 modules are inserted in a VAXBI backplane, the control-panel functions and the console functions on serial-line unit 0 are available only for the primary processor placed in VAXBI slot K1J1. The external signal enabling the physical console function is PNL CNSL LOG H (module I/O pin D52). The PCM module drives this signal false on the primary processor to enable the control-panel and console functions on the KA820 module in slot K1J1. On attached processors PNL CNSL LOG H is true, enabling the logical console function.

Figure 3–2 shows the sequence of events and options on power-up. The remainder of this chapter describes these sequences in detail.

NOTE

In Figure 3–2, W/C refers to the state of the warm start (restart-in-progress) and cold start (bootstrap-in-progress) flags handled by microcode. I cmd, T cmd, and N cmd refer to the Initialize, Test, and Next commands (see Chapter 4).

3.2 Self-Test

Self-test is the first step in the KA820 power-up sequence. It consists of microcode that checks major KA820 functions. If self-test is successful, microcode indicates success by lighting two yellow light-emitting diodes (LEDs) on the module and printing messages on the console. Self-test then passes control to the initialization and restart routines. If self-test detects an error, it fails to light the yellow LEDs and enters console mode. (The red LEDs are lit when the processor is in console mode, regardless of the self-test result.) The red FAULT LED on the control panel lights during self-test and remains lit if any VAXBI node fails self-test.

Self-test begins when the BI DC LO L signal becomes false, in response to any of five events:

1. Normal power-up.
2. An operator pushes the Restart button on the control panel, simulating power-up.
3. Console T command.

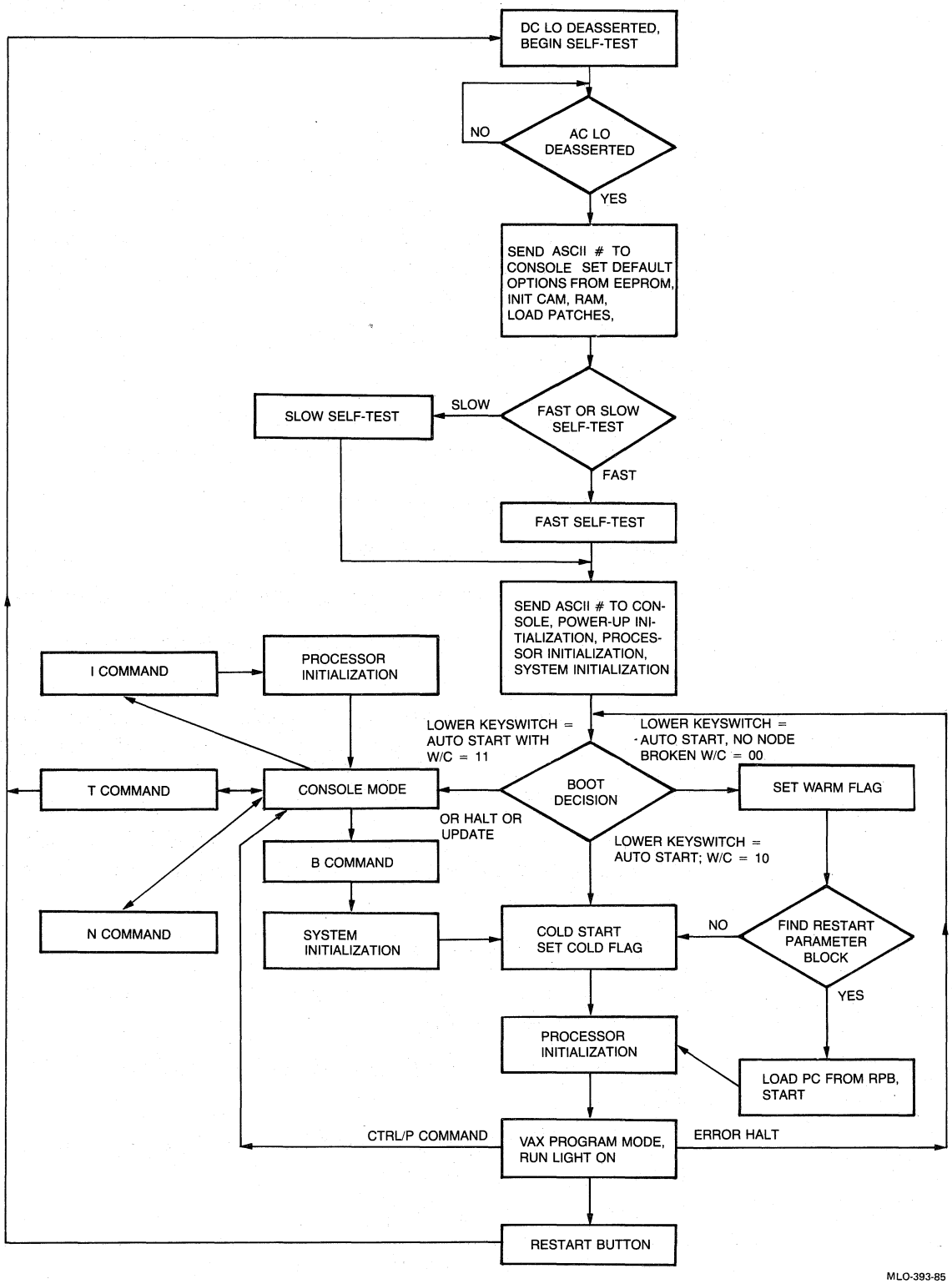


Figure 3–2: Power-Up Microcode Flow

4. Software sets the Reset bit (PCntl CSR bit ⟨28⟩).
5. The KA820 module or another node sets the Node Reset (NRST) bit (VAXBI CSR bit ⟨10⟩).

BI DC LO L in turn:

- Sets the Broke bit (BIIC CSR bit ⟨12⟩). This bit makes the self-test status of the KA820 module available to other nodes on the VAXBI bus.
- Clears the Self-Test-Pass bit (PCntl CSR bit ⟨25⟩). When clear, this bit drives the wired-OR signal BI BAD L, which lights the red FAULT light on the control panel. When set, this bit drives the two yellow LEDs on the module and clears the Broke bit.
- Sets the hardware-fault-state bit (HFSB). This bit lights the two red LEDs on the module.

Microcode then waits for the deassertion of BI AC LO L before proceeding. When BI AC LO L becomes false, microcode performs the following functions:

1. Set the console serial-line unit baud rate.
2. Send ⟨CR⟩ ⟨LF⟩ to the console.
3. Send a # character to the console, indicating that self-test has begun.
4. Calculate a checksum on the control-store patches stored in the EEPROM.
5. Compare the stored checksum with the calculated checksum. If the comparison fails, print the error code ?4A on the console terminal and halt.
6. Initialize the control-store CAM and RAM.
7. Load and check primary control-store patches from the EEPROM into the control-store RAM. On failure, print the error code ?4A on the console terminal and halt.
8. Enable control-store patches.
9. Check the operation of the console serial-line unit for all baud rates in loopback mode (see Chapter 4).
10. Clear the hardware-fault-state bit and turn off the red LEDs on the module.
11. Read the EEPROM and set defaults according to the options specified there:

    F chip enable or disable

    BTB enable or disable

    Cache enable or disable

    RCX50 self-test enable or disable

Then the self-test microcode branches according to the state of the BI STF bit, port controller CSR bit ⟨27⟩.

If BI STF L is true, fast self-test is already complete.

If BI STF L is false (the normal condition) or microcode is executing a console T command, slow self-test continues, performing a comprehensive check of the KA820 module.

Thirteen sections make up the slow portion of self-test. On the successful completion of each section, microcode sends an uppercase ASCII character (A through N, skipping L), corresponding to the hardware tested, to the console terminal. If microcode detects an error, it passes control to the console microcode. Table 3–3 lists the slow self-test functions and the corresponding characters displayed on the console terminal.

Table 3–3: Slow Self-Test Checks

Test	Code	Hardware Tested
Control-store test	A	Control store, primary patches, MIB bus, I/E chip
I/E chip internals test	B	I/E chip
DAL interface test	C	Interface of I/E and M chips and interconnecting DAL bus
M chip internals test	D	M chip, and DAL, PAL, and CAL buses
BTB array test; disable or enable BTB according to data in EEPROM; skip test if BTB is disabled	E	BTB array, and DAL, CAL, and PAL buses
Cache array test; disable or enable cache according to data in EEPROM; skip test if cache is disabled	F	Cache, and DAL, CAL, and PAL buses
I/E chip and M chip interaction test; skip test if BTB or cache is disabled	G	I/E chip, M chip
Port-controller CSR test	H	Port controller
EEPROM test	I	EEPROM, EEPROM boot code, port controller, PCI bus
Boot RAM test	J	Boot RAM, port controller, PCI bus
F chip test; disable or enable F chip according to data in EEPROM; skip test if F chip or cache is disabled	K	F chip and interconnecting DAL bus
RCX50 controller test; skip test if RCX50 is disabled	M	RCX50 controller, interface driver circuitry, port controller, PCI bus, cable
BIIC test	N	BIIC, port controller, BCI bus

If a device is disabled, self-test skips the corresponding test and prints a dot (.) on the console terminal in place of the letter.

After successful completion of either the fast self-test or the slow self-test, the microcode:

1. Sends another ASCII # character to the console
2. Sends ⟨CR⟩⟨LF⟩ to the console
3. Clears the Broke bit (VAXBI CSR bit ⟨12⟩)
4. Sets the Self-Test-Pass bit (PCntl CSR bit ⟨25⟩), which lights the yellow LEDs on the module to indicate success
5. Passes control to the power-up initialization microcode

The following example shows console output indicating a successful slow self-test.

```
#
#ABCDEFGHIJK.MN#
```

The # characters mark the beginning and end of self-test.

If any KA820 module component is missing or disabled, self-test skips the corresponding test and prints a dot instead. For example, if the BTB is not enabled, the console output for successful self-test is #ABCD.FGHIJK.MN#.

The next example shows console output indicating a self-test failure.

```
#
#ABCDE
?40
>>>
```

This shows that test F failed and there is a fault in the cache array or a related bus. ?40 is a console error message indicating that self-test failed on power-up (see Chapter 4 for a list of console error codes). The three greater-than signs, >>>, make up the console prompt symbol.

The following example shows console output on a successful fast self-test.

```
#
##
```

NOTE

Occasionally, on power-up, the console may print a few random characters before printing the self-test.

Note that in multiprocessor computer systems, attached KA820 processors in VAXBI slots other than slot K1J1 will automatically execute the slow self-test when BI DC LO L and BI AC LO L are cycled, unless the BI STF L signal is asserted on the VAXBI bus. However, attached processors will not report self-test status on the console. You can check the LEDs on these KA820 modules to determine self-test status, and software can read the Broke bit for each module.

3.3 Initialization

Initialization microcode takes control of the KA820 processor at the end of self-test. Initialization occurs in three phases:

1. Power-up initialization
2. Processor initialization
3. System initialization

At the end of this sequence the processor and main memory are ready to begin executing VAX instructions and restart or bootstrap the operating system.

3.3.1 Power-Up Initialization

Power-up initialization microcode performs five steps:

1. Load the power-up halt code 03 in the Argument Pointer (AP).
2. Clear the restart-in-progress and bootstrap-in-progress flags in the M chip.
3. Clear the Busy bit in the RXCD Register in the port controller.
4. Read the control-store patch revision number and CPU revision number from the EEPROM and load them into the VAXBI Device Register (DTYPE) in the BIIC (see Appendix D).
5. Load the BCI Control and Status Register in the BIIC, setting seven bits:

   Bit ⟨3⟩, RTOEVEN

   Bit ⟨4⟩, PNXTEN

   Bit ⟨5⟩, IPINTREN

   Bit ⟨6⟩, INTREN

   Bit ⟨8⟩, UCSREN

   Bit ⟨9⟩, WINVALEN

   Bit ⟨10⟩, INVALEN

   This setting prepares the KA820 module for VAXBI transactions. See Appendix D for an explanation of these bit functions.

3.3.2 Processor Initialization

Processor initialization puts the KA820 processor in a known state, ready to execute VAX instructions. The KA820 module must be initialized after an error halt before it resumes processing. Microcode performs this step automatically on power-up (or simulated power-up), after a console T command, and in response to a console I command.

Processor initialization involves 24 steps:

1. Turn off test mode features.
2. Load the value 041F 0000 (hex) in the Processor Status Longword (PSL) (see the *VAX Architecture Handbook* for details).

3. Clear the Map Enable Register (MAPEN).
4. Enable memory-management microcode traps.
5. Clear the MTB and BTB tags.
6. Ensure that the hardware-fault-state bit (HFSB) is clear.
7. Halt prefetching of VAX instructions.
8. Ensure that the M chip refresh function is working.
9. Load the P0LR Register with 0400 0000 (hex), setting the AST level (ASTLVL) to 4 (see Appendix F and the *VAX Architecture Handbook* for details).
10. Set the Interval Clock Control/Status Register (ICCS) to:

    Turn off the interval timer.

    Disable interval timer interrupts.

    Clear the interval timer error bit.
11. Turn on the Time of Day Register (TODR).
12. Clear software interrupts.
13. Clear the Software Interrupt Summary Register (SISR).
14. Clear the Receive Interrupt Enable bit and the Transmit Interrupt Enable bit for serial-line units 0, 1, 2, and 3.
15. Clear pending interrupts at levels 14, 15, and 16 in the ISTATUS Register.
16. Clear the P1LR Register, disabling the performance monitor enable (PME) function.
17. Clear the first part done (FPD) restart code in the M chip. The FPD code tells whether the processor has been interrupted in the middle of a long instruction.
18. Clear the machine-check frame in the MTEMP Registers in the M chip.
19. Invalidate the BTB by clearing the tags in the M chip.
20. Invalidate the cache by clearing the tags in the M chip.
21. In the PCntl CSR:
    - Clear the CRD (corrected read data) interrupt bit, bit ⟨0⟩.
    - Clear the CRD interrupt enable bit, bit ⟨2⟩, to disable CRD interrupts.
    - Clear the IP (interprocessor) interrupt bit, bit ⟨3⟩.
    - Clear the write-wrong-parity bits, bits ⟨15⟩ and ⟨23⟩.
    - Clear the console-interrupt bit, bit ⟨08⟩.

- Clear the remote-console-interrupt bit, bit ⟨10⟩, to disable RXCD interrupts.
- Unlock the VAXBI event code by clearing bit ⟨22⟩.
- Clear the Run bit, turning off the RUN light on the control panel, bit ⟨24⟩.

22. Load (or reload) the BCI Control and Status Register as explained in Section 3.3.1, step 5.
23. Load the Device Register (DTYPE) from the EEPROM.
24. Copy the bootstrap section of the EEPROM to the boot RAM, starting at location 2009 0000 (hex) in the boot RAM.

3.3.3 System Initialization

KA820 microcode initializes the computer system to prepare for restarting the operating system and user programs (or the VAX Diagnostic Supervisor) or for bootstrapping the system. The processor performs this function under four conditions:

1. As part of the power-up and simulated power-up sequences
2. In response to a console B command
3. In response to a console T command
4. In response to a console T/M command

Every node on the VAXBI bus tests itself when the BI AC LO L and BI DC LO L signals are cycled from true to false.

The KA820 processor initializes the system by polling each VAXBI node, reading the Device Register, to determine which slots are occupied and which ones contain memory nodes. If bits ⟨15:0⟩ are set to FFFF or 0000 (hex), the node is initializing or Broken. Microcode waits for a ten-second timer to expire before declaring any node faulty. If bits ⟨14:8⟩ in the Device Register are zeros, the device is a VAXBI memory node.

If a slot contains a VAXBI memory node, the KA820 microcode reads the VAXBI Memory Control and Status Register to find the amount of memory on the node and determine whether the node passed its own self-test (the Broke bit should be clear). Bits ⟨28:18⟩ indicate the memory size in 256K-byte increments. From this value the microcode calculates the starting and ending addresses of this memory node and loads them in the Starting and Ending Address Registers in the node's BIIC. The routine adds the memory size to the maximum memory count (initially zero). It then initializes the BCI Control and Status Register of this memory node by setting bit ⟨8⟩ (UCSREN) and bit ⟨13⟩ (STOPEN).

If a slot contains a nonmemory node, KA820 microcode reads the VAXBI Control and Status Register, checking the Broke bit to see if the node has passed its own self-test.

If any VAXBI node has its Broke bit set, the KA820 module does not attempt a warm start sequence. However, the processor may be capable of performing the bootstrap function.

If slow self-test is enabled, microcode prints a node number and a space on the console after finishing with each node that has passed self-test. If a VAXBI slot contains a node with the Broke bit set, microcode prints a minus sign in front of the corresponding node number on the console. If a slot with an ID plug inserted in the backplane is empty, the microcode prints a dot on the console, instead of a node number, in the position corresponding to the ID number for that slot. Microcode also prints a dot if it cannot read the VAXBICSR for the node (this can happen if the BIIC fails its self-test and turns off its driver circuits). Note that if fast self-test is enabled, microcode suppresses this console output.

When the microcode has examined all the nodes on the VAXBI bus, the value stored in the maximum memory count represents the size of VAXBI memory. System initialization microcode prints this value on the console terminal on the line following the VAXBI node status information, before passing control to the bootstrap microcode. Figure 3–3 shows the sequence of events in the system initialization process.

Example 3–1 shows console output for initialization of a particular system.

```
0 1 2 . . -5 . . . . . B . . . .
00400000
```

Example 3–1: System Initialization Console Output

This console message indicates that:

Nodes 0, 1, 2, and B are good.

Node 5 is faulty.

VAXBI slots with node ID plugs 3, 4, 6, 7, 8, 9, A, C, D, E, and F are empty, or the nodes do not respond when microcode reads the VAXBICSR.

The maximum usable VAXBI memory address is 0040 0000 (hex). This indicates that there are four megabytes of physical memory.

When the microcode finishes the system initialization sequence, it branches, depending on the settings of the control-panel switches, as shown in Figure 3–2. If the lower key switch is in the Auto Start position, control passes to the restart microcode. Otherwise the KA820 module enters console mode. The console may be enabled or disabled, according to the position of the upper key switch.

Appendix G lists the contents of the processor registers when the KA820 module enters console mode after power-up.

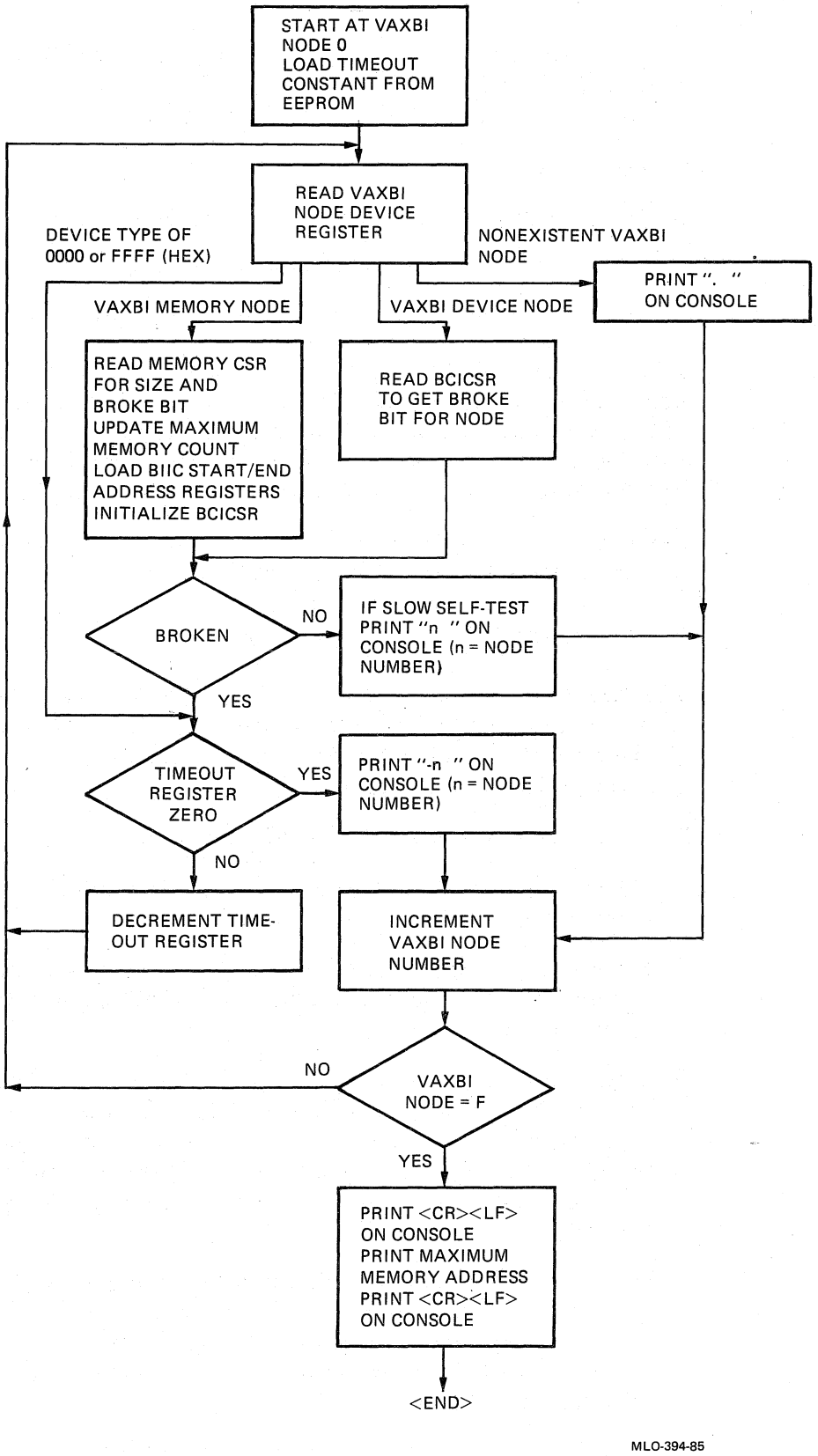


Figure 3–3: System Initialization Sequence

3.4 Restart and Bootstrap

KA820 microcode tries to restart (warm start) the software after initialization, if the RSTRT/HALT bit (PCntl CSR bit ⟨31⟩) is clear (the lower key switch is in the Auto Start position), in response to four kinds of events:

1. Power-up.
2. The processor executes a VAX HALT instruction.
3. The processor halts when it detects a machine-check error.
4. Software writes a 1 to the Reset bit (PCntl CSR bit ⟨28⟩).

If the restart attempt fails, the microcode tries to bootstrap (cold start) the system. If the bootstrap attempt fails, the KA820 processor halts. The microcode makes only one warm start attempt and only one cold start attempt. Internal flags that indicate whether a restart or a bootstrap attempt is already in progress keep the processor from looping continuously.

See Appendix G for the contents of the processor registers when the KA820 module begins executing VAX macrocode.

3.4.1 Restart Function (Warm Start)

The KA820 microcode does not try to warm-start the system if any VAXBI node has its Broke bit set, since the faulty node might have failed following the power failure and might contain data structures or hardware critical to the process that was running.

The warm start function consists of nine steps:

1. Check the restart-in-progress and bootstrap-in-progress flags in the M chip. If either flag is set, the warm start fails.
2. Set the restart-in-progress flag. Although microcode sets this bit, software should clear it by writing 0F03 to the TXDB Register (IPR 23) after a successful warm start.
3. Search main memory for a valid restart parameter block (RPB). The operating system writes the RPB when BI AC LO indicates that power is failing. The RPB lets microcode restart the software where it left off, if the memory is backed up by a battery. If microcode fails to find a valid RPB, the warm start fails. Figure 3–4 shows the four longwords that make up the RPB.
4. If a valid RPB is found, check the software restart-in-progress flag at RPB + C (hex). The warm start fails if the flag is set. Note that restart software should set this flag as soon as possible in the restart sequence. If the warm start is successful, software should clear this flag before passing control to the interrupted process.
5. Load the Stack Pointer (SP) with the physical address of RPB + 200 (hex).

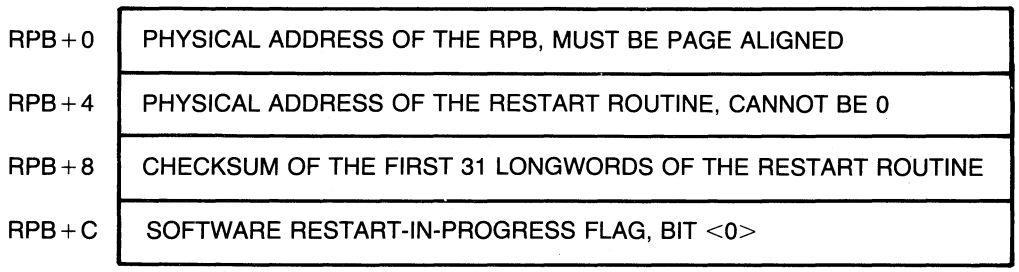


Figure 3–4: Restart Parameter Block Format

6. Load the Argument Pointer (AP) with the halt code (see Chapter 4) describing the reason for the warm start.
7. Execute the processor initialization routine listed in Section 3.3.2.
8. Turn on the control-panel RUN light.
9. Start the processor at the address found in the second longword of the RPB.

When the microcode searches for a valid RPB (step 3), it begins by looking for a page of memory that contains its own address as the first longword. The search for this page begins at address 0000 0200 and continues through the end of memory.

If the search yields a page that contains its own address as the first longword, microcode checks the second longword of the page. If it contains a valid address that is not 0, this is the address of the restart routine. If the second longword is an invalid address or 0, microcode resumes the search for a page that contains its own address as the first longword.

Microcode then reads the first 31 (decimal) longwords of the restart routine and calculates a 32-bit unsigned sum of the contents. If the sum matches the third longword of the page, microcode has found a valid RPB. Otherwise, the search continues.

If the search for a valid RPB fails, microcode tries to bootstrap (cold start) the system.

3.4.2 Bootstrap Function (Cold Start)

The KA820 module tries to bootstrap the system if a warm start attempt fails or if an operator types the B command on the console. Note that unlike the warm start attempt, the bootstrap attempt will occur even if one of the VAXBI nodes has its Broke bit set.

The bootstrap algorithm involves six steps:

1. Check the bootstrap-in-progress flag in the M chip. If the flag is set, the bootstrap fails. The KA820 processor remains halted and control passes to the console microcode.

2. Set the bootstrap-in-progress flag in the M chip, to prevent looping continuously.

3. Find a 64K-byte block of good memory that is page aligned. Note that this search leaves the locations that get checked in an unpredictable condition. If the search fails, the bootstrap fails.

4. Load the general purpose registers as follows:

   R3 — Boot device. In response to a B ⟨ddxn⟩ command from the console, microcode loads R3 with ⟨ddxn⟩. This is the boot device identifier, where dd indicates the device type, x is a hexadecimal digit identifying the VAXBI node to which the device is attached, and n is a hexadecimal digit identifying the device unit number. If the default device is used to bootstrap the system, microcode loads R3 with zeros. Note that this boot device identifier is not necessarily the device type code used by the VMS operating system.

   R5 — Boot parameter. Microcode loads R5 with a boot parameter (boot control flag) in response to a B/R5:⟨data⟩ command from the console (see Chapter 4 and Appendix I).

   R10 — Halt PC. On a bootstrap following power-up, R10 is unpredictable.

   R11 — Halt PSL. On a bootstrap following power-up, R11 is unpredictable.

   AP — Halt code. See Table 4-2 in Chapter 4 for details.

   SP — Starting address of the primary bootstrap (200 (hex) bytes past the start of good memory).

   Microcode leaves the other general registers untouched.

5. Turn on the RUN light on the control panel.

6. Start executing bootstrap macrocode beginning at physical address 2009 0104 (hex) in the boot RAM. The code in the boot RAM loads a boot block which brings in the primary bootstrap routine (generally VMB) from the boot device into main memory, beginning at the second page of the 64K-byte block of good memory. The boot device containing the primary bootstrap may be a local mass-storage device, such as an RX50 diskette. The primary bootstrap then loads a secondary bootstrap, which in turn brings in the operating system or the VAX Diagnostic Supervisor.

3.4.2.1 EEPROM and Boot RAM Bootstrap Considerations — At bootstrap time the boot RAM contains information that lets the KA820 module load the operating system from any of up to ten devices. Microcode copies this information from the EEPROM to the boot RAM as step 24 of the processor initialization sequence.

Ten device designations in the form ⟨ddxn⟩ and ten corresponding boot code addresses occupy EEPROM space beginning at location 2009 8040. The first device listed is the default bootstrap device. The last nine designations identify bootstrap devices that you can select as arguments to the console boot command in the form B ⟨ddxn⟩. The address corresponding to each device designation points to macrocode in the boot RAM that loads the boot block from that device. Chapter 4 of this manual and the *VAX 8200 Owner's Manual* explain how to change the bootstrap information in the EEPROM. Appendix H shows the contents of the EEPROM in detail.

3.4.2.2 Software Responsibilities in the Bootstrap — In addition to loading and starting the operating system, standard bootstrap software is responsible for several other functions:

- Loading the secondary control-store patches into the control-store RAM (before turning on memory management). Section 3.4.2.3 explains this procedure.
- Loading microcode for other VAXBI nodes.
- Checking main memory for all single-bit and double-bit errors, mapping out pages that contain errors.
- Clearing the hardware restart-in-progress and bootstrap-in-progress flags by writing 0F03 and then 0F04 (hex) to the TXDB Register with the MTPR instruction.
- Reading the time in the watch chip, calculating the corresponding 32-bit time value, and loading this value into the TODR Register.
- Clearing the software restart-in-progress flag, RPB + C, bit ⟨0⟩.

3.4.2.3 Loading Secondary Control-Store Patches — Bootstrap software can load secondary control-store patches in one block and then read the patches back to make sure there are no errors. The secondary patches normally overwrite the primary patches loaded into control store from the EEPROM on power-up. The KA820 processor makes three internal registers available to software for handling control-store patches: WCSL, WCSA, and WCSD. The software can use MTPR and MFPR instructions to write and read these registers.

First the bootstrap software should write the entire block of secondary patches into contiguous physical locations in main memory in the format shown in Figure 3–5.

The patch for each control-store word requires seven bytes: five bytes for the microword and two bytes identifying the ROM address that the patch is replacing. The bit order shown in Figure 3–5 is irregular, but it must be followed exactly. In addition to the seven bytes for each patch, the block requires a 7-byte header telling the number of patches in the block, the CAM address to be associated with the beginning of the block, and a checksum.

The number of patches must be an unsigned number in the range of 0 to 3FC (hex).

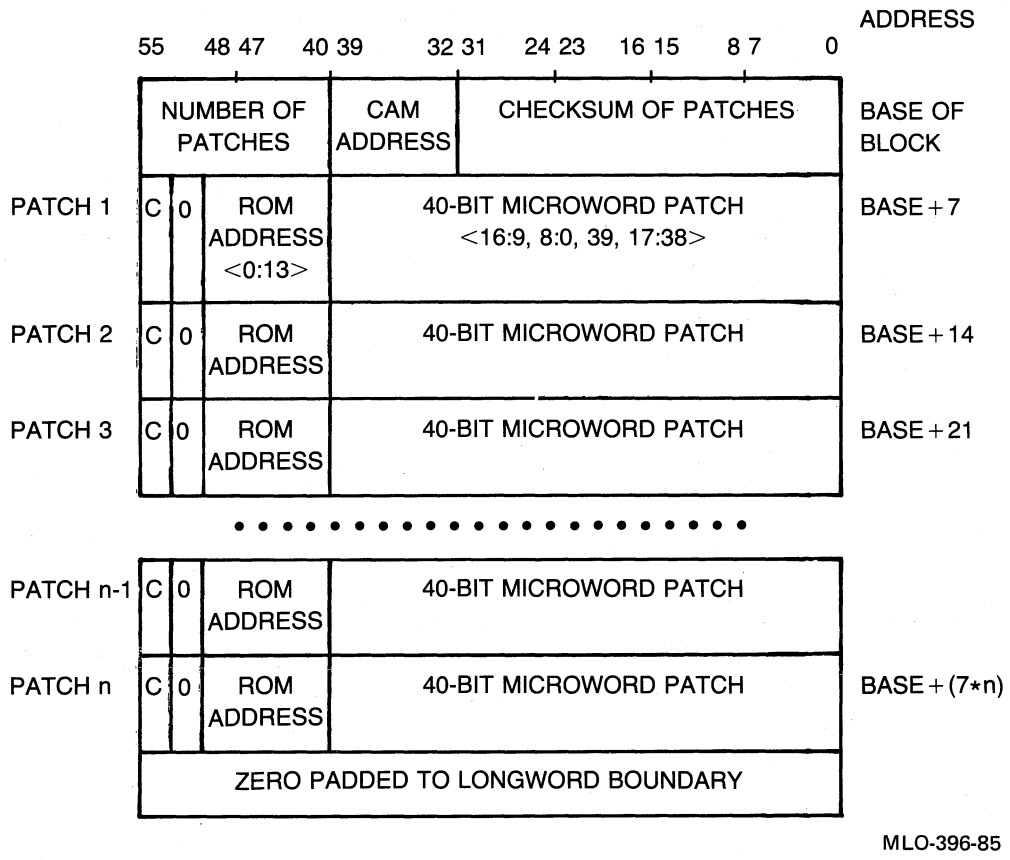


Figure 3–5: Control-Store Patch Block Format

The CAM address is a number in the range of 0 to 9F (hex) that indicates the CAM location that will contain the ROM address corresponding to the first patch.

The checksum stored in the header, when added to the unsigned longword sum of the rest of the patch block, must equal 6969 6969 (hex):

patch-block-sum + checksum = 6969 6969

You can calculate the number of longwords (X) to be summed according this formula:

X = ((number-of-patches + 1) * 7 + 3) / 4

The letter C shown in bit 55 in each patch in Figure 3–5 tells whether the CAM should be loaded with the ROM address being patched. A 1 means load the CAM. Note that the CAM should be loaded only for the first patch in a sequence. See Section 2.1.5 in Chapter 2 for an explanation of control store and the CAM function.

After bootstrap software loads the block of patches into main memory, it can load the whole block into the control-store RAM with one MTPR instruction, as shown in Example 3–2.

```
WCSL = 2E                          ; Identify the WCSL address.
.
.
.
MTPR (patch-block-base), WCSL      ; Load the entire patch block.
```

Example 3–2: Loading a Patch Block into the Control-Store RAM

This instruction loads the physical address of the base of the patch block into the WCSL Register. Figure 3–6 shows the WCSL Register format.

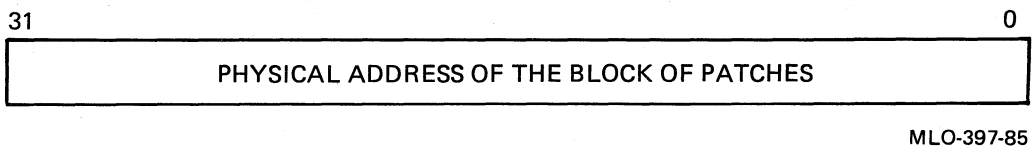


Figure 3–6: WCSL Register Format

Microcode executes this MTPR instruction by reading main memory, beginning at the patch-block base address. It calculates a checksum, compares this with the stored checksum, and loads the entire block into the control-store RAM, if the checksums agree. If the checksums do not agree, microcode does not load the patches. Software can check the condition codes in the Processor Status Longword (PSL) to determine whether the patch load has succeeded.

Condition codes on a successful patch load:

N ⟨— patch-block base address is less than 0
Z ⟨— patch-block base address equals 0
V ⟨— 0 = success
C ⟨— C

Condition codes on a patch-load failure:

N ⟨— patch-block base address is less than 0
Z ⟨— patch-block base address equals 0
V ⟨— 1 = failure
C ⟨— C

Next the bootstrap software should check the accuracy of the patches loaded by reading them back from the control-store RAM. The WCSA and WCSD registers let the software look into the control-store RAM. The software can compare the data read against the patch block in memory. Figure 3–7 shows the formats for WCSA and WCSD.

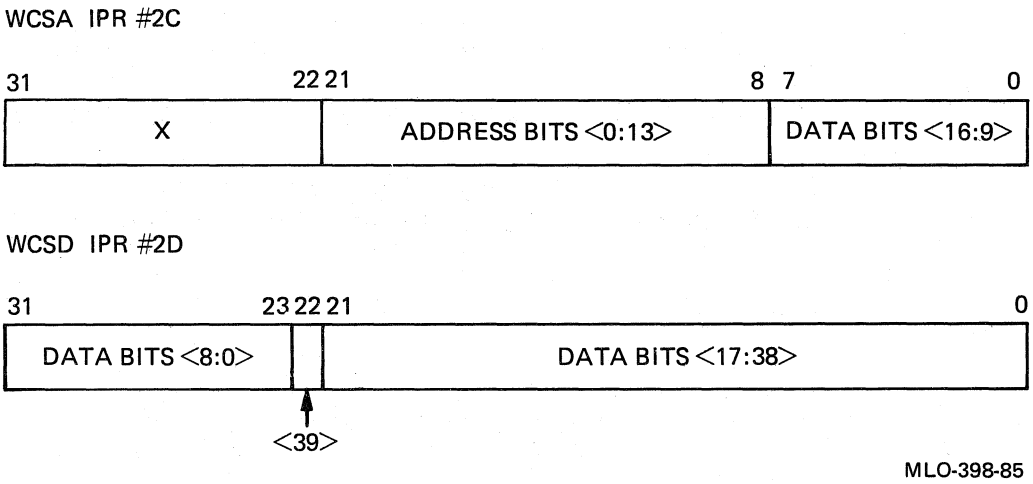


Figure 3–7: WCSA and WCSD Register Formats

To read a control-store RAM location the bootstrap software should write the WCSA Register with an MTPR instruction. Bits ⟨21:8⟩ of the longword must contain the address ⟨0:13⟩ of the control-store ROM location being patched (note that the normal bit order is reversed). Other bits in the longword do not matter. Next the software should use an MFPR instruction to read the WCSD Register to retrieve microword bits ⟨8:0, 39, 17:38⟩. The software should then use another MFPR instruction to the WCSA Register to retrieve microword bits ⟨16:9⟩. Example 3–3 shows this sequence.

```
WCSA = 2C                          ; Identify the WCSA address.
WCSD = 2D                          ; Identify the WCSD address.
  .
  .
  .
MTPR #00172400, WCSA               ; Get the patch for ROM
                                   ;   address 93A (hex)
                                   ;   (bits are reversed).
MFPR WCSD, R1                      ; Read bits ⟨8:0, 39,
                                   ;   17:38⟩.
MFPR WCSA, R2                      ; Read bits ⟨16:9⟩.
```

Example 3–3: Reading Control-Store Patches

Note that the software must execute these three instructions in the order shown to read each control-store patch. No interrupts or exceptions are allowed in this sequence.

Control-store ROM addresses range from 0000 to 3BFF (hex). RAM addresses range from 3C00 to 3FFB (hex). Addresses 3FFC to 3FFF (hex) refer to the diagnostic CAM Match Register and should not be used.

When the software reads a control-store ROM address that has been patched, the microcode returns the microword stored in the patch RAM, not the original ROM microword. After loading and checking secondary patches, software

should set bit ⟨16⟩ in the Device Register and bit ⟨8⟩ in the System Identification Register.

3.5 Sample Multiprocessor Configuration Start Sequence

In a multiprocessor VAXBI computer system only one KA820 module (the processor in VAXBI slot K1J1) attempts to restart or bootstrap the system when BI DC LO L and BI AC LO L are cycled or when an operator types the B command on the console terminal. Figure 3–8 shows a specific multiprocessor configuration.

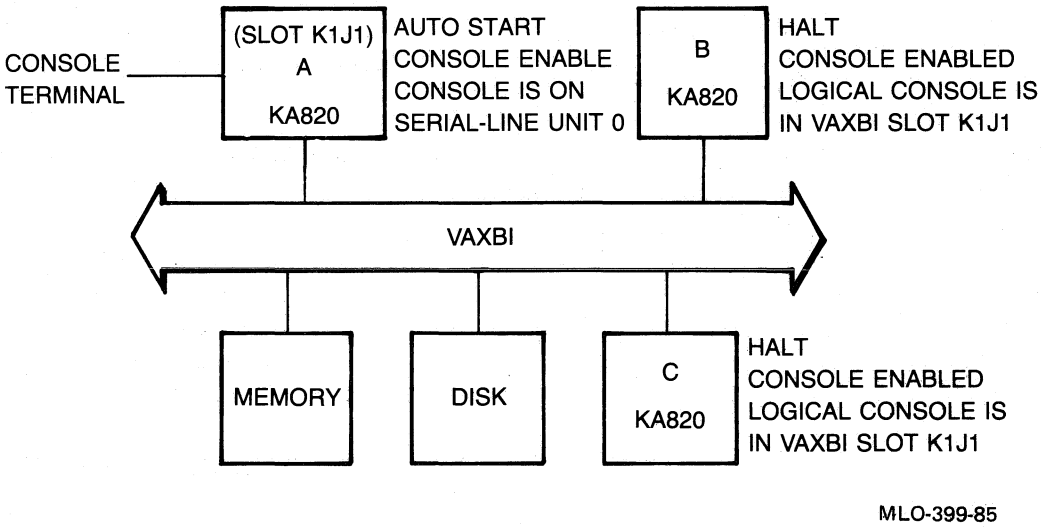


Figure 3–8: Sample Multiprocessor Configuration

In this configuration processor A is the primary processor. B and C are attached processors. On power-up all three KA820 processors perform self-test, power-up initialization, processor initialization, and system initialization sequences. Each processor polls the VAXBI bus, sizing memory and loading starting and ending addresses in the memory BIICs.

The primary processor then tries to restart the software. If this fails, it tries to bootstrap the system.

The attached processors enter the console mode automatically. Microcode in each attached processor reads the logical console byte in the EEPROM and sends the following 3 lines of characters to the RXCD Register of the primary processor (the logical console).

```
?03
     PC = xxxxxxxx
)))
```

An attached processor waits a maximum of one second for the primary processor to read each character and then sends the next character. If the primary processor is not in the forwarding console mode or running software that reads its RXCD Register, the characters sent from an attached processor are ignored and unavailable to the console operator. If you enter the console mode on the primary processor and then use the console Z command (see Section 4.3.12 in Chapter 4) the console terminal will display characters as it receives them from the attached processor you have accessed. If you wait until the attached processor has finished sending characters before using the console Z command, the console terminal will display only >, the last character sent in the console prompt, >>>.

If the primary processor succeeds in restarting or booting, software running in the primary processor can respond to the prompts from the attached processors and start them. The primary processor should send console commands to the RXCD Registers in the attached processors to set up their internal registers, load their secondary control-store patches, and start the processors running.

Example 3–4 shows a sequence of console commands that might be used to start an attached processor.

```
I<CR>                                  ! Initialize the attached
                                       !   processor.

D/I <address> <data><CR>               ! Set up internal processor
                                       !   registers.
  .
  .
  .
S <address><CR>                        ! Start the attached processor
                                       !   running.
```

Example 3–4: Commands to Start an Attached Processor

See Chapter 4 for an explanation of these console commands.

Chapter 4
Console Functions

KA820 microcode emulates the control-panel lights and paddle switches of older computer systems with an ASCII console. The KA820 console functions are compatible with those of other VAX consoles; they let you:

- Start and stop the processor.
- Examine and deposit data in registers and in memory.
- Test the hardware with the self-test microcode and macrodiagnostic programs.
- Bootstrap the system.
- Single step through VAX macrocode.
- Run stand-alone programs without using the operating system.

The KA820 console lets you perform these functions from a terminal or another KA820 processor on the VAXBI bus (logical console). You can also load blocks of data through the console from an automatic load device. The console terminal or automatic load device must be connected directly to serial-line unit 0 on the primary processor installed in VAXBI slot K1J1.

For attached processors in multiprocessor systems, RXCD Registers function like serial-line unit 0, passing commands and responses between the console microcode and the VAXBI bus. See Section 4.6 for a discussion of logical console operation in multiprocessor systems.

4.1 Console States

The KA820 console is always in one of three states when power is on:

1. Program I/O mode (the processor is running VAX macrocode)
2. Console mode (the processor is halted with the console enabled)
3. Processor is halted with the console disabled

In the program I/O mode, the console terminal acts like a conventional VAX system terminal. Microcode passes all characters, with the exception of CTRL/P, from the terminal or RXCD Register through to the software. When

you type CTRL/P on the console terminal (or another processor in the VAXBI backplane sends CTRL/P to the RXCD Register) and the console is enabled, the processor halts and enters the console mode.

In the console mode, characters typed on the console terminal are interpreted as commands or parts of commands. The console mode includes two states: local and forwarding.

- In the local console mode, microcode on the KA820 module connected to the console terminal interprets the console commands.
- In the forwarding console mode, microcode on the KA820 module sends console commands to an attached processor in another slot on the VAXBI bus, and the attached processor interprets the commands.

You can return control to the local console mode by typing CTRL/P. You can return the processor from the local console mode to the program I/O mode with the commands to Start, Continue, or Boot. The N command returns the processor to program I/O mode for one VAX instruction and then passes control back to the console mode.

When the processor is halted and the console is disabled, neither VAX macrocode nor microcode takes any observable action. In a standard VAX 8200 system you can enable the console mode by rotating the upper key switch on the control panel to the Enable position. Bit ⟨29⟩ in the PCntl CSR is a read-only bit that reflects the position of the upper key switch. See Table 3-1 in Chapter 3 for a list of the switches, corresponding module I/O pins, and PCntl CSR bits. Table 4–1 lists four PCntl CSR bits related to the console. See Appendix E for a complete description of this register.

Table 4–1: PCntl CSR Bits Related to the Console

Module I/O Pin	PCntl CSR Bit	Function
D51	⟨31⟩	Restart/Halt power-up option bit
D52	⟨30⟩	Physical/Logical console selection bit
D53	⟨29⟩	Console secure/enable selection bit
—	⟨10⟩	RXCD logical console interrupt enable control bit, set or cleared by software

4.2 Console Entry

The KA820 module enters the console mode in response to any of the following conditions:

1. From program I/O mode with the console enabled (PNL CNSL ENB H on module I/O pin D53 is true)
   - You type CTRL/P on the console terminal, halting the primary processor; or

- The logical console for an attached processor sends CTRL/P to the attached processor's RXCD Register. The logical console function on the attached processor is enabled. PNL CNSL LOG H, module I/O pin D52, is true; PNL CNSL ENB H, module I/O pin D53, is true; and CNSL INTR ENBL, PCntl CSR bit ⟨10⟩, is set; or
- A VAX error halt occurs or a VAX HALT instruction is executed in kernel mode. The lower key switch is in the Halt or Update position (PCntl CSR bit ⟨31⟩ is 1). Or the switch is in the Auto Start position (PCntl CSR bit ⟨31⟩ is 0) and a bootstrap attempt fails.

2. From power-up with the console enabled
   - Self-test fails.
   - Power-up initialization microcode detects a hardware error.
   - The lower key switch is in the Update or Halt position (PCntl CSR bit ⟨31⟩ is 1).
   - The lower key switch is in the Auto Start position (PCntl CSR bit ⟨31⟩ is 0) and a bootstrap attempt fails.

4.2.1 Halt Codes

When the processor halts and enters the console mode, the microcode displays a halt code on the console terminal, followed by the contents of the Program Counter (PC) and the console prompt, >>>. Microcode also deposits the halt code in the Argument Pointer (AP) for use by software, if and when the system is restarted or booted. Table 4–2 lists the halt codes and their meanings.

Table 4–2: Halt Codes

Halt Code	Meaning
?01	Self-test completed successfully.
?02	Console halt: The processor received a CTRL/P or N command from the enabled console source. This could be serial-line unit 0 or a VAXBI node acting as the logical console. The console is enabled.
?03	Power-fail restart.
?04	Interrupt stack not valid: The processor tried to save state on the interrupt stack and found the stack invalid or inaccessible.
?05	CPU double-error halt: The processor tried to report a machine-check error to the operating system and a second machine-check occurred.
?06	HALT instruction executed: The processor executed a VAX HALT instruction while in kernel mode.
?07	Invalid System Control Block (SCB) vector: The vector has bits ⟨1:0⟩ both set.

(Continued on next page)

Table 4–2: Halt Codes (Cont.)

Halt Code	Meaning
?08	No user-writeable control store (WCS): Software tried to use WCS by setting the value 2 in bits ⟨1:0⟩ of a system control block (SCB) vector. However, there is no user WCS on the KA820.
?0A	CHM from interrupt stack: The processor executed a change mode instruction when the interrupt stack bit, bit ⟨26⟩, was set in the Processor Status Longword (PSL).
?0C	System control block (SCB) read error: A hard memory error occurred when the processor tried to read an exception or interrupt vector.

Example 4–1 shows a sample console message following entry to the console mode.

```
?02
     PC = 00000200
>>>
```

Example 4–1: Sample Console Output Following Entry to the Console Mode

This message says that a CTRL/P command stopped the processor with the program counter pointing to address 0000 0200 (hex).

4.3 Console Commands

The following characters have special meaning for the KA820 console microcode.

- ⟨BREAK⟩ — Increment the console baud rate
- B — Boot
- C — Continue
- D — Deposit
- E — Examine
- H — Halt
- I — Initialize the processor
- N — Next
- S — Start
- T — Test
- T/M — Test with menu

- X — Binary load and unload
- Z — VAXBI forward
- ! — Comment
- ⟨ESC⟩ — Forward the next character without interpreting it.
- ⟨CTRL/P⟩ — Halt and enter console mode
- ⟨CTRL/S⟩ — Stop console output
- ⟨CTRL/Q⟩ — Restart console output
- ⟨CTRL/U⟩ — Abort the current command line and the current command

You can use these commands only when the KA820 module is in the console mode and the console is enabled. Type CTRL/P on the console terminal to enter the console mode. The Z, ⟨ESC⟩, and ⟨BREAK⟩ commands are new with the KA820 console. However, some console commands used on other VAX processors are not implemented on the KA820 module; they include:

- F — Find
- U — Unjam
- L — Load
- M — Microstep
- R — Repeat
- S — Set
- @
- ⟨BACKSPACE⟩
- ⟨DELETE⟩

Although specific rules apply to each console command, the following guidelines hold for all console commands:

1. You must abbreviate all commands (except T/M) to the first character. This character should immediately follow the console prompt, ›››.
2. Console microcode ignores all invalid, unrecognized, and unsupported commands, echoing them with the beep (bell) character on the terminal.
3. Console microcode parses all commands as you type them, ignoring invalid characters and echoing them with beep on the terminal.
4. Console microcode takes no action on a command line until you terminate it with a carriage return. Carriage return is echoed as carriage return and line feed. ⟨CTRL/P⟩, ⟨CTRL/S⟩, ⟨CTRL/Q⟩, ⟨CTRL/U⟩, ⟨BREAK⟩, and ⟨ESC⟩ are exceptions; microcode responds to these commands without waiting for a carriage return.

5. You can enter commands with uppercase or lowercase characters. The console echoes your input in the case you use, but it always responds in uppercase.
6. You must use one space to separate the parts of a command. Multiple spaces and tabs are invalid, and the console microcode echoes them with beep.
7. Console microcode interprets all numbers in commands and responses as hexadecimal values: addresses, data, and counts.
8. You can type valid qualifiers immediately following any part of the command: command character, address, data, or count.
9. You must abbreviate the symbolic address PSL to P.
10. Console microcode does not check the ranges for addresses or data. Values that are too small are extended on the left with zeros. Values that are too big are truncated.
11. You can abort any console command by typing CTRL/P or CTRL/U.

Table 4–3: Symbols Used in Console Command Descriptions

Symbol	Meaning
[ ]	An expression enclosed in square brackets is optional.
⟨ ⟩	Angle brackets enclose category names, such as ⟨address⟩ or ⟨qualifier⟩.
⟨count⟩	An 8-digit hexadecimal count. You can omit leading zeros.
⟨address⟩	An address argument. You can omit leading zeros. The command descriptions explain what address types are valid for each command.
⟨data⟩	A numeric argument. You can omit leading zeros.
⟨qualifier⟩	A command modifier. The command descriptions identify valid qualifiers for each command.
RET	Carriage return. Use RET to terminate each command except control character commands, ⟨ESC⟩, and ⟨BREAK⟩.

4.3.1 Change Console Baud Rate Command (⟨BREAK⟩)

⟨BREAK⟩

With the KA820 microcode in the console mode, you can set the baud rate for serial-line unit 0 by pressing the BREAK key on the console terminal. Microcode makes eight baud rates available:

150 baud
300 baud
600 baud

1200 baud
2400 baud
4800 baud
9600 baud
19200 baud

Each time you press BREAK, the console microcode briefly lights the control-panel RUN light (PNL RUN LED L, module I/O pin D49), increments the baud rate to the next higher speed, and in the new baud rate prints on the console terminal:

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

If these characters do not appear or they are garbled on the terminal, the KA820 module is not matched to the baud rate for your terminal. Keep typing BREAK till you find the right speed and the console prompt symbol is ungarbled. DIGITAL ships the KA820 module with the default baud rate for serial-line unit 0 set at 1200. The default baud rate used on power-up is set according to data stored in byte 6 in the EEPROM customer option space (see Table 4–4). Four other considerations apply to use of the BREAK key:

1. When the console microcode detects <BREAK>, it aborts the current command line and stops parsing or executing any command in progress.
2. Microcode sets the transmit and receive baud rates to the same value.
3. Changing the baud rate by pressing BREAK does not change the default baud rate stored in the EEPROM. The console will go back to the default baud rate after a power-down/power-up sequence.
4. You cannot forward <BREAK> to another processor with the Z command.

4.3.2 Boot Command (B)

```
B[<qualifier>] [<ddxn>] RET
```

The B command lets you boot the system. In response to this command, microcode loads and starts the software, taking the following steps:

1. Initialize the KA820 processor. This includes loading bootstrap code from the EEPROM into the boot RAM.
2. Load General Purpose Register R3 with the boot device specification <ddxn>. The boot code in the boot RAM compares the device specification with ten possible values in the EEPROM. If there is no match, microcode prints ?44 on the console terminal.
3. Load General Purpose Register R5 with the data (specified in the command qualifier) to be passed to the bootstrap software.
4. Initialize the system, polling the VAXBI bus and sizing memory.
5. Search for a 64K-byte block of good memory.

6. Invalidate the cache.
7. Start executing the VAX bootstrap code in the boot RAM if microcode has found a 64K-byte block of good memory.

The qualifier for the B command is optional. It takes the form:

/R5:⟨data⟩

where ⟨data⟩ is a 32-bit value (boot parameter) to be passed to the bootstrap software. Appendix I lists the software boot control flags used by the VMB primary bootstrap code and the VMS operating system. Microcode loads the data into R5. If you do not specify a qualifier, microcode loads R5 with 0.

The boot device specification is optional as well. It takes the form:

⟨ddxn⟩

where

dd	identifies the boot device type.
x	identifies the VAXBI node number to which the boot device is attached.
n	identifies the unit number of the boot device.

If you specify the boot device, you must use all four characters. Microcode loads the ASCII equivalent of these characters in R3. Note that this format is not the same as the device specification format for the VMS operating system.

If you do not specify the boot device, microcode loads 0 into R3 and selects the default boot device. This is the first of ten devices identified in the EEPROM beginning at location 2009 8180.

If microcode finds a 64K-byte block of good memory, it adds 200 (hex) to the address of the first page of the block and loads this value in the Stack Pointer (SP). It then transfers control to VAX bootstrap code in the boot RAM at address 2009 0104.

If microcode cannot find a 64K-byte block of good memory, it sends the error message ?43 to the console terminal and prompts for the next command. Example 4–2 shows two successful B commands and one failure.

```
>>>B                                  ! Boot using the default device.
0 . . . 4 5 . . . . . . . . . F       ! Nodes 0, 4, 5, and F are
                                      !   present on the VAXBI bus.
00200000                              ! System memory size in hex:
                                      !   two megabytes.
                                      ! Microcode loads R3 with 0 and R5
                                      !   with 0.

>>>B/R5:10 DU51                       ! Boot using device DU51.
0 . . . 4 5 . . . . . . . . . F       ! VAXBI node configuration.
00200000                              ! System memory size in hex.
                                      ! Microcode loads 4455 3531 (hex),
                                      !   the ASCII equivalent of DU51,
                                      !   into R3, and loads 10 (hex)
                                      !   into R5 (specifying the VAX
                                      !   Diagnostic Supervisor).
```

```
>>>B                                  ! Boot using the default device.
0 . . . 4 5 . . . . . . . . . . F    ! VAXBI node configuration.
00200000                             ! System memory size in hex.
                                     ! Microcode loads R3 with 0 and R5
                                     !   with 0.
?43                                  ! Microcode fails to find a
                                     !   64K-byte block of good memory.
   PC = 2009017F                     ! Halted at this location. The
                                     !   processor has not started
                                     !   executing bootstrap code
                                     !   from the EEPROM.
>>>                                  ! Console prompt.
```

Example 4–2: Representative Boot Commands

4.3.3 Continue Command (C)

C(RET)

The C command lets you continue executing software at the point where it has been stopped. In response to this command, microcode passes control to VAX macrocode. Instruction execution begins at the address specified in the Program Counter. Continue microcode invalidates the cache but it does not initialize the processor before starting instruction execution.

4.3.4 Deposit and Examine Commands (D and E)

D[⟨qualifiers⟩] ⟨address⟩[⟨qualifiers⟩] ⟨data⟩[⟨qualifiers⟩](RET)

E[⟨qualifiers⟩] [⟨address⟩] [⟨qualifiers⟩](RET)

The D and E commands let you write and read data in memory and registers almost anywhere in the computer system (exceptions include the VAXBI node private spaces on other nodes).

The KA820 microcode divides address space into six categories:

1. Physical addresses
2. Virtual addresses
3. Internal processor register addresses
4. General purpose register addresses
5. EEPROM customer options addresses
6. M chip internal registers

In response to the D command, microcode deposits the data in the address specified. You must specify an address and some data with the D command; if you omit either, microcode displays the error message ?44 on the terminal.

In response to the E command, console microcode reads the contents of the address you specify. The console terminal displays a character describing the address space (P, I, G, E, or M), the address, and the data. The address argument is optional in the E command.

The D and E commands take the same qualifiers (except that the D command does not take the /M qualifier). You can type the qualifiers in any order and in any of the positions shown in the command format. Each qualifier specifies a data size or an address space.

Data size qualifiers:

/B — The data size is byte.

/W — The data size is word.

/L — The data size is longword.

Address space qualifiers:

/P — The address space is physical memory. You can use size qualifiers /B, /W, or /L.

/V — The address space is virtual memory. If memory mapping is not enabled, the microcode treats the address as physical. When you examine a virtual memory location (E/V) and memory mapping is enabled, microcode displays the physical address corresponding to the virtual address you request. You can use the size qualifiers /B, /W, or /L.

/I — The address space is the set of internal processor registers (IPRs) accessible to software with MTPR and MFPR instructions. See Appendix F. The size is longword, and the microcode ignores any size qualifier specified.

/G — The address space is the set of general purpose registers R0 — R15. The size is longword, and the microcode ignores any size qualifier specified. The address you specify must be in the range 0 to F (hex).

/E — The address space is the customer option section of the EEPROM (see Table 4–4). The size is byte, and the microcode ignores any size qualifier specified. If you use the /W or /L qualifier with the E/E command, the console terminal will display 2 or 4 bytes, but only the low order byte is valid. You can write data in 24 (decimal) EEPROM locations if the lower key switch on the control panel is in the Update position (PNL ENB WT EEPROM H, at module I/O pin D50, is true). The address you specify must be in the range 0 to 17 (hex). If the address is out of range or the lower key switch is not pointing to Update (PNL ENB WT EEPROM H, at module I/O pin D50, is false), the microcode displays the error message ?47 on the terminal.

When you write data in the EEPROM, microcode waits for 10 milliseconds following each D command to allow the EEPROM to load the data. Note that you can examine these EEPROM locations even when the lower key switch is not in the Update position (when PNL ENB WT EEPROM H, at module I/O pin D50, is false). Table 4–4 shows the EEPROM customer option section addresses accessible with the D/E and E/E commands.

Table 4–4: EEPROM Customer Option Section Addresses Accessible with the D/E and E/E Commands

Address	Function	Implementation
0–2	Reserved to DIGITAL	
3	RCX50 self-test enable	Bit ⟨4⟩ (0 = enable, 1 = disable) Bit ⟨3⟩ must be 1 Bits ⟨7:5;2:2⟩ must be 0
4	Logical console node ID	Bits ⟨3:0⟩ identify the VAXBI node number of the logical console Bits ⟨7:4⟩ must be zero
5	Reserved to DIGITAL	
6	Default console baud rate	Bits ⟨7:0⟩ = 30 – 150 baud 31 – 300 baud 32 – 600 baud 33 – 1200 baud 34 – 2400 baud 35 – 4800 baud 36 – 9600 baud 37 – 19200 baud
7	F chip disable BTB disable Cache disable	Bit ⟨0⟩ (1 = disable, 0 = enable) Bit ⟨1⟩ (1 = disable, 0 = enable, must be zero) Bit ⟨2⟩ (1 = disable, 0 = enable) Bits ⟨7:4⟩ must be zero
8–17	Unused	16 bytes

/M	—	The address space is the set of 64 CPU internal registers in the M chip. The size is longword, and the microcode ignores any size qualifier specified. You can use the E/M command following a double-error halt to read the contents of the stack stored in the M chip registers. Table 5–3 in Chapter 5 lists the contents of the M chip registers that contain useful information on a CPU double-error halt. You cannot examine M chip register 1F (hex). This is the Processor Status Longword Temporary Register. If you try, the console will respond with the error message ?45. The /M qualifier is available only with the E (examine) command. It is not available in combination with the D (deposit) command, and you cannot write data in the M chip registers.

The qualifiers for the D and E commands are optional. The default address space is physical, the data size is longword, and the default address is 0 under three conditions:

1. After processor initialization
2. After entry into console mode
3. After execution of an N command

Otherwise the address space and data size used in the last D or E command are the defaults, and the default address is the last address plus the last data size used in a D or E command.

You can use the symbolic address P to deposit or examine data in the Processor Status Longword (PSL). The size is longword, and the microcode ignores any size qualifier specified. Following a D command to the PSL, the default address is set to the PSL for a subsequent E command. Depositing data in the PSL may leave the processor in an unpredictable state when it returns to program I/O mode. Example 4–3 shows D and E commands in console dialog.

NOTES

The default console baud rate is initially set to 1200.

Do not change byte 7; bits ⟨2:0⟩ of byte 7 should always be 0.

```
>>>D/L/P 5050 35353535                  ! Deposit 35353535 (hex) at
                                        !  physical address 0000 5050.
>>>E/L/P 5050                           ! Examine the longword at address
                                        !  0000 5050.
      P    00005050    35353535         ! Physical address 0000 5050
                                        !  contains the data 35353535
                                        !  (hex).
>>>E                                    ! Examine the next location.
     P    00005054    00FF00FF

>>>D/W/P 5054 0607                      ! Deposit 0607 (hex) at address
                                        !  0000 5054.

>>>E/B/P                                ! Examine the byte at physical
                                        !  address 0000 5056.
     P    00005056    FF

>>>E/L/V 0204                           ! Examine the longword at virtual
                                        !  address 0000 0204.
     P    00030404    69696969         ! The physical address is
                                        !  0003 0404.

>>>E/G 2                                ! Examine General Purpose
                                        !  Register R2.

     G    00000002    00050005

>>>E/E 6                                ! Examine byte 6 in the customer
                                        !  options section of the EEPROM.
     P    2009817C    33
>>>D/E 6 32                             ! Set the default console
                                        !  baud rate to 600. The lower
                                        !  key switch must be in the
                                        !  Update position.
>>>
```

Example 4–3: Sample Console Dialog Using the D and E Commands

4.3.5 Halt Command (H)

H(RET)

The H command has no effect on the processor. When the KA820 module is in console mode, it is already halted. When the processor is running in program I/O mode, CTRL/P halts the processor. The console H command merely provides compatibility with other VAX consoles. In response to the H command, console microcode prints the contents of the Program Counter on the terminal:

```
>>>H
      PC = <address>
>>>
```

4.3.6 Initialize Command (I)

I(RET)

The I command lets you put the processor in a known state, ready to execute VAX instructions. In response to this command, the microcode performs the processor initialization sequence described in Section 3.3.2 in Chapter 3. It does not initialize the computer system, and no other VAXBI node is affected.

4.3.7 Next Command (N)

N(RET)

The N command lets you step through VAX macrocode one instruction at a time. In response to this command, console microcode executes the VAX instruction at the address currently contained in the Program Counter. Execution then stops and the console displays:

```
?02
      PC = <address>
>>>
```

The PC displayed is the address of the next VAX instruction.

4.3.8 Start Command (S)

S [<address>](RET)

The S command lets you start program execution from the console, beginning where you choose. The KA820 module begins executing VAX instructions at the address you specify. The address argument is optional. If you omit it, the Program Counter identifies the default starting address.

The S command is equivalent to a D command followed by a C command, where the D command deposits an address in the Program Counter. Start microcode invalidates the cache, but it does not initialize the processor before starting instruction execution.

4.3.9 Test Command (T)

T(RET)

The T command lets you test the KA820 module from the console. In response to this command, console microcode executes the slow self-test described in Chapter 3, Section 3.2. The configuration of jumpers on the PCM module has no effect on the tests executed. First, microcode sets the VAXBI RESET bit, bit ⟨28⟩ in the PCntl CSR Register, to start a power-down/power-up sequence. This forces the control-store microaddress to 0000.

Before executing the slow self-test, the processor loads primary control-store patches from the EEPROM and sets the default baud rate on serial-line unit 0. If the KA820 module passes the slow self-test, the microcode prints #ABCDEFGHIJK.MN# on the terminal and performs power-up initialization, processor initialization, and system initialization. Self-test prints a dot instead of a letter to indicate a missing or disabled KA820 module component. If the KA820 module fails the slow self-test, the microcode prints only letters corresponding to the tests passed.

Example 4–4 shows the console output in response to a T command, when the KA820 module passes the slow self test.

```
>>>T
#
#ABCDEFGHIJK.MN#
0 . . . 4 5 . . . . . . . . . F
00200000
?01                                    ! Self-test passed.
      PC = 22222222
>>>
```

Example 4–4: Console Output Showing a Successful Slow Self-Test

In this example VAXBI nodes 0, 4, 5, and F are installed and have passed self-test. VAXBI memory contains 200000 (hex) bytes. The program counter points to address 22222222.

Example 4–5 shows console output when the processor fails the slow self-test.

```
>>>T
#
#ABC
?40
      PC = 22222222
>>>
```

Example 4–5: Console Output Showing a Slow Self-Test Failure

In this example test D fails, indicating an error in the M chip (see Table 3–3 in Chapter 3). The error code ?40 confirms the self-test failure.

4.3.10 Test with Menu Command (T/M)

T/M(RET)

The T/M command lets you boot software from a device designated in the EEPROM. The ASCII code for this device is loaded at address 2009 C008.

Typing the T/M command is equivalent to typing B/R5:11. The microcode does not perform a power-down/power-up sequence or self-test. Instead, the microcode:

1. Loads General Purpose Register R3 with 0.
2. Loads General Purpose Register R5 with 11 (hex).
3. Performs system initialization.
4. Searches for a page-aligned 64K-byte block of good memory.
5. Passes control to the VAX boot code in the boot RAM at physical address 2009 0104 (hex).

4.3.11 Binary Load and Unload Command (X)

X[⟨qualifier⟩] ⟨address⟩ ⟨count⟩(RET)⟨checksum⟩

The X command allows an automatic load device to communicate with the console microcode. This command can be used in the DIGITAL manufacturing process, but it is not suitable for use by an operator using a keyboard. In response to the X command, console microcode reads or writes the number of bytes you specify at the address you specify. The address space is always physical.

The optional /P qualifier makes the KA820 console compatible with other VAX consoles.

Bit ⟨31⟩ of the count is a load or unload signal. If bit ⟨31⟩ is clear, microcode loads data from serial-line unit 0 into memory. If bit ⟨31⟩ is set, microcode reads data from memory and sends it to serial-line unit 0. The remaining bits in the count tell how many bytes to load or unload.

The checksum following the carriage return is an 8-bit command checksum that includes the ASCII value of the carriage return. If the sum of the command characters plus the command checksum equals zero, the console issues a prompt symbol and sends or receives data. If the command checksum is not zero, the console issues the error message ?48 and an input prompt. The command checksum feature lets you escape from inadvertent entry into the binary load mode from the console terminal.

On a binary load command the console microcode accepts the number of bytes specified in the count plus an additional byte containing the data checksum. The console loads all characters, including CTRL/P, CTRL/S, and CTRL/Q, without interpreting them. If the 8-bit sum of the data plus the data checksum is not zero, the microcode responds with the ?48 error message.

On a binary unload command the console microcode sends the number of bytes specified in the count plus an 8-bit data checksum that the automatic load device can use to verify the data it receives. The console responds to CTRL/P, CTRL/S, and CTRL/Q commands from serial-line unit 0 during the unload function.

4.3.12 VAXBI Forward Command (Z)

```
Z <node-number>(RET)
```

The Z command lets you forward characters from the console terminal connected to serial-line unit 0 on the primary processor (VAXBI slot K1J1) to an attached processor on the VAXBI bus. The node-number argument is a hexadecimal character (0 — F) that identifies the attached processor.

In response to the Z command, the console microcode enters the forwarding console mode and sends all characters typed at the console terminal (except (CTRL/P) and (ESC)) to the RXCD Register on the node you select. The selected node echoes all console commands by writing to the RXCD Register in the primary processor.

Console microcode on the primary processor may find that the remote node's RXCD Register is busy for more than one second. In this case, the primary processor overwrites the character in the remote node's RXCD Register by writing first with the Busy bit off and then with the Busy bit on. The two write functions generate a new interrupt on the remote node. This procedure ensures that a remote processor will respond to a CTRL/P command, even if it is hung.

In multiprocessor systems, the primary processor serves as the logical console for attached processors. The physical/logical console selection signal (CNSL LOG, PCntl CSR bit ⟨30⟩) is 0 by default on the primary processor, selecting serial-line unit 0 as the physical console source. In attached processors, the CNSL LOG signal is 1 by default, enabling another processor to serve as the logical console. The EEPROM in each attached processor contains the node number (ID) of the processor that can serve as the logical console (byte 4 of the customer option section of the EEPROM).

You can exit from the forwarding console mode to the local console mode by typing (CTRL/P) on the console terminal. To send a CTRL/P command to an attached processor you must type (ESC) first. ⟨ESC⟩ unconditionally forwards the next character to the attached processor, even a ⟨CTRL/P⟩ command or another ⟨ESC⟩ character. When you type (ESC) (CTRL/P), the microcode forwards CTRL/P, halting the attached processor. The attached processor then enters the console mode, ready to accept other console commands. Figure 4–1 shows the function of the ⟨ESC⟩ command in combination with the Z command. Notice that after forwarding a character, microcode checks for the presence of a character in its own RXCD Register.

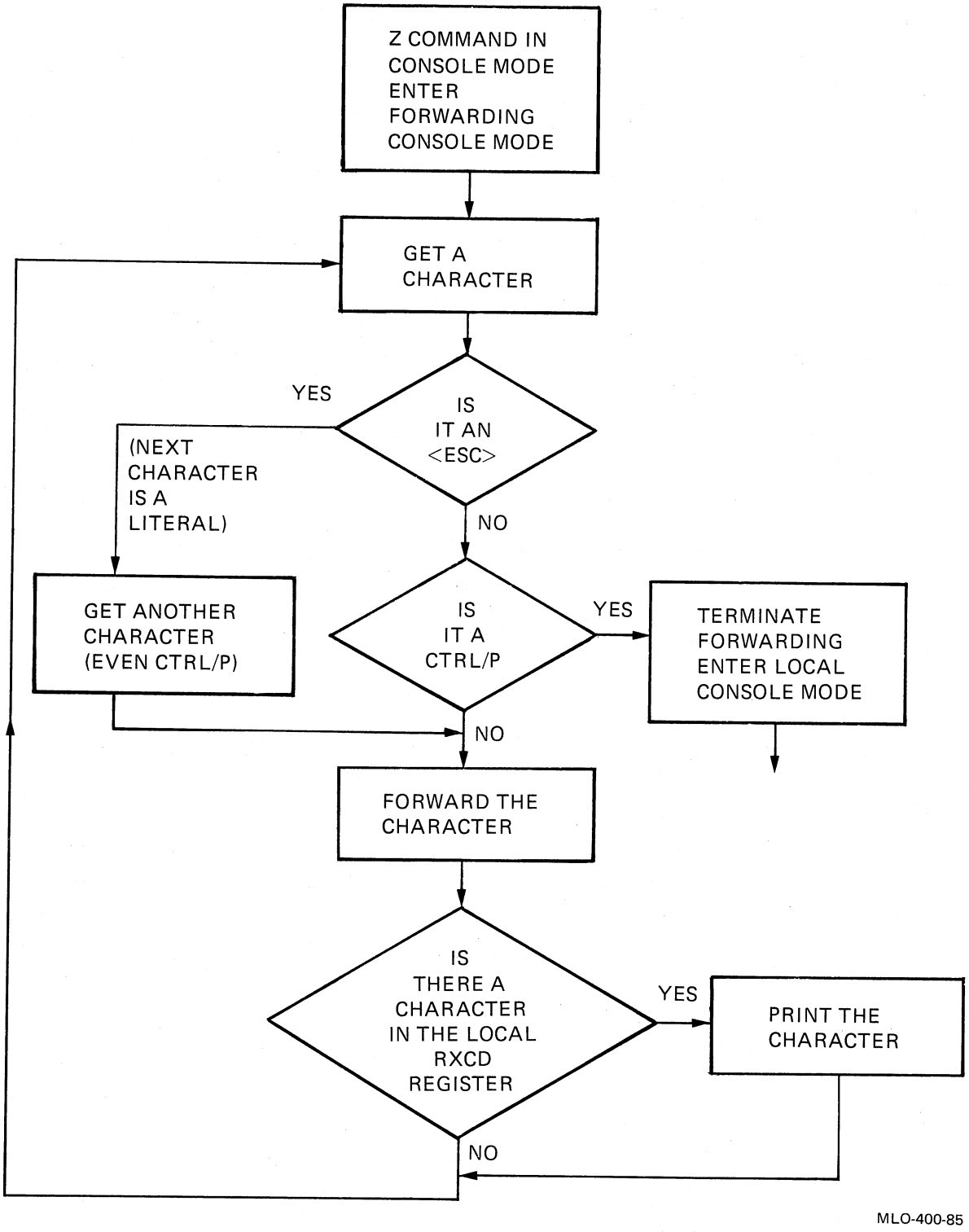


Figure 4–1: Use of ⟨ESC⟩ with the Z Command

4.3.13 Console Comment Command (!)

```
![<comment characters...>][RET]
```

You can enter comments in your console dialog if you use the comment command. Console microcode echoes the characters typed after the exclamation point, but takes no action. This is handy if you plan to save the console output displayed on a hard-copy terminal.

4.3.14 Enter Console Mode Command (<CTRL/P>)

```
<CTRL/P>
```

You can halt the processor and enter the console mode by typing [CTRL/P], if the console is enabled (PCntl CSR bit <29> is 1). The processor responds by displaying the halt code ?02, the contents of the program counter, and the console prompt symbol on the terminal. See Sections 4.3.15 and 4.6 for information on sending a CTRL/P command to an attached processor.

4.3.15 Forward Next Character Command (<ESC>)

```
<ESC>
```

<ESC> works in combination with the Z command. When the console is in forwarding console mode and you type [CTRL/P] on the console terminal, the microcode normally terminates the forwarding console mode and enters the local console mode. You can prevent this response by preceding the CTRL/P command with the <ESC> command. <ESC> forces the console microcode to forward the following character without interpreting it.

4.3.16 Stop Console Output Command (<CTRL/S>)

```
<CTRL/S>
```

You can stop the flow of console output to the console terminal by typing [CTRL/S]. In response to CTRL/S, console microcode also discards all input characters except CTRL/P, CTRL/Q, and CTRL/U. It echoes each discarded character with beep.

4.3.17 Restart Console Output Command (<CTRL/Q>)

```
<CTRL/Q>
```

You can allow the stopped flow of console output to resume by typing [CTRL/Q].

4.3.18 Abort Command Line Command (⟨CTRL/U⟩)

```
⟨CTRL/U⟩
```

If you make a mistake in a command line, you can start again by typing CTRL/U before you press the carriage return. Console microcode then ignores the entire line and displays a fresh prompt symbol.

4.4 Console Error Codes

When the console microcode tries and fails to execute a function, it displays an error code on the console terminal. Like the halt codes, each console error code begins with a question mark. Table 4–5 lists these codes and their meanings.

Table 4–5: Console Error Codes

Console Error Code	Meaning
?40	Self-test failure.
?41	BI AC LO L timeout: BI AC LO L not deasserted within 10 seconds of power-up.
?42	A warm start or bootstrap attempt has failed because the bootstrap-in-progress flag is already set.
?43	Microcode cannot find a 64K-byte block of good memory while trying to bootstrap the system.
?44	Unrecognized console command or boot device specification.
?45	Memory reference not allowed. A console D, E, or S command required a memory reference, and a translation-not-valid fault or access violation occurred.
?46	Invalid access to an internal processor register.
?47	Invalid access to the EEPROM. Either the address specified by a D/E or E/E command is outside the valid range of 0 to 17 (hex), or writing to the EEPROM is inhibited (PNL ENB WT EEPROM H, I/O module pin D50 is false). Microcode detects the second case with an automatic read-after-write check to the location accessed.
?48	Incorrect checksum of command or data in an X command.
?4A	Error while loading primary control-store patches from the EEPROM.
?4B	Checksum error in a boot program in the EEPROM.
?4C	Hardware error. This may be an MIB or DAL parity error or a VAXBI error. Or it may result from reference to nonexistent memory or a nonexistent device.

4.5 Loading Control-Store Patches from the Console

In normal KA820 module operation microcode loads primary control-store patches from the EEPROM before beginning self-test, and bootstrap software loads secondary patches from a bootstrap device. Section 3.4.2.3 of Chapter 3 explains how to write software that loads control-store patches by manipulating the internal processor registers WCSL, WCSA, and WCSD. However, you can also use the console to manipulate these registers and load control-store patches. This process requires three steps:

1. Write the block of patches to main memory, using the console D or X command, according to the format shown in Figure 3–5. The block must include the number of patches, the beginning CAM address, and a checksum, as shown there.
2. Load the block of patches into the control-store RAM by depositing the main-memory starting address of the block in the WCSL Register (IPR 2E). The address must be physical.
3. Check the loaded patches by reading them back from control store. For each microword patch:
   - Deposit the address, bits ⟨0:13⟩ of the patched control-store ROM location, in the WCSA Register (IPR 2C), bits ⟨21:8⟩.
   - Examine the WCSD Register (IPR 2D) to retrieve microword bits ⟨8:0, 39, 17:38⟩, in that order.
   - Examine the WCSA Register to retrieve microword bits ⟨16:9⟩ in the first byte.

Figure 3–7 in Chapter 3 shows the bit configuration of the WCSA and WCSD registers. Example 4–6 shows the console output during a control-store patch load sequence.

```
>>>D/P F000 43214321                ! Deposit a block of control-store
>>>D/P F004 87658765                !  patches in main memory.
  .
  .
  .
>>>D/I 2E F000                      ! Transfer the entire block of
                                    !  patches from main memory to the
                                    !  control-store RAM, by writing
                                    !  to the WCSL Register.

>>>D/I 2C 0033F800                  ! Start the 3-command sequence to
                                    !  examine the patch for ROM
                                    !  address 07F3, by writing the
                                    !  address (inverted) in bits
                                    !  ⟨21:8⟩ of the WCSA Register.

>>>E/I 2D                           ! Examine microword bits
                                    !  ⟨8:0, 39, 17:38⟩, by reading
                                    !  the WCSD Register.
I    0000002D    EE93001F
```

```
>>>E/I 2C                                  ! Examine microword bits <16:9>, by
                                           !   reading the WCSA Register.
I    0000002C    0033F81E

>>>D/I 2C 000BF800                         ! Examine the next patch.
  .
  .
  .
>>>
```

Example 4–6: Loading and Checking Control-Store Patches from the Console

4.6 Logical Console Operation

In a multiprocessor system the primary processor can store and forward console commands for attached processors from the console mode and from the program I/O mode. With the primary processor in the local console mode, you begin the logical console session by typing the Z command.

In Example 4–7 the attached processor (node 4) is initially running in the program I/O mode, executing software. The operator halts program execution in the attached processor, invalidates a BTB entry in the attached processor, restarts the attached processor, and returns to local console mode.

```
>>>Z 4                                    ! Enter forwarding console
                                          !   mode and send subsequent
                                          !   commands to node 4.

>>>ESC CTRL/P                             ! Halt the node-4 processor.
?02                                       ! The attached processor halts
     PC = 00000700                        !   at physical address 0000 0700.

>>>D/I 3A 00109800                        ! Invalidate a BTB entry in
                                          !   the attached processor.
                                          !   See Appendix F.

>>>C                                      ! Restart the attached processor
                                          !   where it left off.

>>>CTRL/P                                 ! Return to local console
                                          !   mode on the primary processor.
```

Example 4–7: Logical Console Dialog Displayed on the Terminal

When software executing on the primary processor performs the same logical console functions, the Z and <ESC> commands are unnecessary. Example 4–8 shows the sequence of commands that the software should send, one character at a time, to the attached processor's RXCD Register, using MTPR instructions.

```
<CTRL/P>                          ! The CTRL/P character halts
                                  !   the attached processor.

D/I 3A 00109800<RET>              ! Invalidate a BTB entry in
                                  !   the attached processor.

C<RET>                            ! Restart the attached processor
                                  !   where it left off.
```

Example 4–8: Primary Processor Software Performs Logical Console Functions

Software can access the RXCD Register on the attached processor at address bb + 200, where bb is the base address of the node.

In response to the CTRL/P command, console microcode in the attached processor echoes the command and sends

```
?02
      PC = 00000700
>>>
```

to the RXCD Register on the primary processor. In response to the D/I command, console microcode in the attached processor merely echoes the command and sends the prompt symbol. Software in the primary processor must wait till it receives the console prompt before sending the next command.

Chapter 5
Handling Exceptions and Interrupts

The KA820 processor implements exceptions and interrupts according to VAX architecture specifications (see the *VAX Architecture Handbook*). Some exceptions and interrupts are specific to the KA820 processor and systems that use the VAXBI bus, while others are standard on all VAX processors. KA820 microcode, or another node on the VAXBI bus, or software can signal an event that requires attention, by generating an exception vector or an interrupt vector. The vector identifies a memory location containing a pointer. The pointer in turn locates a software service routine that can handle the event requiring attention. The microcode deals with any exception immediately by transferring control to an appropriate exception handler, but it defers each interrupt until the priority level of the current process drops below the priority level of the interrupt.

An exception occurs synchronously with respect to the process currently running; it often results from a condition caused by execution of a specific VAX instruction. Exceptions are generally repeatable, so that if a program generating an exception is run again, the exception occurs again at exactly the same point.

Traps, faults, and aborts are the three kinds of exceptions:

1. A trap exception occurs at the end of a VAX instruction, leaving the registers and memory in a consistent state. Integer-overflow is an example.
2. A fault exception occurs during execution of a VAX instruction, but it also leaves the registers and memory in a consistent state. Translation-not-valid is an example.
3. An abort exception occurs during execution of a VAX instruction in response to a serious system failure, leaving the registers and memory in an unpredictable state. Machine check is an example. Most exceptions leave the interrupt priority level at IPL 0. Machine-check and kernel-stack-not-valid exceptions, however, require the highest priority (they require immediate attention with no interruptions) and therefore raise the IPL to 1F (hex).

An interrupt occurs asynchronously with respect to the process currently running. Completion of an I/O function and power failure are examples of events that interrupt normal process execution. Microcode assigns an interrupt priority level for each interrupt vector. Software generated interrupts use the lowest 15 interrupt priority levels (1 to F hex). Hardware and micro-

code generated interrupts take the high interrupt priority levels (10 to 1F hex). VAXBI nodes can use four interrupt levels (BI INTR4 to BI INTR7), corresponding to IPL 14 to IPL 17. Table 5–1 lists the types of interrupts and exceptions and the corresponding interrupt priority levels.

Table 5–1: Interrupt Priority Levels on the KA820 Module

Interrupt Priority Level (hex)	Interrupt or Exception Source Type
1F	Software Machine-check exception Kernel-stack-not-valid exception Power-up microcode
1E	BI AC LO L becomes true
1D to 1B	Not used
1A	Corrected read data (CRD)
19 and 18	Not used
17	BI INTR7
16	BI INTR6 ICCS Interval Timer
15	BI INTR5
14	BI INTR4 RXCD IPINTR RCX50 Serial-line unit 0 Serial-line unit 1 Serial-line unit 2 Serial-line unit 3
13 to 10	Not used
F to 1	Software interrupt levels 15 to 1 (decimal)

In cases where two or more interrupts occur at the same level, priority follows the order shown in Table 5–1. For example, BI INTR4 and serial-line unit 0 both use IPL 14, but BI INTR4 takes precedence. When microcode responds to an interrupt, it sets the IPL shown in the table to block subsequent interrupts at or below that level until the handler for the interrupt is finished.

NOTE

Software can mask interrupts by writing a value specifying an interrupt level to the Interrupt Priority Level Register (IPR 12 hex) with the MTPR instruction. Interrupts with levels at or below the

value written are masked. The value 1F (hex) masks all interrupts with the exception of interrupts for the RXCD Register and serial-line unit 0. Microcode reschedules the RXCD Register and serial-line unit 0 interrupts at IPL 14.

Operating system software includes routines that handle exceptions and interrupts. If you do not use a standard operating system, you must supply your own exception and interrupt condition handlers.

5.1 System Control Block

KA820 microcode and system software cooperate to maintain the system control block, a structure made up of two or more pages of vectors, each beginning on a page boundary. The System Control Block Base Register (IPR 11 hex) points to the first vector in the block, vector 0. The vectors in the system control block are offsets from the base address.

Microcode on the KA820 module defines the first half page of vectors (0 to FC hex). Software defines the vectors from 100 to 3FFC (hex). The VMS operating system software for the KA820 module assigns the vectors from 100 to 1FC (hex) to native VAXBI devices, by node number, leaving vectors 200 through 3FFC (hex) available for I/O controllers and devices. Table 5–2 lists the vector assignments in the system control block. A dash (—) means that microcode leaves the interrupt priority level unchanged.

Table 5–2: System Control Block Vector Assignments on the KA820 Module

Vector (hex)	Function	Exception or Interrupt	Interrupt Priority Level (hex)
0	UNIBUS or VAXBI passive release	Interrupt	14 to 27
4	Machine-check abort	Exception	1F
8	Kernel-stack-not-valid abort	Exception	1F
C	Power fail	Interrupt	1E
10	Reserved or privileged instruction fault	Exception	—
14	XFC instruction fault (customer reserved instruction; user microcode is not supported on the KA820)	Exception	—
18	Reserved operand fault or abort	Exception	—
1C	Reserved addressing mode fault	Exception	—
20	Access-control-violation fault	Exception	—
24	Translation-not-valid fault	Exception	—
28	Trace-pending fault	Exception	—
2C	Breakpoint fault	Exception	—

(Continued on next page)

Table 5–2: System Control Block Vector Assignments on the KA820 Module (Cont.)

Vector (hex)	Function	Exception or Interrupt	Interrupt Priority Level (hex)
30	Reserved to DIGITAL	—	—
34	Arithmetic trap or fault	Exception	—
38, 3C	Reserved to DIGITAL	—	—
40	CHMK instruction trap (change mode to kernel)	Exception	—
44	CHME instruction trap (change mode to executive)	Exception	—
48	CHMS instruction trap (change mode to supervisor)	Exception	—
4C	CHMU instruction trap (change mode to user)	Exception	—
50	VAXBI bus error	Interrupt	14
54	Corrected read data (CRD)	Enterrupt	1A
58	RXCD	Interrupt	14
5C to 7C	Reserved to DIGITAL	—	—
80	Interprocessor interrupt	Interrupt	14
84 to BC	Software interrupt	Interrupt	1 to F
C0	Interval timer	Interrupt	16
C4	Reserved to DIGITAL	Interrupt	14
C8	Serial-line unit 1 receive	Interrupt	14
CC	Serial-line unit 1 transmit	Interrupt	14
D0	Serial-line unit 2 receive	Interrupt	14
D4	Serial-line unit 2 transmit	Interrupt	14
D8	Serial-line unit 2 receive	Interrupt	14
DC	Serial-line unit 3 transmit	Interrupt	14
E0 to EC	Reserved to DIGITAL	—	—
F0	RCX50 interface	Interrupt	14
F4	Reserved to DIGITAL	—	—
F8	Console terminal receive	Interrupt	14
FC	Console terminal transmit	Interrupt	14
100 to 3FFC	Vectors are defined and loaded by software	Interrupt	14 to 17

NOTES

KA820 microcode checks the values of interrupt vectors coming from the VAXBI bus. It remaps to vector 50 any vector in the range of 4 to 4C (hex) received from another VAXBI node. In this way the processor detects invalid vectors from VAXBI nodes; VAXBI node interrupt vectors should be 100 or above.

The KA820 module uses vector 0 for the passive release function. When microcode responds to a VAXBI interrupt with an IDENT transaction, the interrupting node can reply with vector 0 if the condition requiring the interrupt has passed. In addition, microcode generates vector 0 if there is a NO ACK confirmation in response to a KA820-generated IDENT transaction. System software must supply condition handling macrocode corresponding to vector 0, but little action is required. The condition handler can simply add 1 to a counter and return to the interrupted process with an REI instruction.

5.2 Machine-Check Exceptions

The KA820 module responds to nine kinds of hardware and microcode related errors as machine-check conditions:

1. BTB tag parity errors
2. Cache tag parity errors
3. BTB data parity errors
4. Cache data parity errors
5. VAXBI node or transaction errors
6. MIB bus parity errors
7. Unforeseen microcode situations
8. Interrupts with unexpected interrupt priority levels
9. Port controller timeout

When KA820 microcode discovers a machine-check error condition, it immediately aborts the current process and raises the interrupt priority level to 1F (hex) to mask all interrupts. Microcode then takes the following action:

1. Set the hardware-fault state flag to prevent the microcode from detecting other errors until it has dealt with this one. The hardware-fault state flag lights the two red LEDs on the module. Note that this flag is inaccessible to software.
2. Set the machine-check condition flag. If the flag is already set, this is a double error. Microcode halts the processor and enters the console mode (if enabled).

3. Push on the stack error information from internal registers in the M chip. The information is then available to the software.
4. Clear the hardware-fault state flag and turn off the two red LEDs on the module.
5. Pass control to the machine-check exception condition handler at the address specified by vector 4.

The condition handler must evaluate the information on the stack and then either attempt to recover from the error or halt the processor. If there is any chance of producing catastrophic results, such as corrupting the data base or producing wrong answers, software should execute the HALT instruction. If the condition handler does try to recover, it should take the following steps:

1. Restore the machine state as required.
2. Write (any value will do) to the MCESR Register (IPR 26 hex) to clear the machine-check condition flag. If the condition handler performs this instruction too soon, occurrence of a second machine-check error will cause the microcode and the condition handler to loop, unable to halt in response to the double error.

Table 5–3: Machine-Check Stack

Location in Memory (hex)	Data Available	Corresponding M Chip Internal Register	Appropriate Console Examine Command Following CPU Double-Error Halt
(SP)	Byte count on stack	—	none
(SP) + 8	Parameter 1	MTEMPB Register	E/M B
(SP) + C	Virtual Address Register contents	MTEMP13 Register	E/M 13
(SP) + 10	Virtual Address Prime Register contents	MTEMP.PSL.TEMP Register	none
(SP) + 14	Memory Address Register contents	MTEMP9 Register	E/M 9
(SP) + 18	Status word	MTEMPC Register	E/M C
(SP) + 1C	PC at failure	MTEMPF Register	E/M F
(SP) + 20	Microcode PC at failure	MTEMP10 Register	E/M 10
(SP) + 24	Current PC	—	E/G F
(SP) + 28	Current PSL	—	E P

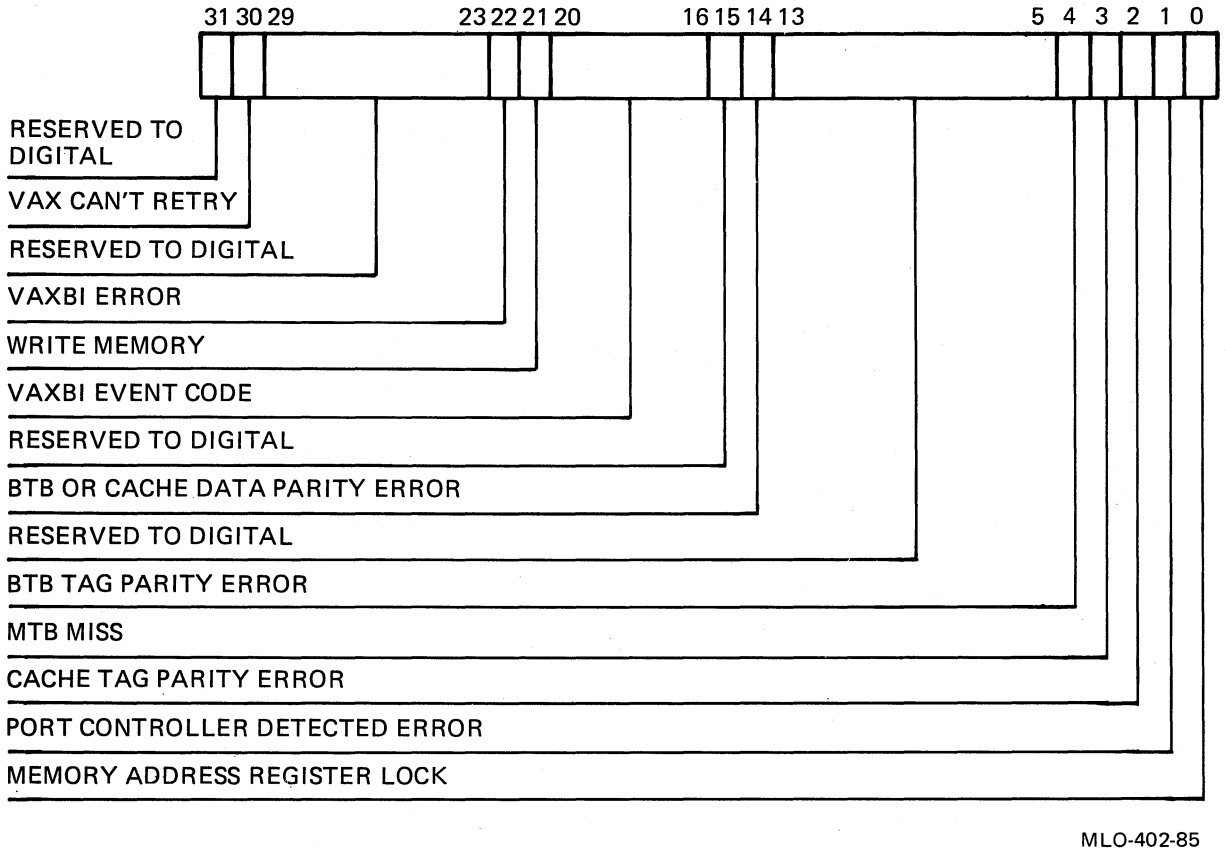


Figure 5–1: Machine-Check Status Word Bit Layout

3. Execute an REI (Return from Exception or Interrupt) instruction to give control back to the process that was interrupted. Note that if the machine-check exception indicates an MIB bus parity error, the REI instruction microcode may be unable to identify the process to which control should return.

4. Check and clear all VAXBI node Bus Error Registers.

5.2.1 Machine-Check Stack

Machine-check microcode pushes eleven longwords on the stack. Eight of these are copies of internal registers in the M chip. This information serves as a snapshot of the machine state at the time of error detection.

5.2.1.1 Byte Count, (SP) — This is the first entry on the machine-check stack. It tells how many bytes are on the stack for this error (20 hex). Note, however, that the current PC and current PSL are not included in this byte count, even though they are pushed on the stack.

5.2.1.2 Parameter 1, (SP) + 8, MTEMPB Register — On detection of a BTB tag parity error or cache tag parity error, microcode loads the parameter 1 longword with the corresponding tag.

5.2.1.3 Virtual Address Register, (SP) + C, MTEMP13 Register — This points to a virtual address that may be related to the machine-check error.

5.2.1.4 Virtual Address Prime Register, (SP) + 10, MTEMP.PSL.TEMP Register — Like the Virtual Address Register, this points to a virtual address that may be related to the machine-check error. This information is unavailable following a CPU double-error halt.

5.2.1.5 Memory Address Register, (SP) + 14, MTEMP9 Register — This points to a physical address that may be related to the machine-check error.

5.2.1.6 Status Word, (SP) + 18, MTEMPC Register — The machine-check status word often contains the most useful error information available to condition handler software. The hardware copies bits ⟨22:16, 14⟩ from the corresponding bits in the PCntl CSR. Figure 5–1 shows the bit configuration.

- Bit ⟨30⟩ VAX Can't Retry (VCR) — Microcode sets or clears this flag to tell the exception condition handler whether to restart the VAX instruction that the saved program counter ((SP) + 1C) points to. Microcode clears the VCR flag at the beginning of each instruction and at the resumption of each interrupted instruction when the first-part-done (FPD) flag is set in the processor status longword. Microcode sets the VCR flag in response to any of three conditions:

  — Reference to an address in I/O space

  — A successful write transaction to memory

  — A successful write transaction to a general purpose register

  The VCR flag may be incorrect for cache data parity errors and BTB data parity errors, for some VAXBI errors, and for errors that occur with the FPD flag set. You can retry an instruction associated with a VAXBI error if accurate address information is available on the stack. Note, however, that when a machine-check exception occurs on a write transaction to memory, the address information saved on the stack is invalid.

  If the exception condition handler decides not to retry the failing VAX instruction, it can take one of two steps:

  Log out the current process if the failing instruction belongs to a process running in the user mode or the supervisor mode.

  Execute a HALT instruction if the failing instruction belongs to a process running in the executive mode or the kernel mode.

The execution of HALT may trigger a restart (warm start).

- Bit ⟨22⟩ VAXBI Error — When the port controller detects a VAXBI error, it sets this flag, locking the write-memory flag, bit ⟨21⟩, and the VAXBI event code, bits ⟨20:16⟩. Therefore, whenever status word bit ⟨22⟩ is set, bits ⟨21:16⟩ are valid and pertinent. After handling the error, the exception condition handler should clear bit ⟨22⟩ in the PCntl CSR to unlock the write-memory flag and VAXBI event code bits, PCntl CSR bits ⟨21:16⟩.

- Bit ⟨21⟩ Write Memory — The port controller sets this flag when the KA820 module writes to memory. If status word bits ⟨22⟩ and ⟨21⟩ are both set, the machine-check condition is the result of a VAXBI write-memory error.

- Bits ⟨20:16⟩ VAXBI Event Code — When status word bit ⟨22⟩ is set, these five bits form a code that defines the VAXBI transaction that preceded detection of the error. The Memory Address Register points to the physical address that failed, unless the KA820 module was performing a write transaction on the VAXBI bus.

  When bit ⟨22⟩ is clear, the VAXBI event code is not useful error information, because it reflects whatever is happening on the VAXBI bus at the moment. Table 5–4 lists the 32 VAXBI event codes and their meanings.

- Bit ⟨14⟩ BTB or Cache Data Parity Error — When microcode reads the cache or BTB, the port controller checks the parity of the data. If the port controller detects an error, it sets status word bit ⟨14⟩. No other information describing cache and BTB data parity errors is available.

- Bit ⟨12⟩ Port Controller Timeout Error — The port controller starts a timer when it receives a command from the CPU to perform a VAXBI transaction. If the transaction has not been completed 12.8 milliseconds later, the port controller sets this bit.

- Bit ⟨4⟩ BTB Tag Parity Error — The microcode has detected a parity error while reading a BTB tag entry.

- Bit ⟨3⟩ MTB Miss — A miss in the MTB has caused the microcode to read the BTB. This information is useful to self-test microcode.

- Bit ⟨2⟩ Cache Tag Parity Error — The microcode has detected a parity error while reading a cache tag entry.

- Bit ⟨1⟩ Port Controller Detected Error — When this flag is set, it indicates that the port controller detected the error that produced the machine-check condition. Status word bits ⟨22:16⟩ and bit ⟨14⟩, copied from the corresponding bits in the PCntl CSR, are valid and pertinent.

- Bit ⟨0⟩ Memory Address Register Locked — When the Memory Address Register is locked, status word bits ⟨22:1⟩ are valid. The contents of the Memory Address Register are not valid, however, unless bit ⟨21⟩ is clear, indicating that the error did not occur during a memory write transaction.

Table 5–4: VAXBI Event Codes: Status Word Bits ⟨20:16⟩

Code (hex)	Event
0	No event (NEV)
1	Master Port Transaction Complete (MCP)
2	ACK received for Slave Read Data (AKRSD)
3	Bus Timeout (BTO); the VAXBI Error bit is set (status word bit ⟨22⟩)
4	Self-Test Passed (STP)
5	RETRY CNF Received for Master Port Command (RCR)
6	Internal Register Written (IRW)
7	Advanced RETRY CNF Received (ARCR)
8	NO ACK or Illegal CNF Received for INTR command (NICI)
9	NO ACK or Illegal CNF Received for Force-Bit IPINTR/STOP Command (NICIPS)
A	ACK CNF Received for Error Vector (AKRE)
B	IDENT ARB Lost (IAL)
C	External Vector Selected — Level 4 (EVS4)
D	External Vector Selected — Level 5 (EVS5)
E	External Vector Selected — Level 6 (EVS6)
F	External Vector Selected — Level 7 (EVS7)
10	Stall Timeout on Slave Transaction (STO)
11	Bad Parity Received During Slave Transaction (BPS)
12	Illegal CNF received for slave data (ICRSD)
13	Bus Busy Error (BBE)
14	ACK CNF Received for Non-Error Vector at Level 4 (AKRNE4)
15	ACK CNF Received for Non-Error Vector at Level 5 (AKRNE5)
16	ACK CNF Received for Non-Error Vector at Level 6 (AKRNE6)
17	ACK CNF Received for Non-Error Vector at Level 7 (AKRNE7)
18	Read Data Substitute or RESERVED Status Code Received (RDSR). The KA820 module has attempted a VAXBI memory read transaction and failed. The responding memory node has returned data with two or more wrong bits, because the error is uncorrectable. VAXBI Error, status word bit ⟨22⟩, is set.
19	Illegal CNF Received for Master Port Command (ICRMC). The KA820 module has received an illegal confirmation code in response to a VAXBI command. VAXBI Error, status word bit ⟨22⟩, is set.
1A	NO ACK CNF Received for Master Port Command (NCRMC). The KA820 module has received a NO ACK response to a command. Reference to a nonexistent memory location or an invalid I/O space may have caused the error. VAXBI Error, status word bit ⟨22⟩, is set.
1B	Bad Parity Received (BPR). This gives advance indication that a master or slave parity error has occured.
1C	Illegal CNF Received by Master Port for Data Cycle (ICRMD). The KA820 module has received an illegal confirmation code in response to a VAXBI command. VAXBI Error, status word bit ⟨22⟩, is set.

(Continued on next page)

Table 5–4: VAXBI Event Codes: Status Word Bits ⟨20:16⟩ (Cont.)

Code (hex)	Event
1D	Retry Timeout (RTO). The KA820 module has retried a transaction 4096 times consecutively without success. VAXBI Error, status word bit ⟨22⟩, is set.
1E	Bad Parity Received During Master Port Transaction (BPM). While it was the VAXBI master, the KA820 module detected bad parity on the bus. VAXBI Error, status word bit ⟨22⟩, is set.
1F	Master Transmit Check Error (MTCE). The KA820 module was the only driver on the VAXBI bus when it detected data on the bus that did not match the data sent. VAXBI Error, status word bit ⟨22⟩, is set.

5.2.1.7 Program Counter at Failure, (SP) + 1C — Microcode saves the contents of the program counter here at the moment when it detects a machine-check error. This address may point to the middle of a partially executed instruction.

5.2.1.8 MicroPC at Failure, (SP) + 20 (hex) — This identifies the address of the next microinstruction at the time of error detection. This information is not useful on a memory write transaction error.

5.2.1.9 Current Program Counter, (SP) + 24 (hex) — This longword points to the beginning of the VAX instruction that was executing at the time of the failure. If the software performs an REI instruction to retry the instruction that failed, program execution resumes at this address.

5.2.1.10 Current Processor Status Longword, (SP) + 28 (hex) — The PSL is useful to the condition handler because it defines the status of the process that was executing at the time of the machine check. The first part done flag (FPD), bit ⟨27⟩, may be especially relevant.

If the FPD flag is clear, repeating the instruction should produce the correct result. If the FPD flag is set, execution can resume in the middle of the instruction, since microcode has packed up the partially executed instruction. However, if the exception results from an MIB parity error, the microcode state is incomplete, and execution of the instruction may produce another error, regardless of the state of the FPD flag.

5.3 CPU Double-Error Halt Considerations

When microcode responds to an error condition, it sets the machine-check condition flag. If another machine-check error occurs before a macrocode machine-check handler resets this flag with an MTPR to MCESR instruction, then machine-check microcode transfers control to the CPU double-error halt handling microcode. The KA820 module then enters the console mode (if enabled), displays the CPU double-error halt code (?05), and leaves the control-panel RUN light off.

Microcode does not push error information on the stack for a CPU double error, and any information on the stack may be invalid. However, the console Examine command gives you access to most of the same information, in case you want to analyze the error.

Use the console command:

```
>>>E/M <M chip address>
```

to read parameter 1, the Virtual Address Register, the Memory Address Register, the status word, the Program Counter at failure, and the micro-PC at failure in the M chip registers. Table 5–3 lists the appropriate console Examine commands.

Use the console command:

```
>>>E/G F
```

to read the current Program Counter.

Use the console command:

```
>>>E P
```

to read the current Processor Status Longword.

When you no longer need to save the failing machine state, you can further isolate a hardware error by running self-test.

5.4 Power-Up and Console Mode Errors

An error that normally causes a machine-check exception can occur while the KA820 module is performing power-up or console functions. For example, system initialization microcode may detect an invalid confirmation code on the VAXBI bus as a response to a KA820 command. A memory node may respond with read data substitute (RDS) instead of the data required in a console examine command. Or the console operator can cause an error by trying to examine a nonexistent memory location.

Since there may be no condition handling software and no system control block set up in memory, microcode cannot deal with these errors in the normal way. Microcode sets an internal error-state flag in such situations, to prevent the machine-check exception microcode from taking control. Whenever this flag is set, control passes to the console microcode, which attempts to display an error message (generally ?4C) on the console terminal.

Chapter 6
Dedicated I/O and Memory Devices

The KA820 module incorporates four dedicated I/O devices and two dedicated memory devices on the module, and it connects directly to two off-module I/O devices. This chapter tells what you need to know about each device to write driver software that controls it.

Four serial-line units implemented with four UARTS (UART0, 1, 2, 3) in the M chip make the KA820 module directly accessible to terminals and modems. Using the serial-line units you can connect up to four terminals directly to the KA820 module without using a VAXBI slot for a communications device. UART0, on serial-line unit 0, connects to the console device (normally a hard-copy terminal).

The PCI bus connects the port controller with two on-board devices: the EEPROM and boot RAM; and it runs off the module to connect with the watch chip and the RCX50 diskette controller. Addresses for these devices lie within the VAXBI node private space. They are accessible only to the local KA820 module, so other VAXBI nodes cannot read or write them.

The PCI bus is a 16-bit asynchronous bus that multiplexes data and addresses. Although the bus carries data on all 16 bits, only bits ⟨14:1⟩ are available for addresses. Address bit ⟨0⟩ is unused. Data in the EEPROM, RCX50 controller, and watch chip is accessible one byte at a time. Data in the boot RAM is accessible with any standard VAX instruction length. Table 6–1 shows the range of addresses assigned to each PCI bus device.

Table 6–1: PCI Device Addresses and Accessibility

Address (hex)	Device	Data Length for Access	Appropriate Move Instructions
2009 0000 to 2009 1FFE	Boot RAM	Longword oriented on longword boundaries (read or write masked)	MOVB, MOVW, MOVL, MOVQ, MOVO, MOVC
2009 8000 to 2009 FFFE	EEPROM	Byte oriented on word boundaries	MOVB, MOVW
200B 00XX	RCX50 controller	Byte oriented on word boundaries	MOVB, MOVW
200B 80XX	Watch chip	Byte oriented on word boundaries	MOVB, MOVW

The byte-oriented devices (EEPROM, watch chip, and RCX50 controller) use only even addresses. For example, the first four bytes for each device are accessible at offsets 0, 2, 4, and 6 from the base address. All read and write transfers occur on word boundaries. Software must use either byte-length instructions, such as MOVB, or word-length instructions, such as MOVW, to read and write the byte-oriented devices. However, for byte-length instructions, the autoincrement and autodecrement addressing modes will not work properly, because they access each byte location twice. Word-length instructions move two bytes of data, but only the low byte is valid. When address bit ⟨1⟩ is 1, valid data is in byte 2 on the DAL bus; when address bit ⟨1⟩ is 0, valid data is in byte 0 on the DAL bus. Microcode realigns the DAL data to ensure that the valid data is in the low byte. Autoincrement and autodecrement addressing modes work properly for word-length instructions, accessing even locations on the byte-oriented devices. However, the upper byte in each transfer is invalid.

Microcode moves data to and from the boot RAM in longwords, but it masks the data according to the instruction. Therefore, data transfer instructions can use any data length at any address, and microcode implementing the instructions moves only the data specified to or from the address specified.

NOTE

Serial-line units, the RCX50 controller, and the watch chip are accessible only to the KA820 module used as the primary processor.

6.1 Serial-Line Units

Each of the four serial-line units is implemented in a UART in the M chip, and each UART makes four privileged processor registers available to software:

Receive CSR

Receive data buffer

Transmit CSR

Transmit data buffer

The serial-line units are full duplex, so they can transmit and receive data simultaneously. When interrupts are enabled, character transfer occurs one byte at a time, and each serial-line unit interrupts the processor at IPL 14 (hex) every time it receives or transmits a character.

To service an interrupt, interrupt handling microcode passes control to a software device driver identified by a vector in the system control block. Software can read and write the UART registers with MFPR and MTPR instructions. Table 6–2 lists the IPR addresses and functions of the four registers for each of the four serial-line units.

Table 6–2: Serial-Line Unit Registers

IPR # (hex)	Register	Function	Serial-Line Unit	SCB Vector
20	RXCS	Console Receive CSR	UART0	F8
21	RXDB	Console Receive Data Buffer	UART0	—
22	TXCS	Console Transmit CSR	UART0	FC
23	TXDB	Console Transmit Data Buffer	UART0	—
50	RXCS1	Serial-Line Unit 1 Receive CSR	UART1	C8
51	RXDB1	Serial-Line Unit 1 Receive Data Buffer	UART1	—
52	TXCS1	Serial-Line Unit 1 Transmit CSR	UART1	CC
53	TSDB1	Serial-Line Unit 1 Transmit Data Buffer	UART1	—
54	RXCS2	Serial-Line Unit 2 Receive CSR	UART2	D0
55	RXDB2	Serial-Line Unit 2 Receive Data Buffer	UART2	—
56	TXCS2	Serial-Line Unit 2 Transmit CSR	UART2	D4
57	TXDB2	Serial-Line Unit 2 Transmit Data Buffer	UART2	—
58	RXCS3	Serial-Line Unit 3 Receive CSR	UART3	D8
59	RXDB3	Serial-Line Unit 3 Receive Data Buffer	UART3	—
5A	TXCS3	Serial-Line Unit 3 Transmit CSR	UART3	DC
5B	TXDB3	Serial-Line Unit 3 Transmit Data Buffer	UART3	—

6.1.1 Receive Control and Status Registers (Read/Write)

RXCS IPR 20 Vector F8

RXCS1 IPR 50 Vector C8

RXCS2 IPR 54 Vector D0

RXCS3 IPR 58 Vector D8

Software can control two serial-line unit functions by writing to the RXCS Registers:

1. Enable or disable receive interrupts from the serial-line unit.
2. Enable or disable the diagnostic loopback function on the serial-line unit.

Software can check three serial-line unit functions by reading these registers:

1. Check whether a character has been received in the receive data buffer of the serial-line unit.
2. Check whether receive interrupts are enabled or disabled for the serial-line unit.
3. Check whether the loopback function is enabled or disabled for the serial-line unit.

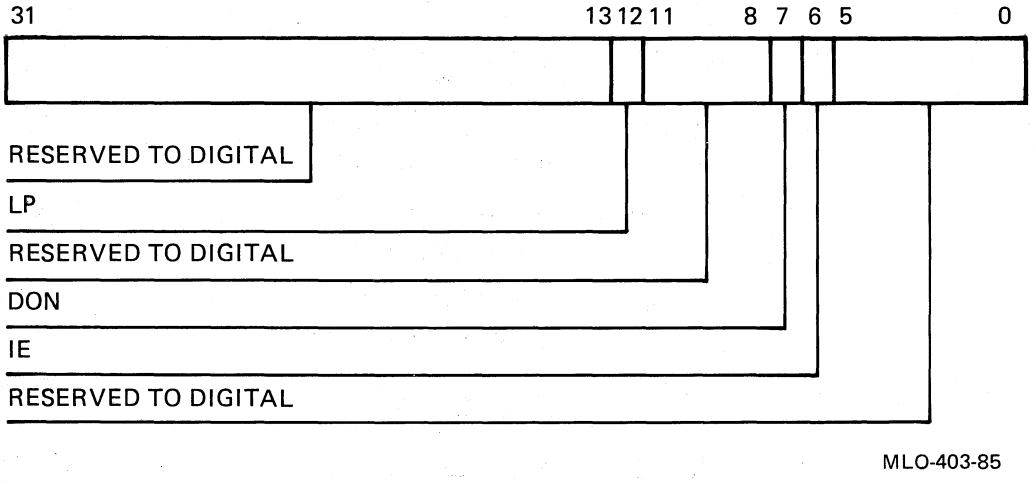


Figure 6–1: Receive CSR Bit Format

6.1.1.1 Bit ⟨12⟩ LP (Loopback Enable, Read/Write) — Diagnostic software can set this bit by writing 1 to it as an aid in testing the serial-line unit. The loopback function changes the input so that it arrives at the receive data buffer from the loopback bus instead of the external serial line.

Software must also set the loopback bit in one of the four transmit CSRs to make use of the loopback function. All four serial-line units can have RXCS loopback bits set at once.

Power-up initialization microcode clears the loopback bit.

6.1.1.2 Bit ⟨7⟩ DON (Done, Read Only) — The serial-line unit sets this bit when it finishes receiving a character in the receive data buffer. If the interrupt enable bit is set, DON interrupts the CPU at IPL 14 (hex).

When software responds to the interrupt by reading the associated Receive Data Buffer Register with an MFPR instruction, MFPR microcode clears the DON bit.

Power-up initialization microcode clears the DON bit.

6.1.1.3 Bit ⟨6⟩ Interrupt Enable (Read/Write) — Software can set (write 1) or clear (write 0) this bit to enable or disable receive interrupts.

Power-up initialization microcode clears the interrupt enable bit. Microcode sets the bit for serial-line unit 0 when entering console mode.

6.1.2 Receive Data Buffer Registers (Read Only)

RXDB IPR 21

RXDB1 IPR 51

RXDB2 IPR 55

RXDB3 IPR 59

Receive Data Buffer Registers provide eight data bits (enough for one character) and two status bits. Power-up initialization microcode clears the registers. After software reads an RXDB register with an MFPR instruction, microcode clears the register automatically.

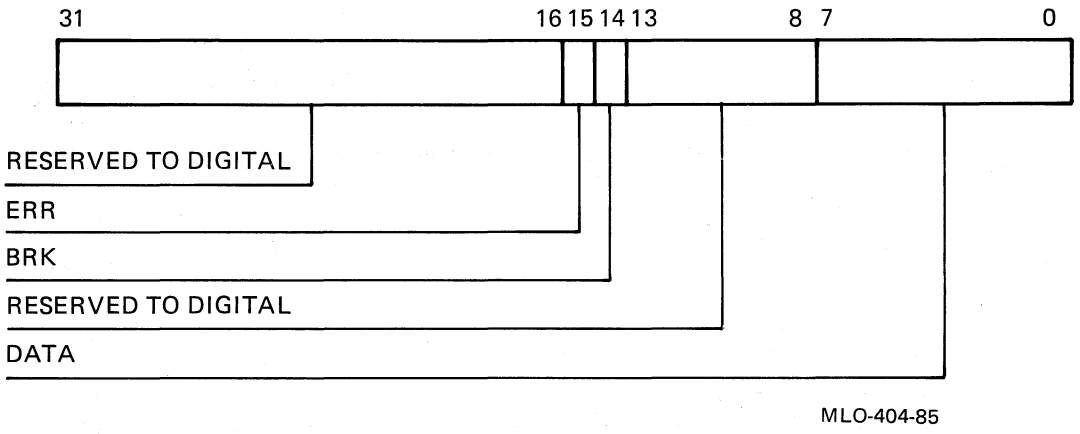


Figure 6–2: Receive Data Buffer Register Format

6.1.2.1 Bit ⟨15⟩ ERR (Error on Received Character, Read Only) — Hardware sets this bit when it detects an overrun or framing error in the received data.

6.1.2.2 Bit ⟨14⟩ BRK (Break, Read Only) — Hardware sets this bit when it detects the 0 state on the serial line for longer than the transmission time for one character at the current baud rate.

6.1.2.3 Bits ⟨7:0⟩ DATA (Received Data, Read Only) — The serial-line unit stores received data in this field.

6.1.3 Transmit Control and Status Registers (Read/Write)

TXCS IPR 22 Vector FC

TXCS1 IPR 52 Vector CC

TXCS2 IPR 56 Vector D4

TXCS3 IPR 5A Vector DC

Software can write to these registers with MTPR instructions to:

1. Enable or disable the loopback function.
2. Start or end transmission of a break.
3. Change the baud rate for the serial-line unit.
4. Enable or disable transmit interrupts from the serial-line unit.

Software can check two serial-line unit functions by reading these registers with MFPR instructions:

1. Check whether the serial-line unit has finished transmitting a character.
2. Check whether transmit interrupts are enabled.

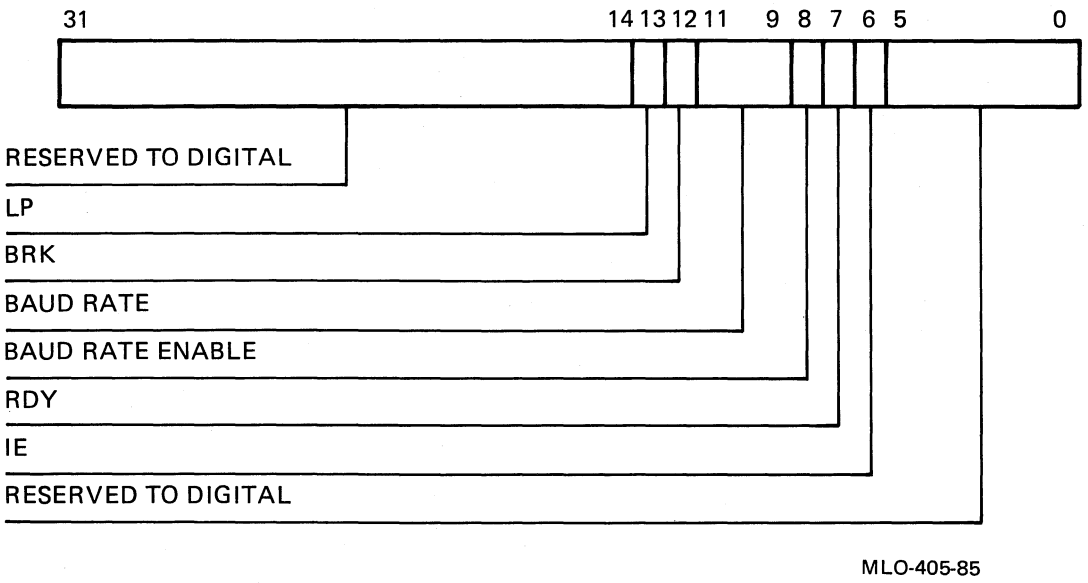


Figure 6–3: Transmit Control Status Register Format

6.1.3.1 Bit ⟨13⟩ LP (Loopback, Write Only) — Diagnostic software sets this bit by writing 1 to it as an aid in testing the serial-line unit. The loopback function changes the serial-line unit output from the external line to the Receive Data Buffer. The loopback bit in one of the four serial-line unit receive CSRs must be set to complete the loop. Software should not set the Transmit Loopback bit for more than one serial-line unit at one time.

Power-up initialization microcode clears the Transmit Loopback bit.

6.1.3.2 Bit ⟨12⟩ BRK (Break, Write Only) — Software can start transmitting a break by writing 1 to this bit. The serial-line unit transmits the break signal for at least the transmission time of one character and continues to transmit break until software writes the bit with 0.

Power-up initialization microcode clears the Break bit.

6.1.3.3 Bits ⟨11:9⟩ Baud Rate (Write Only) — Power-up initialization microcode sets the transmit and receive baud rate for serial-line units 1, 2, and 3 to 1200. It sets the baud rate for serial-line unit 0, the console line, to the baud rate specified in the EEPROM (see Table 4-4 in Chapter 4).

Software or the console operator can set a new baud rate for sending and receiving data by writing a 3-bit value in bits ⟨11:9⟩ and at the same time writing 1 to bit ⟨8⟩, the baud rate enable bit. Table 6–3 shows the value corresponding to each available baud rate.

6.1.3.4 Bit ⟨8⟩ BRE (Baud Rate Enable, Write Only) — Software should write this bit with 1 when it sets the baud rate for a serial-line unit.

Table 6–3: Setting the Baud Rate for a Serial-Line Unit

Binary Value in Transmit CSR Bits ⟨11:9⟩	Baud Rate (Decimal)
000	150
001	300
010	600
011	1200
100	2400
101	4800
110	9600
111	19200

6.1.3.5 Bit ⟨7⟩ RDY (Ready, Read Only) — A serial-line unit sets this bit when it is ready to transmit a character. This happens when the transmit data buffer field is empty and can accept another character for transmission. If the interrupt enable bit is also set, the serial-line unit interrupts the processor at IPL 14 (hex).

When RDY is 0, the serial-line unit is still in the process of transmitting a character.

6.1.3.6 Bit ⟨6⟩ IE (Interrupt Enable, Read/Write) — Software can enable or disable transmit interrupts by writing 1 or 0 to this bit.

Power-up initialization microcode clears the interrupt enable bit to disable transmit interrupts. Microcode sets the bit for serial-line unit 0 when entering console mode.

6.1.4 Transmit Data Buffer Registers (Write Only)

TXDB IPR 23

TXDB1 IPE 53

TXDB2 IPE 57

TXDB3 IPE 5B

Software can transmit data on any of the four serial-line units, one byte at a time, by writing data in the low order byte of the Transmit Data Buffer Register, when the IE and RDY bits in the corresponding TXCS Register are set.

The Transmit Data Buffer Register for serial-line unit 0, the console line, differs from the other Transmit Data Buffer Registers because it includes an extra 4-bit field to indicate whether the information in the low byte is data or a console command.

Figure 6–4 shows the format for the Transmit Data Buffer Registers for serial-line units 1, 2, and 3.

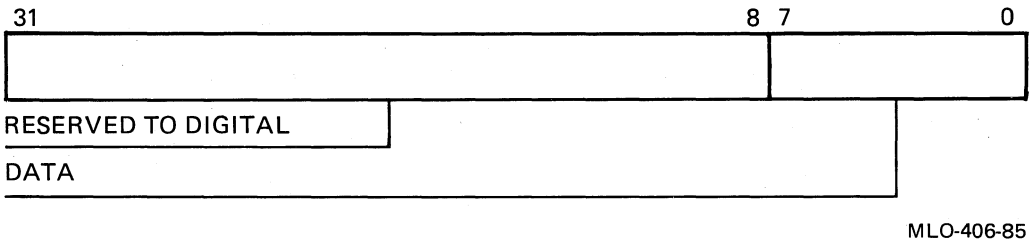


Figure 6–4: Serial-Line Units 1, 2, and 3 Transmit Data Buffer (TXDB1, 2, 3) Format

Figure 6–5 shows the format for the Transmit Data Buffer Register for serial-line unit 0.

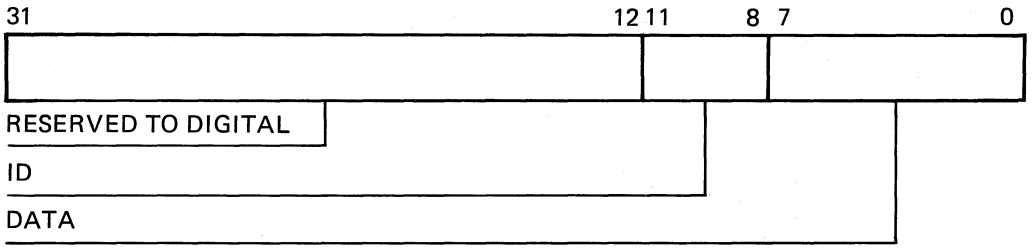


Figure 6–5: Serial-Line Unit 0 Transmit Data Buffer (TXDB) Format

6.1.4.1 Bits ⟨11:8⟩ of TXDB (ID Field, Write Only) — Software must write F (hex) to this field when it uses this register to send a command to the console microcode. When software writes 0 to this field, serial-line unit 0 transmits the data in the low byte on the serial line.

6.1.4.2 Bits ⟨7:0⟩ of TXDB, (Command or Transmit Data, Write Only) — When the accompanying ID field is 0, hardware transmits the data that software writes in bits ⟨7:0⟩ on serial-line unit 0.

When the ID field is F (hex), console microcode interprets the data in bits ⟨7:0⟩ as follows:

2 = Restart the system.

3 = Clear the (warm) restart-in-progress flag in the M chip.

4 = Clear the (cold) bootstrap-in-progress flag in the M chip.

Bootstrap code and restart routines should write 0000 0F03 and then 0000 0F04 (hex) to TXDB to clear the restart-in-progress and bootstrap-in-progress flags in the M chip.

Software can restart the processor by writing 0000 0F02 (hex) to TXDB. In response, microcode sets the BI RESET bit (PCntl CSR bit ⟨28⟩), initiating a power-down/power-up sequence. The action of the KA820 module depends on the setting of the lower key switch on the control panel (external signal PNL RSTRT HLT H on module I/O pin D51).

6.1.4.3 Bits ⟨7:0⟩ of TXDB1, 2, and 3 (Transmit Data, Write Only) — The serial-line transmits the character that software writes in the low byte.

6.2 Using the EEPROM

The EEPROM on the KA820 module is a 16K-byte, electrically-erasable, programmable, read-only memory implemented in two integrated circuits. You can change the data in the EEPROM, yet the data remains valid when the power is off. EEPROM data defines:

1. Customer choices for KA820 module options
2. VAX bootstrap macrocode that processor initialization microcode copies into the boot RAM (this macrocode then loads a primary bootstrap routine into main memory from one of up to ten boot devices)
3. Primary patches that power-up microcode loads into the control-store RAM

Table 6–4: EEPROM Map

Location	Data
First 8K-Byte EEPROM Chip	
2009 8000 to 2009 8002	Constants used in the EEPROM test
2009 8004 to 2009 812E	Reserved to DIGITAL
2009 8130 to 2009 813E	8 bytes reserved for users
2009 8140 to 2009 8156	Miscellaneous constants and reserved locations
2009 8158	VAXBI self-test timeout constant
2009 8160 to 2009 816C	Reserved to DIGITAL
2009 8168 to 2009 816E	VAXBI device type data
2009 8170 to 2009 8174	Unused
2009 8176	RCX50 self-test disable
2009 8178 to 2009 817C	Console data
2009 817E	F chip, BTB, and Cache disable
2009 8180 to 2009 81FE	Reserved to DIGITAL
2009 8200 to 2009 87FE	Boot dispatcher
2009 8A00 to 2009 BFFE	Control-store patches and related data
Second 8K-Byte EEPROM Chip	
2009 C000 to 2009 COAE	Descriptors for ten boot devices
2009 C0B0 to 2009 FFFE	Boot code

Appendix H lists the EEPROM contents at the bit level.

You can use any of three methods to read and write data in the EEPROM, one byte at a time:

1. Privileged software can refer to the EEPROM through node private space addresses using instructions such as MOVB.
2. EEPROM Utility.
3. Console commands D/E and E/E.

Read access to EEPROM locations is unrestricted with methods 1 and 2. However, you cannot write data to the EEPROM unless the signal PNL ENB WT EEPROM H on module I/O pin D50 is true (normally determined by the lower key switch on the control panel).

On a write transfer, the EEPROM latches the address and data and then performs a 10-millisecond update cycle. If you develop software that writes data in the EEPROM, use a software timer to ensure that the intervals between write transfers exceed 10 milliseconds.

The EEPROM options available with the D/E and E/E console commands occupy part of the first EEPROM chip. They include:

RCX50 controller self-test enable and disable

Console logical node ID selection

Console baud rate selection

See Table 4-4 in Chapter 4 for information on console access to these options. See the *VAX 8200 Owner's Manual* for an introduction to the EEPROM Utility.

Writing data to the EEPROM of an attached processor requires special procedures:

1. Load into main memory VAX code that writes the EEPROM.
2. Cause the attached processor to execute the VAX code that you have loaded in main memory.

WARNING

Do not use the Z command or the RXCD Registers to write data to the EEPROM of an attached processor.

6.3 Boot RAM

The boot RAM is an 8K-byte memory dedicated to bootstrap functions. At boot time the CPU loads boot code from the EEPROM into the boot RAM and then executes the boot code directly from the boot RAM.

6.4 Using the Watch Chip

The battery-backed-up watch chip keeps the time of year when power is removed from the computer system. On power-up, software can read the time from the watch chip and set the Time of Day Register (TODR), unless the power outage has exceeded the limits of the battery. Notice that the TODR stores time in a 32-bit format and increments the time every 10 milliseconds. The watch chip, in contrast, stores time in units of seconds, minutes, hours, days, months, and years and updates the time every second.

The watch chip stores time, status, and control information in 8-bit registers. The PCI bus rotates watch chip register addresses and data in a way that is not transparent to software, as shown in Figure 6–7.

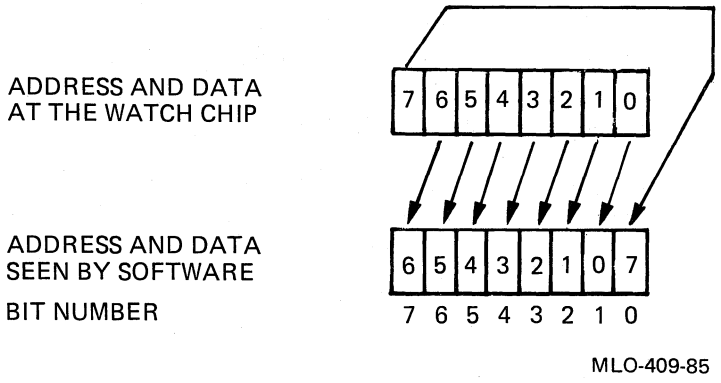


Figure 6–6: Watch Chip Bit Rotation on the PCI Bus

You can access the watch chip registers on word boundaries with MOVB or MOVW instructions, and the data will always be offset one bit to the left. Hardware implicitly multiplies read data by 2 and divides write data by 2.

Table 6–5: Watch Chip Registers

Address	Function
200B 8000	Seconds
200B 8002	Reserved to DIGITAL
200B 8004	Minutes
200B 8006	Reserved to DIGITAL
200B 8008	Hours
200B 800A	Reserved to DIGITAL
200B 800C	Reserved to DIGITAL
200B 800E	Day of the month
200B 0810	Month
200B 8012	Year
200B 8014	CSR A — BUSY bit
200B 8016	CSR B — OFF bit
200B 8018	CSR C — Reserved to DIGITAL
200B 801A	CSR D — VALID bit
200B 801C to 200B 807E	50 bytes RAM reserved to DIGITAL

Software should therefore divide read data by 2 and multiply data to be written by 2, as shown in Tables 6–5 and 6–6. Table 6–5 lists the watch chip registers.

Table 6–6 shows the ranges of values possible for each of the date and time registers as stored in the watch chip and as read or written by software.

Table 6–6: Watch Chip Data Interpretation

Address	Units	Decimal Range	Hex Range at Chip	Hex Range Seen by Software
200B 8000	Seconds	0 — 59	00 — 3B	00 — 76
200B 8004	Minutes	0 — 59	00 — 3B	00 — 76
200B 8008	Hours	0 — 23	00 — 17	00 — 2E
200B 800E	Day of the month	1 — 31	01 — 1F	02 — 2E
200B 0810	Month	1 — 12	01 — 0C	02 — 18
200B 8012	Year	0 — 99	00 — 63	00 — C6

Table 6–7 shows how your software should interpret a specific date and time read from the watch chip.

Table 6–7: Watch Chip Date and Time Sample

	Watch Chip Output		Data Read by Software	
Time	Binary	Hex	Binary	Hex
21 seconds	0001 0101	15	0010 1010	2A
58 minutes	0011 1010	3A	0111 0100	74
5 hours	0000 0101	05	0000 1010	0A
15th day	0000 1111	0F	0001 1110	1E
February	0000 0010	02	0000 0100	04
82nd year	0101 0010	52	1010 0100	A4

The control and status registers on the watch chip let software:

1. Check the validity of the date and time registers
2. Set the time
3. Stop and start the chip.

In the control and status register descriptions that follow, the bit formats show the register data already rotated, as the software reads and writes it. For example, a bit shown as bit 0 is really bit 7 on the watch chip.

6.4.1 Watch Chip CSR A, Address 200B 8014

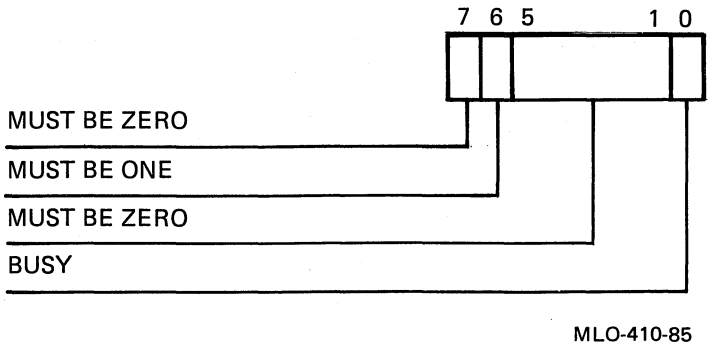


Figure 6–7: Watch Chip CSR A Format

- Bit ⟨0⟩, BUSY (Read only) —

  BUSY = 1: The watch chip is busy with an update cycle; date and time registers are undefined.

  BUSY = 0: The watch chip is not busy; date and time registers are valid.

  Software should check BUSY before reading the date and time registers.

  The watch chip sets BUSY for 2 milliseconds every second. If software finds BUSY set, it should try again in 2 milliseconds.

  If BUSY is cleared, software has at least 244 microseconds to read the date and time registers before the next update cycle. Reading the registers should take less than 40 microseconds.

- Bits ⟨7:1⟩, Miscellaneous setup bits (Read/write) — Software must write these bits as shown in Figure 6–8 before it sets the time in the watch chip.

6.4.2 Watch Chip CSR B, Address 200B 8016

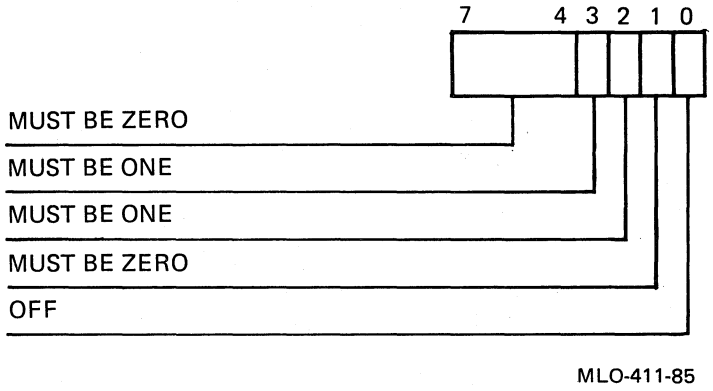


Figure 6–8: Watch Chip CSR B Format

- Bit ⟨0⟩, OFF (Read/write) — Software must stop the watch chip by setting OFF before it loads the date and time registers.

  Software can start the watch chip after setting the time by clearing OFF.

- Bits ⟨7:1⟩, Miscellaneous setup bits (Read/write) — Software must write these bits as shown in Figure 6–9, when it sets or clears the OFF bit.

6.4.3 Watch Chip CSR C, Address 200B 8018

This register is reserved to DIGITAL.

6.4.4 Watch Chip CSR D, Address 200B 801A

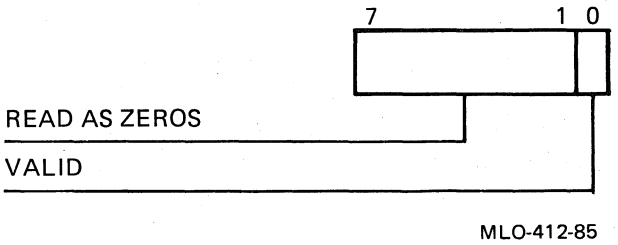


Figure 6–9: Watch Chip CSR D Format

- Bit ⟨0⟩, VALID (Read only) — The VALID bit tells whether the time represented in the watch chip registers is correct. If the battery backup voltage falls below the required level, a sensing circuit clears the VALID bit.

  VALID = 1: Watch chip registers are valid.

  VALID = 0: Watch chip registers are invalid.

  The watch chip sets VALID to 1 after software reads CSR D. Therefore, when software reads VALID as 0, it should immediately update the date and time registers in the watch chip. The state of the VALID bit will otherwise be misleading.

6.4.5 Bootstrap Software Date and Time Responsibilities

Bootstrap software should take the following steps concerning date and time.

1. Determine whether the value in the TODR is correct. If the value is correct, skip the remaining five steps. If it is incorrect, go to step 2. Note that if you set the initial value to a high number, any lower value must be wrong.
2. Read the VALID bit in the watch chip CSR D to determine whether the contents of the watch chip are correct.
3. If VALID is 1, read the BUSY bit in CSR A. If VALID is 0, go to step 6.
4. If BUSY is 0, read the watch chip date and time registers, convert the time to the 32-bit format, and set the TODR.
5. If BUSY is 1, wait until it is 0 and then go to step 4.
6. If VALID is 0, prompt the operator for the date and time, convert the date and time to the 32-bit format, and load this value in the TODR. Then convert the date and time to the watch chip register formats and load the watch chip as follows:
   - Write 0D (hex) to CSR B to stop the watch chip.
   - Initialize CSR A by writing 40 (hex). (See Figure 6–8.)
   - Write the date and time in the appropriate registers.
   - Start the watch chip by writing 0C (hex) to CSR B. (See Figure 6–9.)

6.4.6 Compatibility with VMS and ULTRIX

If you write your own system software, you may want your use of the watch chip to be compatible with that of VMS and ULTRIX-32. These operating systems always set the year in the watch chip to 1982 (software saves the current year in a location on disk). If the current year is a leap year, the Day and Month registers in the watch chip will be ahead by one day, beginning in March.

VMS and ULTRIX reset the date and time in the watch chip the first time the system is bootstrapped, following 1 January, resetting the Year Register from 1983 to 1982.

NOTE

VMS uses local standard time; ULTRIX-32 uses Greenwich mean time.

6.5 Controlling the RCX50 Controller

The RCX50 controller enables software to read and write diskette data one byte at a time. The controller provides random access to the 800 512-byte

blocks of data on one side of each diskette. The data side of a diskette contains 80 tracks, and each track contains 10 sectors. Each sector stores 512 bytes of data. Software accesses the data within each sector sequentially, however, beginning with the first byte.

Software communicates with the RCX50 controller by writing and reading ten 8-bit registers. Some of the registers have several functions. For example, when software writes to Register RX5CS0, it defines the command function. When software reads Register RX5CS0, it checks the completion and error status of the last command executed.

After the RCX50 controller performs the requested function, it interrupts the processor if interrupts are enabled. Software should ensure that bit ⟨7⟩ (RXIE) in the port controller CSR is set.

Consider Register RX5CS4 as another example. Following a data transfer command in which a seek error occurs, Register RX5CS4 contains the number of the incorrect track. Following a maintenance command such as *restore drive*, Register RX5CS4 contains system configuration data indicating what disks are available and which ones are double sided. Unlike data in the watch chip registers, RCX50 controller register data is not rotated during data transfers. Table 6–8 lists the registers and their uses.

Table 6–8: RCX50 Controller Register Functions

Address (hex)	Register	Command or Data Transfer Function	Status Following a Data Transfer Command	Status Following a Maintenance Command
200B 0004	RX5CS0	Write to load device and function select codes	Contains command completion and error information	Contains command completion and error information
200B 0006	RX5CS1	Write to load track number	Contains error message	Contains error message
200B 0008	RX5CS2	Write to load sector number	Contains current track number	Contains current track number
200B 000A	RX5CS3	—	Contains current sector number	Contains current status on unit and volume
200B 000C	RX5CS4	—	Contains incorrect track number	Contains system configuration data
200B 000E	RX5CS5	Write to load extended function code	—	—

(Continued on next page)

Table 6–8: RCX50 Controller Register Functions (Cont.)

Address (hex)	Register	Command or Data Transfer Function	Status Following a Data Transfer Command	Status Following a Maintenance Command
200B 0010	RX5EB	Read data buffer	—	—
200B 0012	RX5CA	Read to clear data buffer address	—	—
200B 0014	RX6GO	Read to start	—	—
200B 0016	RX5FB	Write data buffer	—	—

6.5.1 Data Transfer Examples

Examples of command sequences that write and read data on a diskette may help to clarify standard uses of these registers.

Write data example:

1. Read Register RX5CA to set the data buffer address to 0.
2. Load Register RX5FB, one byte at a time, with the 512 bytes of data to be stored.
3. Load Register RX5CS0 to select

   drive 0, disk 1, side 0

   the write sector function code (111 binary)

   the normal motor timeout option (0) (see Section 6.5.2.1)
4. Load Register RX5CS1 with 23 (hex), the track to be written.
5. Load Register RX5CS2 with 5 (hex), the sector to be written.
6. Read Register RX5GO to start executing the write sector command.

   The controller writes the data stored in the data buffer to the disk, moving the 512 bytes previously written to Register RX5FB to the track and sector selected.
7. When the RCX50 controller interrupts the CPU (vector F0), read Register RX5CS0 to check command completion status. The DONE bit should be set, and the ERROR bit should be clear.

Read data example:

1. Read Register RX5CA to set the data buffer address to 0.
2. Load Register RX5CS0 to select

   drive 0, disk 0, side 0

   the read sector function code (100 binary)

   the normal motor timeout option (0)

3. Load Register RX5CS1 with 1A (hex), the track to be read.
4. Load Register RX5CS2 with 7 (hex), the sector to be read.
5. Read Register RX5GO to start executing the read sector command. The controller copies 512 bytes from the track and sector selected into the data buffer.
6. When the RCX50 controller interrupts the CPU, read Register RX5CS0 to check the command completion status.
7. If the DONE bit is set and the ERROR bit is clear in Register RX5CS0, read Register RX5EB 512 times to move the requested data from the data buffer to main memory.
8. Note that the KA825 is capable of servicing an interrupt and clearing the interrupt pending bit in the PCtrl before the RCX50 controller deasserts INTRQA. This results in the interrupt pending bit being reset in the PCtrl. When writing RCX50 drivers, handle this occurrence either by clearing the interrupt pending bit in the PCtrl by macro code, or by ignoring RX interrupts when the RX5CS0 DONE bit is not set.

6.5.2 Register RX5CS0, Address 200B 0004

RX5CS0 is central to control of the RCX50 controller. It has two formats: command function format and status format (data transfer status and maintenance status).

6.5.2.1 RX5CS0 Command Function — When software writes to this register, the RCX50 interprets the information as a command, according to the format shown.

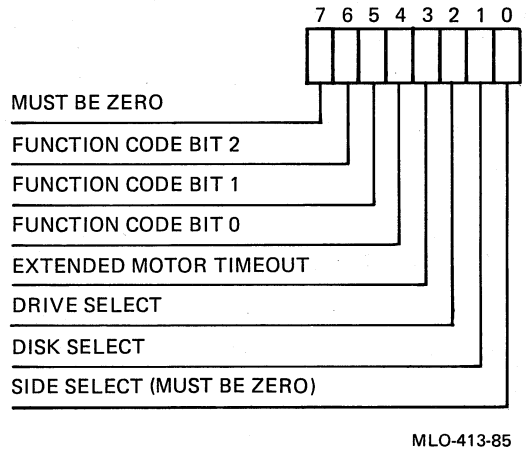


Figure 6–10: Register RX5CS0 Command Function Format

- Bit ⟨0⟩, Reserved to DIGITAL, must be zero.
- Bits ⟨2:1⟩, Drive, and Disk Select — This field selects the surface to read or write, as shown in Table 6–9.

Table 6–9: Diskette Surface Selection Code Interpretation

Unit Number	Diskette Surface	Bit 2 Drive Select	Bit 1 Disk Select	Bit 0 Side Select
unit 0	drive 0 disk 0 side 0	0	0	0
unit 1	drive 0 disk 1 side 0	0	1	0
unit 2	drive 1 disk 0 side 0	1	0	0
unit 3	drive 1 disk 1 side 0	1	1	0

NOTES

Standard configurations incorporating the KA820 module provide one disk drive that uses single-sided diskettes. Side 0 is the data side.

The unit numbers shown in Table 6–9 refer to unit number used in the RX5CS3 Current Status Register. These unit numbers do not refer to VMS unit numbers.

- Bit ⟨3⟩, Extended Motor Timeout — Following completion of a disk access command, the spindle normally stops rotating in 3 seconds. You can extend the rotation for 30 seconds by writing 1 to bit ⟨3⟩. This function is useful when you perform a sequence of data transfers to or from the diskette.
- Bits ⟨6:4⟩, Function Code

  The RCX50 controller executes eight major functions (Table 6–10).
- Read Status Function (000) — The RCX50 controller responds to this maintenance function by showing in Registers RX5CS0 through RX5CS4 the status of the track selected in RX5CS2 on the diskette surface selected by RX5CS0 bits ⟨2:0⟩. RX5CS3 contains the *volume changed* status, generally the most useful status information. The read status function resets all *volume changed* information.
- Maintenance Mode Function (001) — System start-up software and diagnostic software can use this maintenance function to test the RCX50 controller. Registers RX5CS0 through RX5CS4 show the status of the controller following completion of the test.

Table 6–10: RCX50 Function Codes

Binary Function Code in RX5CS0 Bits ⟨6:4⟩	Function	Corresponding Register Status
000	Read status	Maintenance status
001	Maintenance mode	Maintenance status
010	Restore drive	Maintenance status
011	RCX50 initialize	Maintenance status
100	Read sector	Data transfer status
101	Extended function	Data transfer status
110	Read address	Data transfer status
111	Write sector	Data transfer status

- RCX50 Restore Drive Function (010) — The RCX50 seeks track 0 on the specified diskette surface in response to this maintenance function. Registers RX5CS0 through RX5CS4 show the maintenance status.
- RCX50 Initialize Function (011) — The RCX50 restores the drive, checks the drive status, and tests the controller in response to this maintenance function. Registers RX5CS0 through RX5CS4 show the maintenance status.
- Read Sector Function (100) — The RCX50 reads the sector specified in RX5CS2 in the track specified in RX5CS1 in response to this data transfer function. It stores the 512 bytes read in the data buffer and shows data transfer status in Registers RX5CS0 through RX5CS4.
- Extend Function (101) — The RCX50 looks in Register RX5CS5 for the code defining the type of data transfer function required. Six extended functions are available.

Table 6–11: RCX50 Extended Functions

Extended Function Code (hex)	Function
0	Read with retries, up to 10 times.
1	Write a sector with a deleted data mark in the sector header.
2	Used for nonnative diskettes: report format parameters of the unit specified by Register RX5CS0 bits ⟨2:0⟩ in Registers RX5CS1, RX5CS2, RX5CS3, and RX5CS4.
3	Used for nonnative diskettes: set format parameters specified in Registers RX5CS1, RX5CS2, RX5CS3, and RX5CS4.

(Continued on next page)

Table 6–11: RCX50 Extended Functions (Cont.)

Extended Function Code (hex)	Function
4	Report the RCX50 controller version number in RX5CS2.
5	Read a sector and compare it with the contents of the data buffer. You should load the data buffer with the appropriate values before starting this function.

- Read Address Function (110) — The RCX50 reads the first header found at the current track location on the diskette surface selected by RX5CS0 bits ⟨2:0⟩ in response to this data transfer function. The controller sequentially transfers six bytes to Register RX5EB:

  Track address

  Side address

  Sector address

  Sector length

  Header CRC 1

  Header CRC 2

- Write Sector Function (111) — The RCX50 writes the contents of the 512-byte data buffer to the track and sector selected by Registers RX5CS1 and RX5CS2 on the diskette surface selected by RX5CS0 bits ⟨2:0⟩ in response to this data transfer function.

6.5.2.2 RX5CS0 Data Transfer Status and Maintenance Status — Following completion of a data transfer function or maintenance function, the RX5CS0 Register has the following format.

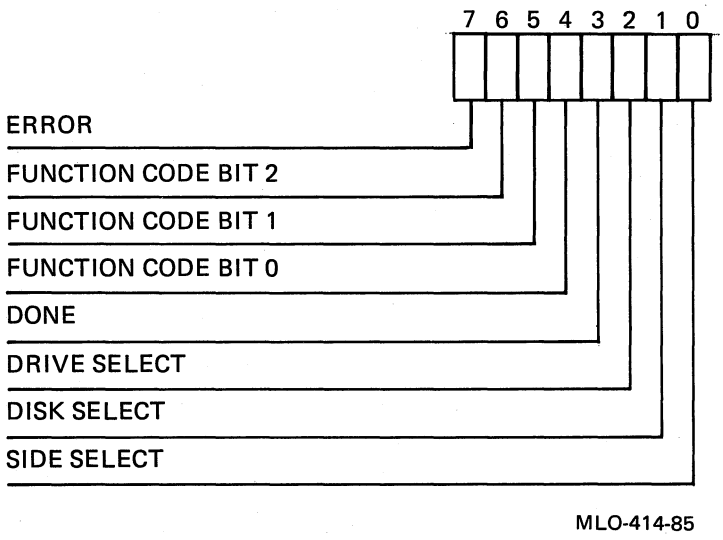


Figure 6–11: Register RX5CS0 Status Format

- Bits ⟨6:4,2:0⟩ — The functions of bits ⟨6:4,2:0⟩ correspond to the functions of these bits in the command function format listed in Section 6.5.2.1.
- Bit ⟨3⟩, DONE (Read only)

  0 = busy

  1 = done

  When the DONE bit is set, the RCX50 controller is not busy; all registers are therefore accessible to the CPU.

  If the RCX50 controller is busy executing a command, software should not try to read this register or any other RCX50 controller register. When it is busy, the controller responds to any read request to any register with a null byte in which bit ⟨3⟩ is 0.

- Bit ⟨7⟩, ERROR (Read only)

  0 = no error

  1 = error

  When the ERROR bit is set, it means that an error occurred during execution of the last command. Software can read the error code in Register RX5CS1 as an aid in defining the cause of the error.

6.5.3 Register RX5CS1, Address 200B 0006

RX5CS1 uses two formats: a command function format (Track Register) and a status format (Error Register).

6.5.3.1 RX5CS1 Command Function, Track Register — Software should load RX5CS1 with the target track number for disk access. Valid track numbers range from 00 to 4F (hex).

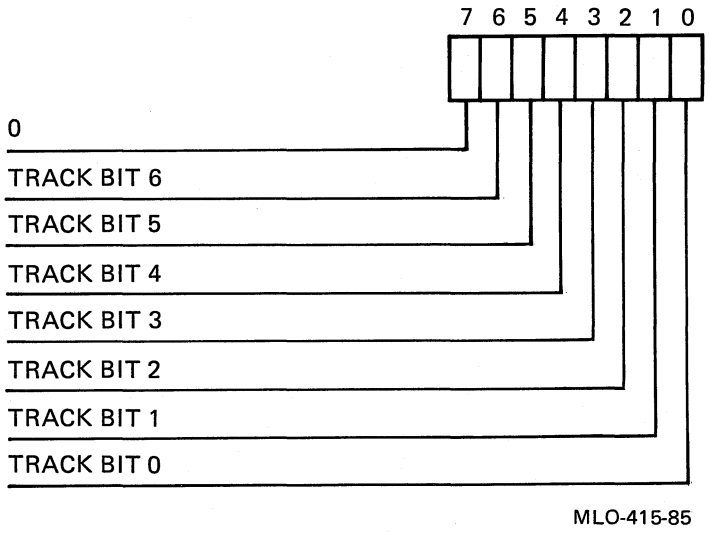


Figure 6–12: RX5CS1 Command Function Format

6.5.3.2 RX5CS1 Data Transfer and Maintenance Status Format, Error Register —

When the ERROR bit in Register RX5CS0 is set, Register RX5CS1 contains an error code defining the error that occurred when the RCX50 executed the last function.

Table 6–12: RCX50 Error Codes Available in Register RX5CS1

Error Code (hex)	Error
00	No error
08	Drive 0 track 00 sensor failure
10	Drive 1 track 00 sensor failure
18	Both drives failed to respond; system has no drives
20	Tried to access a track greater than 4F (hex)
28	Drive fails to seek home
30	Data record not found; data mark (DAM) not found within 43 (decimal) bytes following ID
38	ID record not found
40	Command timeout
48	Selected side is not a match
50	Selected unit is not ready
58	Disk is not installed correctly
60	ID CRC
68	Seek error
70	Data required (DRQ) does not appear within 32 microseconds
78	Soft ID read error
80	Data CRC
88	Lost data; 8051 chip did not respond to data required (DRQ) within 23 microseconds
90	Tried to access an unavailable unit
98	Drive not ready during write
A0	Drive not ready during read
A8	No match for the specified sector
B0	Unit write protected on a write command
B8	Tried to access a sector numbered 0 or greater than A (hex)

(Continued on next page)

Table 6–12: RCX50 Error Codes Available in Register RX5CS1 (Cont.)

Error Code (hex)	Error
C0	Maintenance status only — the low-order 4 bits of the RAM failed to pass the memory test
C8	Maintenance status only — the high-order 4 bits of the RAM failed to pass the memory test
D0	Maintenance status only — no index pulse was detected
D8	Maintenance status only — drive speed is not within limits
E0	Maintenance status only — bad format or a blank disk
E8	Maintenance status only — stepping error
EC	Tried to set unsupported diskette parameters
F0	Maintenance status only — phase-locked loop (PLL) frequency is not within the limits
F4	Tried to read a sector with a deleted data mark
F8	Maintenance status only — data buffer is bad
FC	Tried to write a non-RCX50 diskette

6.5.4 Register RX5CS2, Address 200B 0008

Register RX5CS2 performs two functions and provides two corresponding formats. RX5CS2 is the Sector Register on a data transfer, and it shows the current track following execution of a function.

6.5.4.1 RX5CS2 Data Transfer Format, Sector Register — Before transferring data to or from a diskette, software should load RX5CS2 with the target sector number. The valid range is 01 to 0A (hex).

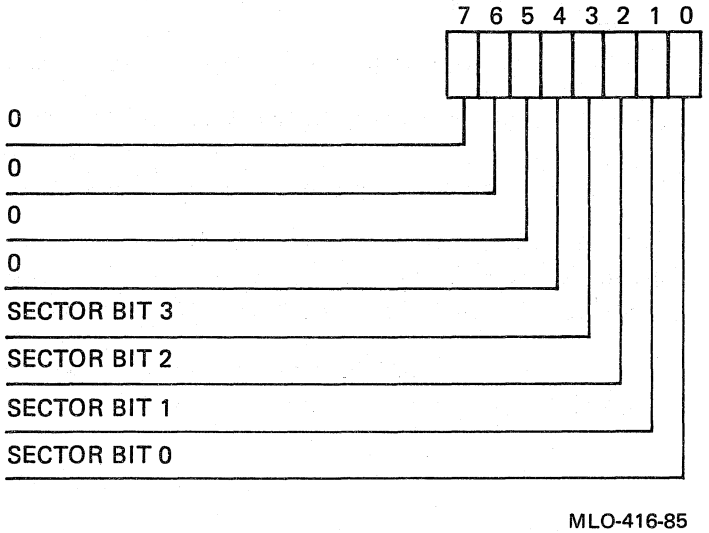


Figure 6–13: RX5CS2 Command Function Format: Sector Register

6.5.4.2 RX5CS2 Data Transfer and Maintenance Status Format, Current Track Register — Following a data transfer function or maintenance function RX5CS2 shows the track number used in the command.

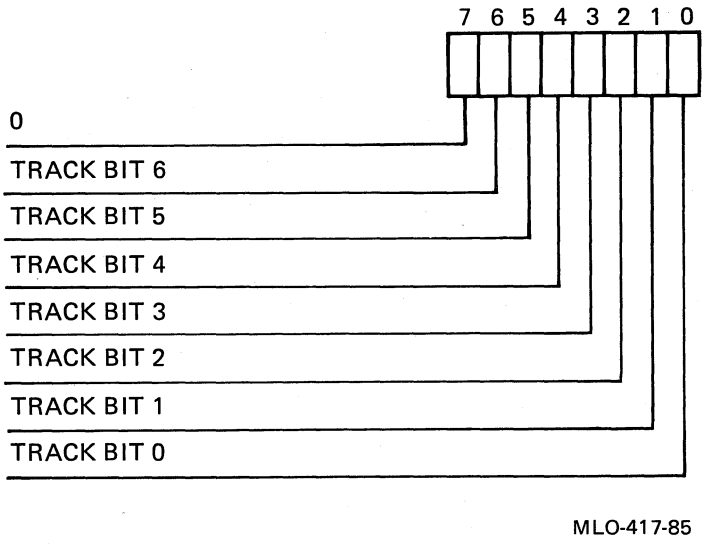


Figure 6–14: RX5CS2 Status Format: Current Track Register

6.5.5 Register RX5CS3, Address 200B 000A

Register RX5CS3 provides two status formats. It identifies the current sector following a data transfer function, and it gives the current RCX50 controller status following a maintenance command.

6.5.5.1 RX5CS3 Data Transfer Status Format, Current Sector Register — Following a data transfer function RX5CS3 shows the current sector number.

6.5.5.2 RX5CS3 Maintenance Status Format, Current Status Register — The first four bits of RX5CS3 show the status of the unit specified by Register RX5CS0 (see Table 6–9 and Figure 6–11).

During controller initialization, bits ⟨3:0⟩ show the status of drive 0, disk 0.

Bits ⟨7:4⟩ show the volume status, updated since the last read status command. These bits are affected only by the read status command. The volume numbers (3-0) correspond to the unit numbers shown in Table 6–9.

You should not normally place diskettes in the diskette drive during initialization. However, the controller tests for diskette presence and sets the corresponding *Volume Changed* bit if it detects a diskette.

Interrupts related to the *Volume Changed* bits are enabled following the first read status command after initialization.

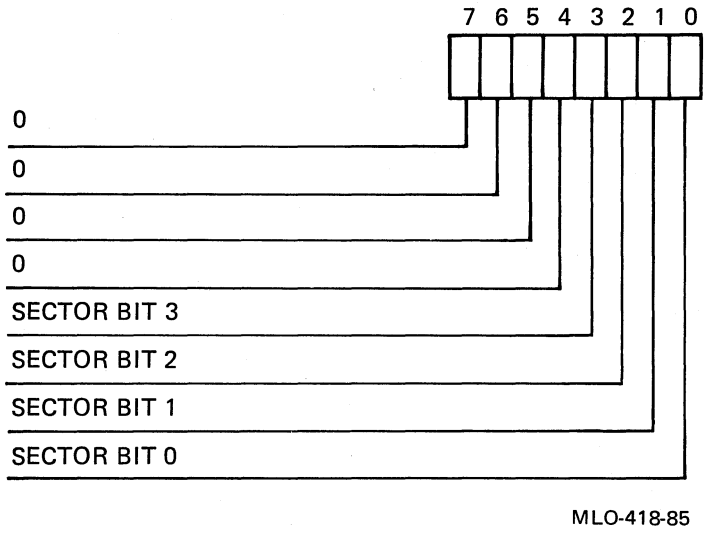


Figure 6–15: RX5CS3 Data Transfer Status Format: Current Sector Register

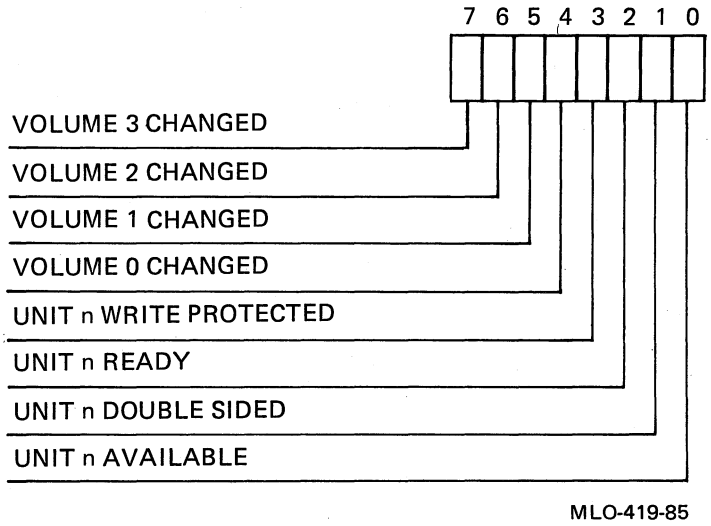


Figure 6–16: RX5CS3 Maintenance Status Format: Current Status Register

- Bit ⟨0⟩, Unit n Available (where n is specified in RX5CS0)

  0 = not available

  1 = available

- Bit ⟨1⟩, Unit n Double Sided (where n is specified in RX5CS0)

  0 = single sided

  1 = double sided

- Bit ⟨2⟩, Unit n Ready (where n is specified in RX5CS0)

  0 = not ready

  1 = ready

- Bit ⟨3⟩, Unit n Write Protected (where n is specified in RX5CS0)

  0 = not write protected

  1 = write protected

- For all Volume Changed bits ⟨7, 6, 5, 4⟩

  0 = The ready signal has not changed since the last read status command

  1 = The ready signal has changed since the last read status command

6.5.6 Register RX5CS4, Address 200B 000C

Register RX5CS4 provides two formats. The data transfer status gives the incorrect track number, following a seek error. The maintenance status gives the system configuration.

6.5.6.1 RX5CS4 Data Transfer Status, Incorrect Track Register — If a seek error occurs when the RCX50 is executing a command, RX5CS4 yields the track number to which the read/write head actually moved. If no seek error occurs, the register contains all zeros.

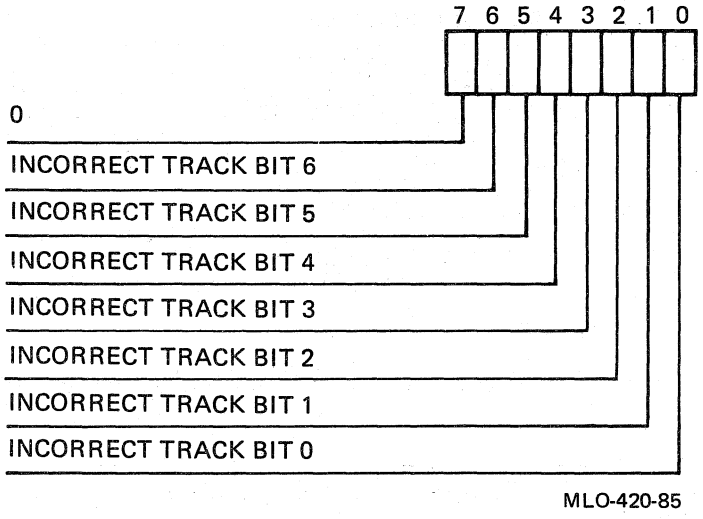


Figure 6–17: RX5CS4 Command Function Format: Incorrect Track Register

6.5.6.2 RX5CS4 Maintenance Status, System Configuration Register — RX5CS4 summarizes the diskette drive configuration. Four bit pairs make up the register format. Each bit pair shows the status of one diskette (volume). The initialization function updates RX5CS4 with the current system configuration.

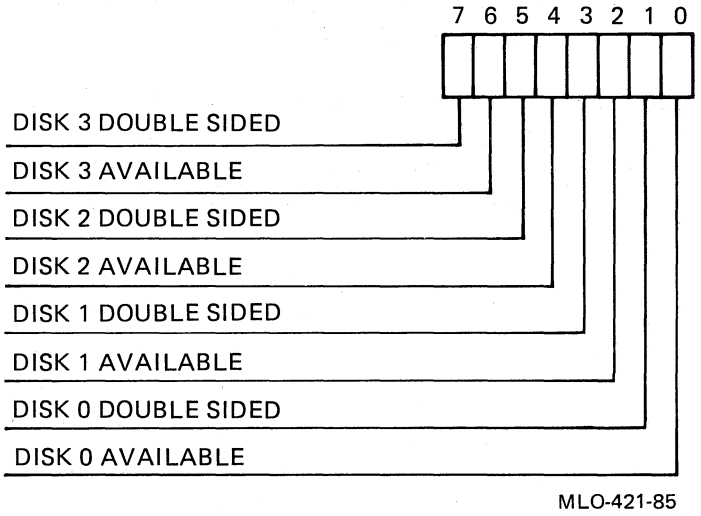


Figure 6–18: RX5CS4 Maintenance Status Format: System Configuration Register

- For all Available bits ⟨6, 4, 2, 0⟩: 0 = the diskette is not present
  1 = the diskette is present
- For all Double Sided bits ⟨7, 5, 3, 1⟩: 0 = single sided
  1 = double sided

NOTE

RX50 supported diskettes are single sided.

6.5.7 Register RX5CS5, Address 200B 000E

Software should load Register RX5CS5 with an extended function code when it loads 101 (binary) in bits ⟨6:4⟩ of Register RX5CS0. Table 6–11 lists the codes and their meanings.

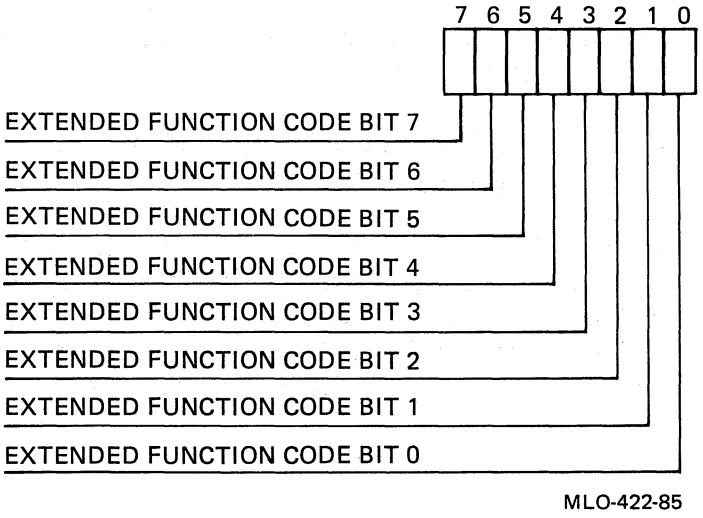


Figure 6–19: RX5CS5 Format: Extended Function

6.5.8 Register RX5EB, Empty Sector Buffer Register, Address 200B 0010

Register RX5EB gives software read access to the 512-byte data buffer on the RCX50 controller, one byte at a time. To read data from a diskette, software should first execute the read sector command and set the address of the data buffer to 0 (see the read data example in Section 6.5.1).

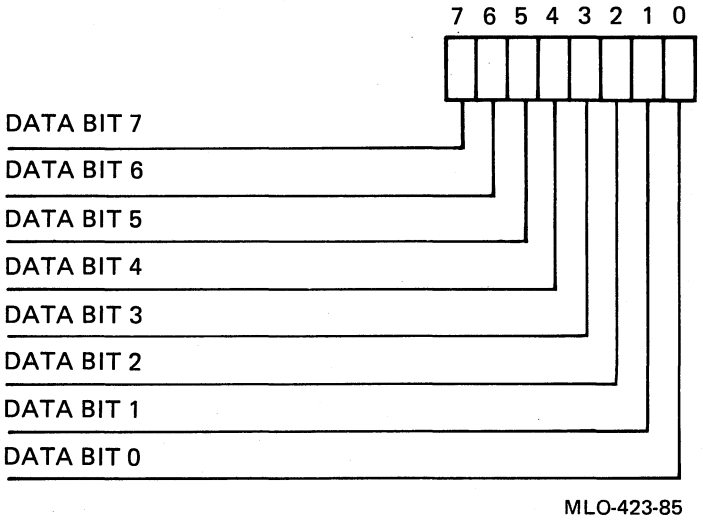


Figure 6–20: RX5EB Format: Empty Sector Buffer Register

Each time software reads Register RX5EB, the controller increments the address of the data buffer location being examined. Software must keep track of the location associated with each byte it reads, since the controller does not supply address information.

6.5.9 Register RX5CA, Clear Address Register, Address 200B 0012

Software sets the address in the data buffer to 0 when it accesses (reads or writes) Register RX5CA.

6.5.10 Register RX5GO, Start Command Register, Address 200B 0014

When software reads or writes Register RX5GO, the controller executes the function specified in Registers RX5CS0 through RX5CS5. This should be the last step software performs in carrying out any RCX50 command. The controller interrupts the CPU when it finishes executing the command. Software should then read the RX5CS0 Register to check the command status.

6.5.11 Register RX5FB, Fill Sector Buffer Register, Address 200B 0016

Register RX5FB gives software write access to the 512-byte data buffer on the RCX50 controller, one byte at a time.

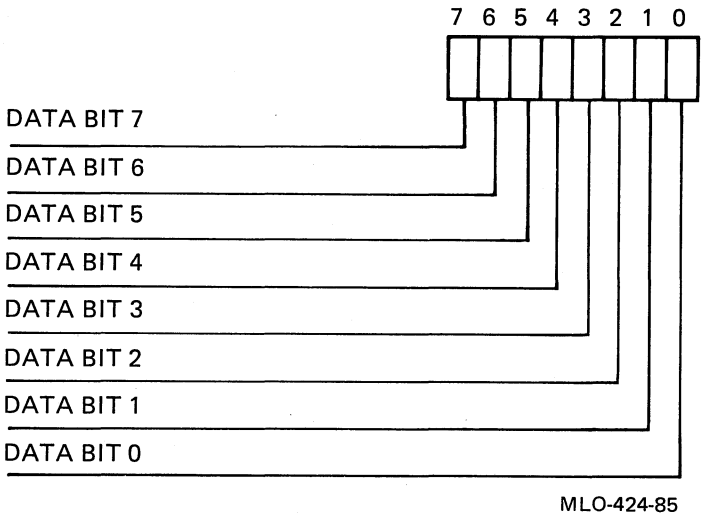


Figure 6–21: RX5FB Format: Fill Sector Buffer Register

Each time software writes to the register, the controller increments the address of the data buffer location being written (see the write data example in Section 6.5.1). Software must keep track of the location accessed with each write transfer, since the controller does not supply address information. Before writing data, software should read Register RX5CA to set the data buffer address to 0.

Chapter 7
KA820 Diagnostics

The optional VAX 8200 system diagnostic package includes programs that test all KA820 functions. Self-test is available to and appropriate for all users. You can order the full VAX diagnostics package through your local DIGITAL Field Service office. These programs provide more complete fault isolation than self-test, but they require detailed knowledge of the VAX 8200 hardware. The standard VAX diagnostic programs explained in this chapter are available if you buy a diagnostic license; they are appropriate only for DIGITAL Field Service engineers and OEM customers who maintain their own systems.

Table 7–1: Diagnostic Program Categories Related to the VAX 8200

Program Category	Intended User	Run-time Environment	Area Tested	Reference
Self-test	All customers and DIGITAL Field Service	Console	KA820 hardware	Chapter 3 of this manual
Separate Cluster Exerciser Programs	DIGITAL Field Service and Licensed customers	VAX Diagnostic Supervisor, levels 4, 3, and 2	VAX-generic and KA820-specific functions	This Chapter
Serial-Line Unit Test	DIGITAL Field Service and licensed customers	VAX Diagnostic Supervisor Level 2R	KA820 Serial-Line Units	This Chapter

The macrodiagnostic programs described here require a variety of run-time environments (diagnostic program levels), because they test a wide range of functions, using a building-block approach.

Level 4 — Programs that run without the VMS operating system (stand-alone) and without the VAX Diagnostic Supervisor, VDS

Level 3 — VDS-based programs that run without VMS (stand-alone)

Level 2 — VDS-based programs that can run either with VMS (on-line) or without VMS (stand-alone)

Level 2R — VDS-based programs that run only with VMS (on-line)

Refer to the *VAX Diagnostic System User's Guide* for a general description of the VAX diagnostic system and an introduction to VDS. The *VAX Diagnostic System User's Guide* offers instructions for building the diagnostic directory (SYSMAINT) and installing and updating the VAX diagnostic software.

Table 7–2 offers an overview of the programs described in this chapter.

Table 7–2: Diagnostic Programs Described in this Chapter

Program Code	Program Name	Run-time Environment	Hardware Tested
EVKAA	VAX-generic cluster exerciser: hardcore instruction test	Level 4 (stand-alone, boot and run from the console)	VAX instruction set used by VDS
EVKAB	VAX-generic cluster exerciser: basic instruction exerciser	Level 2 (on-line or stand-alone)	Basic VAX instruction set, non-privileged
EVKAC	VAX-generic cluster exerciser: floating-point instruction exerciser	Level 2 (on-line or stand-alone)	Floating-point VAX instruction set, nonprivileged
EVKAE	VAX-generic cluster exerciser: privileged architecture exerciser	Level 3 (stand-alone)	Privileged VAX instruction set
EBKAX	KA820-specific cluster exerciser	Level 3 (stand-alone)	KA820 CPU, RX50 drive and RCX50 controller, serial-line units, watch chip, boot RAM, multiprocessor functions
EBDAN	Serial-line unit diagnostic program	Level 2R (on-line only)	Serial-line units 1, 2, and 3

7.1 Load Paths

You can use two kinds of load paths for loading diagnostic programs into main memory:

1. Load from the maintenance system disk, the operating system disk, or other mass storage medium.
2. Load from RX50 diskettes distributed by DIGITAL.

If a hardware failure prevents you from loading diagnostics from the maintenance system disk or another disk attached to a VAXBI node, you can try loading diagnostics another way. If two or three of the load paths are faulty, the fault is probably on the KA820 module and should be reflected in the results of the self-test program. Use the console T command to run self-test. The red and yellow light-emitting diodes (LEDs) on the KA820 module should also help you to identify the problem (see Chapters 3 and 4).

7.2 Test Sequence and Repair Recommendations

The KA820 module and the VAX 8200 system are designed to make repair easy and straightforward. If you suspect a hardware failure, you can try to identify the failing module before calling DIGITAL Field Service.

1. Check the light-emitting diodes (LEDs) on the modules in the VAXBI backplane. If a yellow LED is off, the indicated module is faulty. The KA820 module also provides red LEDs, which indicate a fault when lit, unless the CPU is in the console mode.
2. If the LEDs look right, run self-test by typing CTRL/P and then T on the console terminal. See Chapter 3 for an explanation of self-test on the KA820 module.
3. If self-test reveals no fault, and you suspect a fault on the KA820 module, boot EVKAA, the VAX Hard-Core Instruction Test, from a diskette in the RX50 drive (see Section 7.3).
4. If EVKAA runs without error, boot VDS (EBSAA) stand-alone from the maintenance system disk (see Section 7.4).
5. If you cannot boot VDS from the maintenance system disk or another device connected to a VAXBI node, boot it from the RX50 diskette (secondary load path).
6. When you see the VDS prompt, DS›, run level 2 or 3 diagnostic programs to test the area of the system you think is faulty. If you suspect the KA820 module, run the remaining cluster exerciser programs in the following order:

   EVKAB
   EVKAC
   EVKAE
   EBKAX

   See Sections 7.6 through 7.9.
7. If the cluster exerciser programs do not reveal an error and you suspect a problem with the serial-line units,

   Boot VMS
   Run VDS on-line
   Run the serial-line unit test, EBDAN (see Section 7.10).

7.3 EVKAA, Hard-Core Instruction Test

The Hard-Core Instruction Test checks the set of instructions required to boot and run VDS.

Fifteen tests, arranged in a building block sequence, make up the Hard-Core Instruction Test. The program proceeds from a check of instructions that manipulate the PSL through tests of the INSQUE and REMQUE instructions and console transmitter functions.

7.3.1 Booting EVKAA on the Primary Processor

Boot and run EVKAA on the primary processor by typing the console commands shown in Example 7–1.

```
CTRL/P                    ! Halt the primary processor.
>>>I                      ! Initialize the primary processor.
>>>                       ! Insert the appropriate diskette
                          !   in the RX50 drive.
>>>B CSA1                 ! Boot and run EVKAA.
```

Example 7–1: Booting EVKAA on the Primary Processor

Console microcode loads and starts the program in the first good 64K-byte section of main memory at the base address plus 200 (hex). EVKAA then runs continuously, making a complete pass every 0.2 seconds, unless it finds a hardware error. After the first pass and subsequently after every 10 passes, the program sounds a beep on the console terminal and prints a message:

```
EVKAA Vx.x PASS nnn(X) DONE!
```

If EVKAA finds a hardware error, it prints a message on the console terminal and halts, forcing the KA820 module to return to the console mode.

```
??? ERROR TEST #nn, SUBTEST #nn (instruction-code) failed
(one-line description of failure)
EXPECTED DATA - xxxxxxxx
RECEIVED DATA - xxxxxxxx
?06
             PC =xxxxxxxx
>>>
```

If EVKAA finds a hardware fault, call DIGITAL Field service to repair the fault. If you must repair the system yourself, replace the KA820 module (see Appendix B), since the KA820 module is the unit most likely to be at fault if EVKAA fails. Then check the repair by running the full set of diagnostic programs that test the KA820 module.

7.3.2 EVKAA Prerequisites and Functions

The KA820 processor must meet the following conditions before EVKAA can run:

1. The load path from the RX50 diskette drive must be functional.
2. At least 64K-bytes of main memory, aligned on a page, must be valid.

3. The following instructions must be functional:

BEQL	BICL2	BICPSW	BISL
BISPSW	BRW	CLRL	HALT
INCL	MFPR	MOVL	MOVPSL
MOVW	MOVZBL	MTPR	TSTL
XORL3			

4. Three addressing modes must also be valid:

   (PC)+
   D (PC)
   R

7.4 Using VDS Stand-Alone

You should be able to boot VDS (EBSAA) on any KA820 processor in the VAX 8200 system if EVKAA runs without errors on that processor. VDS requires 512K-bytes of memory.

7.4.1 Booting VDS Stand-Alone on the Primary Processor

Boot VDS on the primary processor from the maintenance system disk by typing the commands shown in Example 7–2 on the console terminal. VDS identifies itself before prompting for operator input.

```
CTRL/P                                  ! Halt the primary processor.
>>>I                                    ! Initialize the primary processor.
>>>B/R5:10 <ddxn>                       ! Boot and run VDS from a device
                                        !  where <ddxn> identifies the boot
                                        !  device type, adapter node number,
                                        !  and boot device unit number (see
                                        !  Section 4.3.2 in Chapter 4).
. . 2 3 4 . . . . . . . . . . F         ! List of nodes found.
xxxxxxxx                                ! Size of VAXBI memory.

                          VAX DIAGNOSTIC SOFTWARE
                                PROPERTY OF
                        DIGITAL EQUIPMENT CORPORATION

                          CONFIDENTIAL AND PROPRIETARY

Use Authorized Only Pursuant to a Valid Right-to-Use License

COPYRIGHT, DIGITAL EQUIPMENT CORPORATION, 1985. ALL RIGHTS
RESERVED.

DIAGNOSTIC SUPERVISOR. ZZ-EBSAA-8.1 6 MAR 1985 13:22:45

DS>                                     ! VDS prompt symbol.
```

Example 7–2: Booting VDS Stand-Alone on the Primary Processor

NOTE

The VAX diagnostic software should be permanently installed in the SYSMAINT directory.

If the VDS bootstrap fails, boot VDS through the secondary load path, using an RX50 diskette. Insert the appropriate diskette in the RX50 drive and type:

```
>>> B CSA1
```

on the console terminal.

7.4.2 Booting VDS Stand-Alone on an Attached Processor

If you want to test an attached KA820 processor in a VAX 8200 system, you must first boot VDS on the primary processor using console commands, then boot VDS on the attached processor using the VDS BOOT n command, and then run diagnostic programs.

```
CTRL/P                                  ! Halt the primary processor.
>>> I                                   ! Initialize the primary processor.
>>> B/R5=10 (dxn)                       ! Boot VDS on the primary processor.
. . 2 3 4 . . . . . . . . . . F         ! Nodes found.
xxxxxxxx                                ! Size of VAXBI memory.

                          VAX DIAGNOSTIC SOFTWARE
                                PROPERTY OF
                        DIGITAL EQUIPMENT CORPORATION

                          CONFIDENTIAL AND PROPRIETARY

Use Authorized Only Pursuant to a Valid Right-to-Use License

COPYRIGHT, DIGITAL EQUIPMENT CORPORATION, 1985. ALL RIGHTS
RESERVED.

DIAGNOSTIC SUPERVISOR. ZZ-EBSAA-8.1 6 AUG 1985 14:22:45

DS) BOOT n                              ! Boot VDS on the attached processor
                                        !   at node n.
  .
  .
  .
DS) RUN (exxxx)                         ! Run diagnostic program (exxxx)
```

Example 7–3: Booting VDS Stand-Alone on an Attached Processor

7.4.3 Help

VDS provides a help feature that gives information on diagnostic programs and VDS commands. Examples:

- HELP
- HELP DEVICE KA820
- HELP SET FLAGS
- HELP EVKAB

7.4.4 Attaching and Selecting the KA820 Module

You must attach and select the KA820 module before running level 2, 2R, or 3 diagnostic programs to test it. You can attach all devices in the system by running the autosizer in the default mode in response to the DS› prompt.

```
DS› RUN EVSBA
```

The autosizer identifies all native DIGITAL devices on the system and passes this information to VDS.

Or, you can attach and select the KA820 module specifically, using the following command format in response to the DS› prompt:

```
DS› ATTACH KA820 HUB KAn msiz id rx0 rx1
```

where:

n	is the KA820 node ID (hex) (for error message print out)
msiz	is the size of memory in kilobytes (decimal)
id	is the CPU's node ID (hex)
rx0	is Y or N (Is scratch diskette present in RX50 drive 0?)
rx1	is Y or N (Is scratch diskette present in RX50 drive 1?)

```
DS› SELECT KAn
```

If you attach an external wrap connector on the back panel of the main cabinet for serial-line unit 1, 2, or 3, attach and select each wrapped serial-line unit as follows:

```
DS› ATTACH SLU KAn TCx0 extlp rate
```

where:

TCA0	is serial-line unit 1
TCB0	is serial-line unit 2
TCC0	is serial-line unit 3
n	is the KA820 node ID
extlp	is Y or N (Is an external wrap connector provided?)
rate	is the desired baud rate (150 to 19200)

```
DS› SELECT TCx0
```

7.4.5 Flags in VDS

The functions of VDS command flags are consistent across all diagnostic programs, but the functions of event flags vary with each diagnostic program.

You can set command flags with the SET FLAGS command, according to the following format:

```
DS› SET FLAGS ‹arg-list›
```

where ‹arg-list› identifies the command flags to be set (for example, SET FLAGS OPERATOR, QUICK). The SHOW FLAGS command lets you see the status of each VDS flag.

Some diagnostic programs use event flags. You can set event flags with the SET EVENT command, according to the following format:

```
DS> SET EVENT <arg-list>
```

where <arg-list> identifies the event flags to be set (for example, SET EVENT 1, 3, 4). The SHOW EVENT command lets you see the status of each event flag.

7.4.6 Test Repetitions

Each diagnostic program that runs with VDS normally completes one pass. You can stop it by typing CTRL/C. If the program finds an error, it prints an error message, aborts the failing subtest, and continues with the next subtest or test, unless the HALT flag is set. If the error is severe, the test or program may be aborted, returning control to VDS.

If you want to run a specific test or set of tests, instead of running the tests in the default section, use the /TEST qualifier with the RUN command:

```
DS> RUN <exxxx>/TEST=<first>[:<last>]
```

where <exxxx> is the program code and <first> and <last> are decimal test numbers.

You can specify a program section to execute by using the /SECTION qualifier with the RUN command:

```
DS> RUN <exxxx>/SECTION=<section-name>
```

You can specify how many times a program runs by using the /PASSES qualifier with the RUN command:

```
DS> RUN <exxxx>/PASSES=<count>
```

where <count> indicates the number of passes (decimal) to run.

Use a pass count of 0 to run a program continuously. If you do this, the program will run until you type CTRL/C or a failure occurs with the HALT flag set.

NOTE

You can replace the RUN command with the combination of LOAD and START, where START takes the same parameters as RUN. After loading a program with LOAD, you can type SHOW TESTS, asking VDS to list the test names and numbers on the terminal.

7.5 Using VDS On-line

Before you run any level 2R diagnostic programs you must run VDS on-line, under the VMS operating system. Then run the program you require. Example 7–4 shows the procedure for running VDS on-line from the SYSMAINT directory and then running EVKAB. Example 7–5 shows the procedure for running VDS on-line from the RX50 diskette drive (the secondary load path) and then running EVKAB. VDS identifies itself before prompting for operator input. In either case you must begin by logging in to the field service account.

```
Username: FIELD
Password: SERVICE

$ SHOW DEFAULT
SYS$SYSROOT:[SYSMAINT]
$ RUN EBSAA                          ! Load and start VDS.

                          VAX DIAGNOSTIC SOFTWARE
                                PROPERTY OF
                        DIGITAL EQUIPMENT CORPORATION

                          CONFIDENTIAL AND PROPRIETARY

Use Authorized Only Pursuant to a Valid Right-to-Use License

COPYRIGHT, DIGITAL EQUIPMENT CORPORATION, 1985. ALL RIGHTS
RESERVED.

DIAGNOSTIC SUPERVISOR. ZZ-EBSAA-8.1 6 SEP 1985 12:42:45

DS) SHOW LOAD                        ! Identify the default load device
                                     !   and directory.
DUA0:[SYSMAINT]
DS) ATTACH KA820 HUB KA2 2048 2 Y N
                                     ! Attach the KA820. Insert
                                     !   a scratch diskette in RX50
                                     !   unit 0.
DS) SELECT KA2                       ! Select the KA820 at node 2.
DS) LOAD EVKAB                       ! Load the EVKAB diagnostic.
DS) SHOW TESTS                       ! List the EVKAB tests.
. Test 1: BRB
. Test 2: BRW
. Test 3: BBC
.............
.............
. Test 161: EDITPC
DS) START                            ! Run  EVKAB.
```

Example 7–4: Running VDS On-Line from the SYSMAINT Directory

```
Username: FIELD
Password: SERVICE
$ MOUNT/OVERRIDE=ID CSA1:
$ SET DEFAULT CSA1:[SYSMAINT]
$ RUN EBSAA                                  ! Load and start VDS.

                          VAX DIAGNOSTIC SOFTWARE
                              PROPERTY OF
                        DIGITAL EQUIPMENT CORPORATION
                        CONFIDENTIAL AND PROPRIETARY

Use Authorized Only Pursuant to a Valid Right-to-Use License

COPYRIGHT, DIGITAL EQUIPMENT CORPORATION, 1985. ALL RIGHTS
RESERVED.

DIAGNOSTIC SUPERVISOR. ZZ-EBSAA-8.1 6 OCT 1985 23:17:25

DS> ATTACH KA820 HUB KA2 2048 2 Y N          ! Attach the KA820.
DS> SELECT KA2                               ! Select the KA820 module
                                             !   at node 2.
DS> RUN EVKAB                                ! Run the EVKAB diagnostic.
```

Example 7–5: Running VDS On-line from RX50 Diskette Drive

NOTE

You cannot run VDS on an attached processor on-line, and you cannot run the autosizer on-line.

7.6 EVKAB, VAX Basic Instruction Exerciser

You can run EVKAB (level 2) either stand-alone or on-line. With VDS running and the KA820 module attached and selected (see Section 7.4.4), run EVKAB as follows. In response to the DS> prompt, type:

```
DS> RUN EVKAB                                ! Run the VAX basic instruction
                                             !   exerciser.
```

Run EVKAA before running EVKAB to assure a systematic test of the KA820 module.

EVKAB tests most of the nonprivileged VAX instruction set, except for the floating point instructions. It consists of 161 tests, each of which checks execution of a specific VAX instruction.

Both EVKAB and EVKAC respond to the command flags; they also respond to event flags 1 through 6, with the following functions:

1. Disable messages relating to the floating-point accelerator.
2. Disable interrupts from the interval timer.
3. Enable interrupts from the interval timer while page faulting is enabled.
4. Enable the continuation of a subtest after error detection (normally the program aborts a subtest when it detects an error).

5. Disable execution of the DIVP instruction while the interval timer is interrupting.

6. Prompt for tests to be executed if the OPERATOR flag is set. Begin testing by typing ⟨CR⟩ twice after selecting the tests you want.

7.7 EVKAC, Floating-Point Instruction Exerciser

You can run EVKAC (level 2) either stand-alone or on-line; VDS must be running, and the KA820 module must be attached and selected (see Section 7.4.4). In response to the DS⟩ prompt, type:

```
DS⟩ RUN EVKAC                    ! Run the VAX floating-point
                                 !  instruction exerciser.
```

Run EVKAA and EVKAB before running EVKAC to assure a systematic test of the KA820 module.

EVKAC consists of 98 tests, each of which tests a floating-point instruction.

EVKAC responds to the command flags and to event flags 1 through 6, as explained in Section 7.6.

7.8 EVKAE, VAX Privileged Architecture Exerciser

EVKAE (level 3) runs only in the stand-alone mode; VDS must be running without VMS, and the KA820 module must be attached and selected (see Section 7.4.4). In response to the DS⟩ prompt, type:

```
DS⟩ RUN EVKAE                    ! Run the VAX privileged
                                 !  architecture exerciser.
```

Run EVKAA, EVKAB, and EVKAC before running EVKAE to assure a systematic test of the KA820 module.

EVKAE consists of 12 tests grouped in 8 sections:

- DEFAULT
- MEM__MGT
- EXCEPTIONS
- PROCESS
- REGISTERS
- CHANGE__MODE
- TIMER
- CONTEXT

7.9 EBKAX, VAX 8200-Specific Cluster Exerciser

EBKAX (level 3) runs only in the stand-alone mode; VDS must be running without VMS, and the KA820 module must be attached and selected (see Section 7.4.4)

EBKAX tests the portions of the KA820 module not specified by the VAX architecture, including the serial-line units. You can expand the test of serial-line unit 1, 2, or 3 by attaching a wrap connector to the plug for that serial-line unit on the rear of the main unit. However, you cannot use a wrap connector on serial-line unit 0 when you run EBKAX, because the console terminal requires that line for communication with the KA820 module and with VDS.

If you use wrap connectors, use the serial-line unit specific ATTACH and SELECT commands for each wrapped serial-line unit as follows:

```
DS) ATTACH SLU KAn TCx0 extlp rate
```

where:

TCA0	is serial-line unit 1
TCB0	is serial-line unit 2
TCC0	is serial-line unit 3
n	is the KA820 node ID
extlp	is Y or N (Is an external wrap connector provided?)
rate	is the desired baud rate (150 to 19200)

```
DS) SELECT TCx0
DS) RUN EBKAX                          ! Run the KA820-specific cluster
                                       !  exerciser.
```

Example 7–6: Running EBKAX

Run EVKAA, EVKAB, EVKAC, and EVKAE before running EBKAX to assure a systematic test of the KA820 module.

EBKAX consists of 57 tests grouped in three sections:

- DEFAULT
- MANUAL INTERVENTION
- MEMORY

7.10 EBDAN, KA820 Serial-Line Unit Diagnostic

EBDAN (level 2R) runs only in the on-line mode, with the VAX Diagnostic Supervisor running under VMS. You must attach both the KA820 module and the serial-line units to be tested, and then select the serial line units to be tested. First attach the KA820 module as explained in Section 7.5. Then load EBDAN, the serial-line unit diagnostic. Then attach and select each serial-line unit to be tested. And then start the test. The following example shows the attaching, selecting, and testing of serial-line unit 1.

```
DS) ATTACH KA820 HUB 2 2048 2 Y N          ! Attach the KA820 module.
DS) LOAD EBDAN                             ! Load the serial-line unit
                                           !   diagnostic.
DS) ATTACH SLU                             ! Attach a serial-line unit.
DEVICE LINK? KA2                           ! The serial-line unit to be
                                           !   tested is on the KA820
                                           !   module at node 2.
DEVICE NAME? TCA0                          ! Serial-line unit 1.
SERIAL-LINE UNIT EXTERNALLY WRAPPED? Y     ! Connect a loopback connector.
BAUD RATE? 1200                            ! Use baud rate 1200.
DS) SELECT TCA0                            ! Select serial-line unit 1 for
                                           !   testing.
DS) START                                  ! Run EBDAN.
```

Example 7–7: Running EBDAN

Run the cluster exerciser programs before running EBDAN to ensure a thorough test of the serial-line units.

EBDAN consists of three tests:

- Per line internal data loopback test
- Per line external data loopback test
- Multiple line external loopback test

If you do not select a specific test with the START command, EBDAN executes the tests according to the information you have provided with the ATTACH and SELECT commands.

Appendix A
KA820 Module I/O Pins and Cables

A.1 Module I/O Pin Definitions

The KA820 module contains 300 module I/O pins along one edge. These pins are arranged in five segments (A, B, C, D, E). Each segment contains 60 pins, 30 on each side of the module.

As you look at the card cage from the backplane side, with the cam at the top, pins corresponding to segments A through E appear from top to bottom. Segments A and B carry the VAXBI signals. Segments C, D, and E carry module-specific signals.

Each of the segments C, D, and E supplies signals to two 30-pin connectors. For purposes of cable positioning, refer to the left connector for segment C as C1 and the right connector as C2. Connectors D1, D2, E1, and E2 are identified similarly.

Segment C carries signals for the four serial-line units. Only connector C2 is used. Segment D carries PCI bus signals and signals that go to the control panel and the PCM module. Segment E carries only the performance monitor enable signal on connector E2.

Figures A–1 through A–5 show the module I/O pin segments from the backplane side of the card cage.

GND	46	–	–	16	BI D31 L	GND	31
	47	–	–	17	BI D30 L		32
GND	03	–	–	18	GND		33
GND	49	–	–	19	GND		34
GND	50	–	–	20	BI D26 L		35
	51	–	–	21	BI D22 L	GND	36
	52	–	–	22	BI D20 L		37
GND	53	–	–	23	BI D18 L	GND	38
	54	–	–	24	BI D17 L		39
GND	55	–	–	25	BI D14 L	GND	40
BI SPARE L	56	–	–	26	BI D12 L		41
GND	57	–	–	27	BI D10 L	GND	42
+5 V	58	–	–	28	+5 V	+5 V	43
GND	59	–	–	29	+5 V	+5 V	44
BI D03 L	60	–	–	30	BI D16 L	BI D08 L	45

GND	31	–	– 01 BI D29 L
	32	–	– 02 BI D28 L
	33	–	– 48
	34	–	– 04 BI D27 L
	35	–	– 05 BI D25 L
GND	36	–	– 06 BI D23 L
	37	–	– 07 BI D21 L
GND	38	–	– 08 BI D19 L
	39	–	– 09 BI D15 L
GND	40	–	– 10 BI D24 L
	41	–	– 11 BI D13 L
GND	42	–	– 12 BI D11 L
+5 V	43	–	– 13 +5 V
+5 V	44	–	– 14 BI D07 L
BI D08 L	45	–	– 15 BI D06 L

MLO-425-85

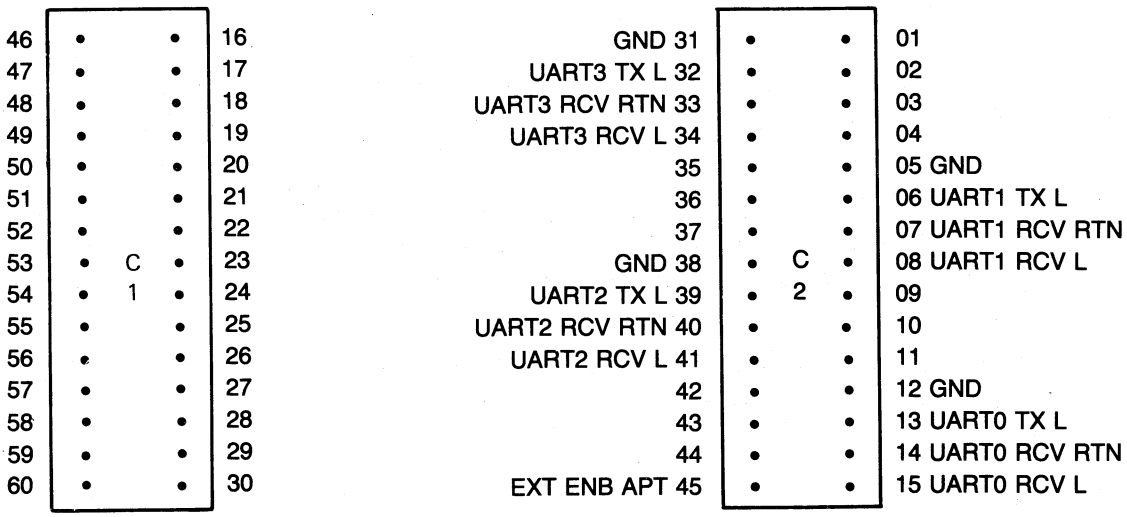
Figure A–1: Module I/O Pins on Segment A Viewed from the Backplane

BI D05 L 46	–	–	16 BI D04 L
BI ID3 H 47	–	–	17 BI D01 L
GND 48	–	–	18 BI I2 L
BI ID1 H 49	–	–	19 BI I0 L
+12 V 50	–	–	20 BI CNF1 L
BI BAD L 51	–	–	21 BI D09 L
GND 52	–	–	22 BI CNF0 L
08	–	–	23
BI RESET L 54	–	–	24
GND 55	–	–	25 BI ECL VCC H
GND 56	–	–	26 BI TIME H
GND 57	–	–	27 BI PHASE H
GND 13	–	–	28 GND
59	–	–	29 GND
60	–	–	30

BI D02 L 31	–	–	01 BI D00 L
GND 32	–	–	02 BI P0 L
BI ID2 H 33	–	–	03 BI I1 L
GND 34	–	–	04 BI CNF2 L
BI ID0 H 35	–	–	05 BI BSY L
BI STF L 36	–	–	06 BI NOARB L
-12 V 37	–	–	07 BI I3 L
38	–	–	53
39	–	–	09 BI DCLO L
BI ACLO L 40	–	–	10 GND
BI TIME L 41	–	–	11 GND
BI PHASE L 42	–	–	12 GND
GND 43	–	–	58 GND
GND 44	–	–	14
45	–	–	15

MLO-426-85

Figure A–2: Module I/O Pins on Segment B Viewed from the Backplane



MLO-427-85

Figure A–3: Module I/O Pins on Connectors C1 and C2 Viewed from the Backplane

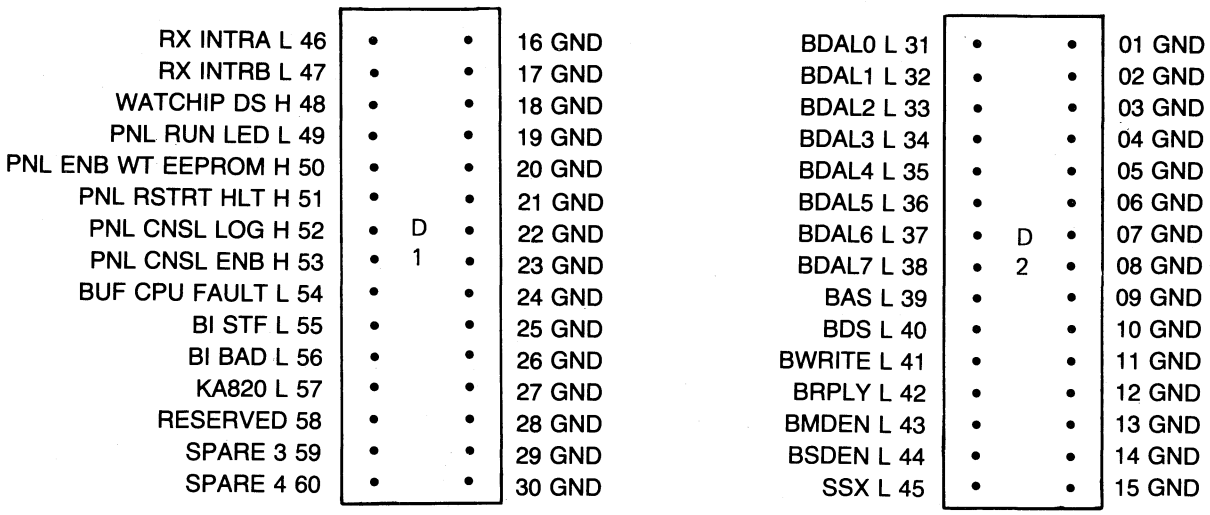


Figure A–4: Module I/O Pins on Connectors D1 and D2 Viewed from the Backplane

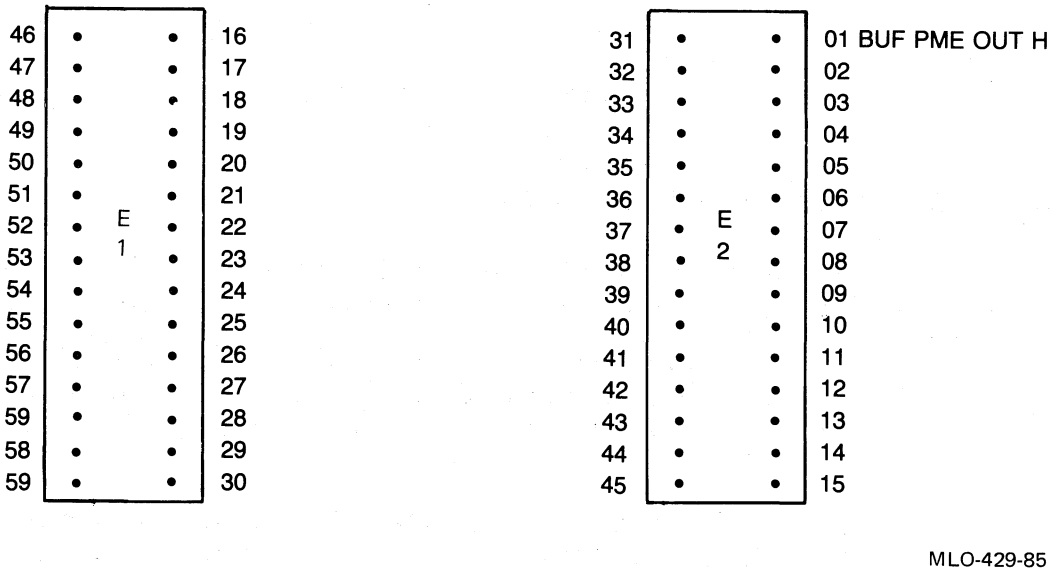
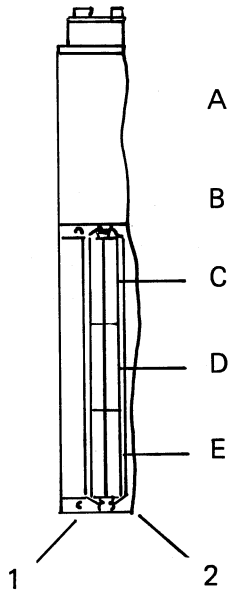


Figure A–5: Module I/O Pins on Connectors E1 and E2 Viewed from the Backplane

Figure A–6 shows the backplane slots viewed from the bottom side of the VAXBI module card cage.



MLO-430A-85

Figure A–6: A Backplane Slot Shown from the Backplane Side of the Card Cage

A.2 Cables Related to the KA820

Two sets of cables run from the C and D connectors on slot K1J1 of the VAXBI card cage. See Appendix B for instructions on how to gain access to the cables.

Four cylindrical cables extend from the C2 connector at slot K1J1 to four separate serial-line unit connectors on the IOCP (I/O connector panel) as shown in Figure A–7. The IOCP connectors convert the serial lines from EIA RS423 to EIA RS232.

Two flat 30-wire ribbon cables extend from the D connectors at slot K1J1 to the J1P1 and J2P2 connectors on the PCM module in the KK810 control panel assembly, as shown in Figure A–7. These cables carry the PCI bus signals from the KA820 module to the watch chip on the PCM module and then out to the optional RCX50 controller.

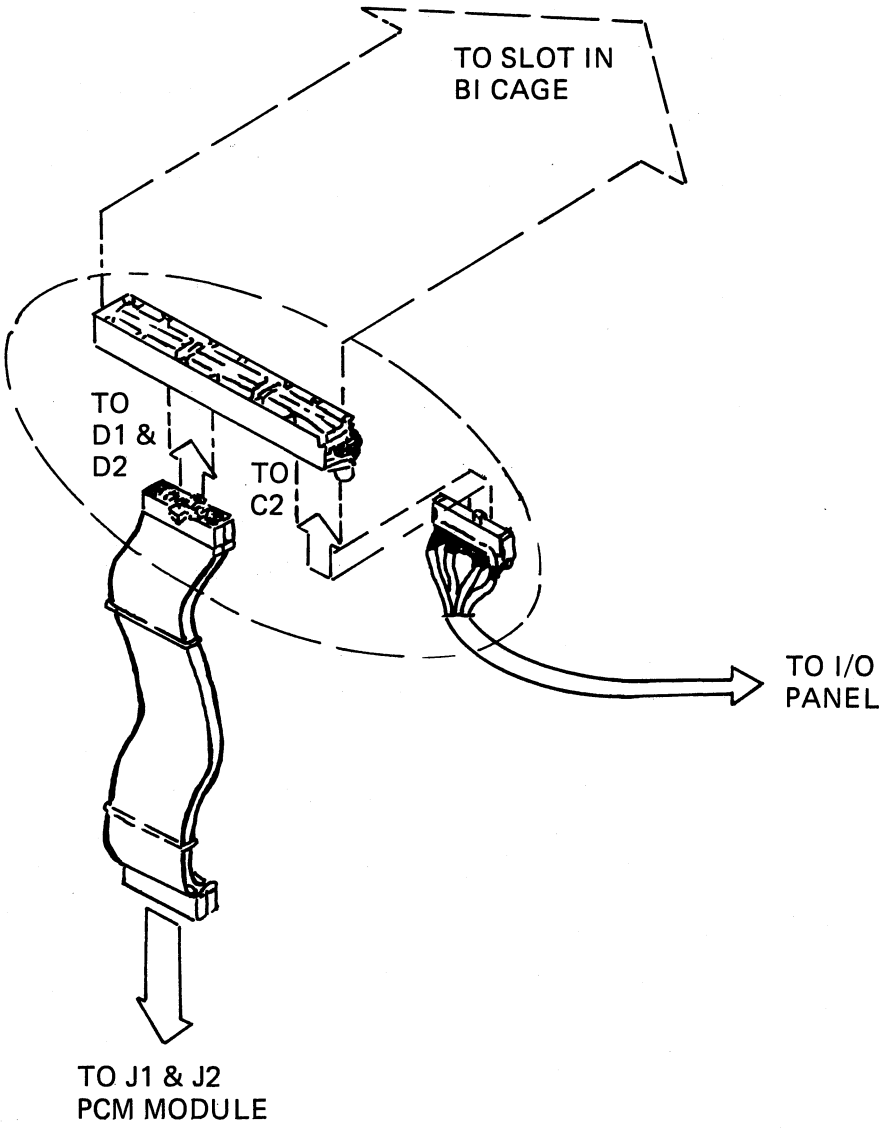


Figure A–7: Cabling for C and D Connectors

In a multiprocessor system, only the KA820 module in slot K1J1 uses these cables. The other KA820 modules do not require cables.

Appendix B
Module Installation and Access to Cables

B.1 Module Installation and Replacement

DIGITAL Field Service engineers install and repair the VAX 8200 system. However, if you determine that a KA820 module in the system is faulty, you can replace it as follows:

1. Turn off power to the VAX 8200 system by rotating the upper key switch counterclockwise to the vertical position.
2. Switch the circuit breaker on the bottom panel at the rear of the main unit to the off position.
3. Pull out the stabilizer bar at the bottom of the cabinet. See the *VAX 8200 System Owner's Manual* for illustrations.
4. Remove the back panel with an Allen wrench (5/32 inch).
5. Release the locking mechanism on the slide and push out the processor drawer from the rear of the main unit.
6. Remove the 14 screws from the panel that covers the top of the drawer, and remove the panel.
7. Attach a ground strap from your wrist to the system ground to avoid damaging the module.
8. If the faulty KA820 module is the primary processor, lift the first black handle on the right side of the module cage. This releases the module in the first slot, K1J1.

   If the faulty KA820 module is an attached processor, lift the black handle for the corresponding VAXBI slot.
9. Remove the module in the exposed slot by lifting it clear.
10. Insert the new KA820 module in the same slot, with the component side of the module to the right.
11. Close the black handle over the new KA820 module.
12. Replace the metal panel on the drawer, and insert and tighten the 14 screws that hold it.

13. Slide the processor drawer back into the cabinet.
14. Push the stabilizer bar back inside the cabinet.

B.2 Gaining Access to the Cables

You can gain access to the cables that supply the KA820 module in slot K1J1 as follows:

1. Turn off power to the VAX 8200 system by rotating the upper key switch counterclockwise to the vertical position.
2. Switch the circuit breaker on the bottom panel at the rear of the main unit to the off position.
3. Pull out the stabilizer bar at the bottom of the cabinet. See the *VAX 8200 System Owner's Manual* for illustrations.
4. Remove the back panel with an Allen wrench (5/32 inch).
5. Release the locking mechanism on the slide and push out the processor drawer from the rear of the main unit.
6. Pull the latches on the drawer sides and raise the drawer to the vertical position.
7. Remove the bottom panel of the processor drawer to expose the bottom of the VAXBI module cage and the cables that supply the KA820 module in slot K1J1.
8. When you are ready to close the main unit, replace the bottom panel, lower the drawer to the horizontal position, and push it back inside the main unit.
9. Push the stabilizer bar back inside the cabinet.

Appendix C
Drive Load Characteristics of Off-Board Signals

C.1 Serial-Line Unit Signals

The UART drivers and receivers for the KA820 serial-line units provide EIA RS423 electrical characteristics. These signals are converted to the RS232 standard by connectors on the back panel. Table C–1 shows the electrical characteristics of the output signals.

Table C–1: Serial-Line Unit Output Signal Characteristics

Output Characteristics	Minimum	Maximum	Units
Voltage low	−5.0	−6.0	Volts
Voltage high	5.0	6.0	Volts

C.2 PCI Bus Off-Board Signals

Eighteen PCI signals are buffered and run off the KA820 module to the watch chip and the RCX50 controller. The PCI signal names change as they leave the module, to conform to the signal names used by the RCX50 controller. The buffer circuits are low-power Schottky devices.

Table C–2 lists the module I/O pin numbers and and names of the PCI signals. Table C–3 lists the pin numbers and names of other signals that run off the KA820 module. Tables C–4 through C–8 list the electrical characteristics of these off-board signals.

Table C–2: PCI Bus Off-board Signals

On-board Signal Name	External Signal Name	Type of Signal at I/O Connection	Driver/ Receiver	Module I/O Pin
PCI DAL 7:0 H	BDAL 7:0 H	Tristate, bidirectional	8307	D38:D31
PCI DAS L	BDS L	Driver output	LS244	D40
RX CS L	SSX L	Driver output	LS244	D45
RX DALO L	BMDEN L	Driver output	LS244	D43
RX DALI L	BSDEN L	Driver output	LS244	D44
PCI READ H	BWRITE L	Driver output	LS244	D41
PCI ALE H	BAS L	Driver output	LS244	D39
RTC DS H	WATCHIP DS H	Driver output	LS244	D48
BUF RX INTRA L	RX INTRA L	TTL input	LS244	D46
BUF RX INTRB L	RX INTRB L	TTL input	LS244	D47
PCI READY L	BRPLY L	TTL input with 470 ohm pull-up	7417	D42

Table C–3: Other Off-board Signals

On-board Signal Name	External Signal Name	Type of Signal at I/O Connection	Driver/ Receiver	Module I/O Pin
PME OUT H	BUF PEM OUT H	Driver output	LS244	E1
BUF BI STF L	BI STF L	Driver output	LS244	D55
ENB APT L	EXT ENB APT L	Driver output	LS244	C45
CNSL LOG H	PNL CNSL LOG H	Driver output	LS244	D52
ENB WT EEPROM H	PNL ENB WT EEPROM H	Driver output	LS244	D50
CNSL ENB H	PNL CNSL ENB H	Driver output	LS244	D53
RSTRT HLT H	PNL RSTRT HLT H	Driver output	LS244	D51
RUN L	PNL RUN LED L	OC driver, 2000-ohm pull-up	7417	D49
BI BAD H	BI BAD L	OC driver, no pull-up	LS38	D56

Table C–4: Driver Output Voltages

Output Voltage Level	Output Current	Minimum Voltage	Typical Voltage	Maximum Voltage	Units
Voh	Ioh = − 3 mA	2.4	3.4	—	Volts
Voh	Ioh = −15 mA	2.0	—	—	Volts
Vol	Iol = 12 mA	—	0.25	0.4	Volts
Vol	Iol = 24 mA	—	0.35	0.5	Volts

Table C–5: Driver Output Current

Output Current Level	Output Voltage	Minimum Current	Maximum Current	Units
Isc (short circuit)	Vcc = 5.5 V	−40	−225	mA
Ioh	Vo = 2.7 V	—	−15	mA
Iol	Vo = 0.4 V	—	24	mA

Table C–6: PCI DAL ⟨7:0⟩ Lines Bidirectional Voltage Levels

Output Voltage Level	Output Current	Minimum Voltage	Maximum Voltage	Units
Voh	Ioh = − 5 mA	2.7	—	Volts
Voh	Ioh = −10 mA	2.4	—	Volts
Vol	Iol = 20 mA	—	0.4	Volts
Vol	Iol = 48 mA	—	0.5	Volts

Table C–7: PCI DAL ⟨7:0⟩ Lines Bidirectional Current Levels

Output Current Level	Output Voltage	Minimum Current	Maximum Current	Units
Isc (short circuit)	—	−25	−150	mA
Iozh (tristate)	Vo = 4.0 V	—	200	mA
Iozl	Vo = 0.4 V	—	−200	mA

Table C–8: PCI Bus Input Signal Voltage and Current Levels

Input Signal Levels	Minimum	Maximum	Units
Vih	2.0	—	Volts
Vil	—	0.8	Volts
Iih at Vi = 2.7 V	—	20.0	uA
Iil at Vil = 0.4 V	—	−10.2	uA

Appendix D
BIIC Registers

System software can program the BIIC registers on the KA820 module to control processor participation in transactions and to handle interrupts on the VAXBI bus. Each BIIC register has a node space address (bb + offset, where bb is the node base address) and a node private space address (2008 0000 + offset). The BIIC does not support the lock function, so it treats IRCI and UWMCI commands like RCI and WCI commands (see Tables 2–4 and 2–5 in Chapter 2 for lists of VAXBI commands used by the KA820 module).

The register bit descriptions that follow use a code to define read, write, and functional characteristics:

DCLOC Cleared following a successful self-test following the deassertion of BI DC LO L.

DCLOL Loaded on the deassertion of BI DC LO L.

DCLOS Set following a successful self-test following the deassertion of BI DC LO L.

DMW Writeable in diagnostic mode.

DS Disable selection: when an enable bit of this type is cleared, the BIIC suppresses the appropriate BCI SEL L and BCI SC 2:0 L assertion and inhibits any KA820 module response to transactions corresponding to the cleared enable bit.

RO Read-only.

R/W Read/Write.

SC Special case: the operation is defined in the detailed description.

W1C Write one to clear. You cannot set a W1C bit.

D.1 Device Register, DTYPE (R/W, DMW, DCLOL)

Processor initialization microcode loads this register; system software should set bit 16 after loading secondary patches.

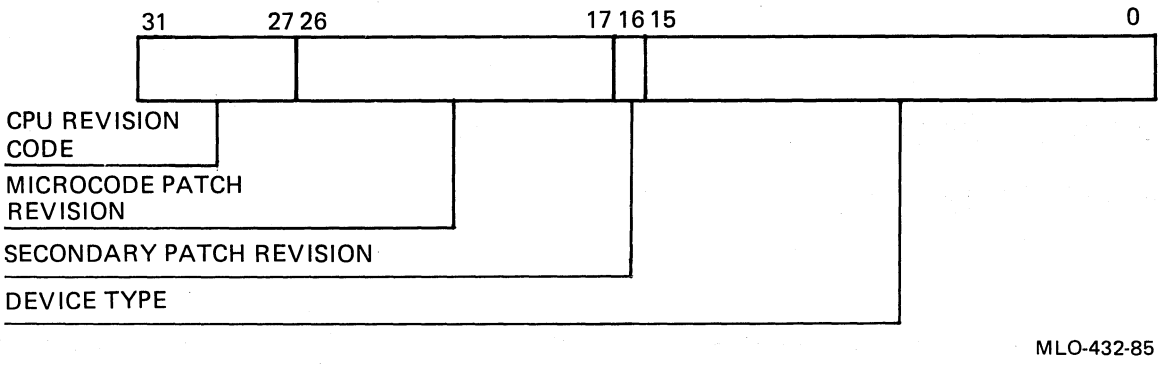


Figure D–1: Device Register (DTYPE)

Node private space address	2008 0000
Nodespace address	bb + 00
CPU Revision Code Bits ⟨31:27⟩	Loaded by processor initialization microcode.
Microcode Patch Revision Bits ⟨26:17⟩	Loaded by processor initialization microcode.
Secondary Patch Revision Bit ⟨16⟩	When bit ⟨16⟩ is set, secondary patches are not needed, or they are needed and have been loaded. When bit ⟨16⟩ is clear, secondary patches are needed but have not been loaded. Processor initialization microcode writes this bit according to data in location 2009 816C in the EEPROM. Code that loads secondary patches should set bit ⟨16⟩ at the same time.
Device Type Bits ⟨15:0⟩	Loaded by processor initialization microcode. The Device Type field identifies the VAXBI node by type. The KA820 module should have 0105 (hex) in Device Type field.

D.2 VAXBI Control and Status Register, VAXBICSR

Processor initialization microcode writes this register to clear the Broke bit following a successful self-test. System software should set the Hard and Soft Error Interrupt Enable bits following initialization to enable error interrupts.

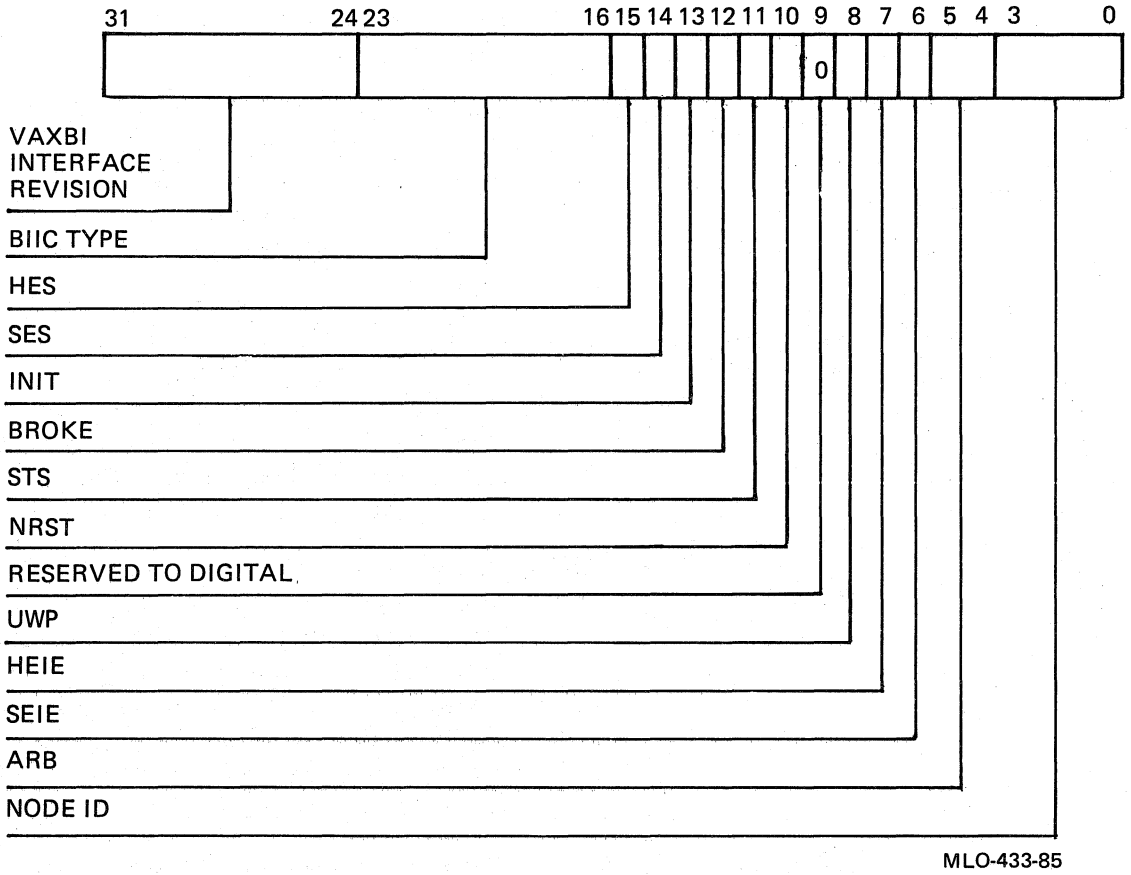


Figure D–2: VAXBI Control and Status Register (VAXBICSR)

Node private space address: 2008 0004

Nodespace address: bb + 04

VAXBI Interface Revision (RO)
Bits ⟨31:24⟩

This field identifies the revision level of the BIIC.

VAXBI Interface Type (RO)
Bits ⟨23:16⟩

The type of VAXBI interface chip (BIIC) used on this node. This field is read as 00000001 (binary) in this implementation.

HES (RO)
Bit ⟨15⟩

Hard Error Summary. When set, HES indicates that one or more of the hard error bits (except TDF) in the Bus Error Register are set.

SES (RO)
Bit ⟨14⟩

Soft Error Summary. When set, SES indicates that one or more of the soft error bits in the Bus Error Register are set.

INIT (W1C, DCLOS)
Bit ⟨13⟩

INIT Bit. INIT is not used by the KA820 module.

BROKE (W1C, DCLOS)
Bit ⟨12⟩

Broke bit. Broke indicates that the KA820 module has not successfully completed its self-test. Microcode clears Broke following successful completion of self-test.

STS (R/W, DCLOS)
Bit ⟨11⟩

Self-Test Status bit. STS shows the result of the BIIC internal self-test. This bit is set to 1 if self-test passes, and it directly enables the BIIC VAXBI driver circuits. Since this is a normal R/W bit, it can be altered by a write command directed at this register.

NRST (SC)
Bit ⟨10⟩

Node Reset. Writing 1 to this bit forces the initiation of the BIIC self-test, as well as assertion of BCI DC LO L, which initiates self-test on the rest of the KA820 module. Read commands return 0 for NRST. The BIIC clears STS when it sets NRST, to allow proper recording of the self-test results.

UWP (W1C, DCLOC)
Bit ⟨15⟩

Unlock Write Pending bit. UWP indicates that this KA820 module has completed an IRCI transaction, but has not issued a subsequent UWMCI transaction. If the KA820 module performs a UWMCI transaction when the UWP bit is not set, the ISE bit in the Bus Error Register will be set, and the BIIC will complete the UWMCI transaction in the normal manner.

HEIE (R/W, DCLOC)
Bit ⟨6⟩

Hard Error Interrupt Enable bit. System software normally sets HEIE to enable an error interrupt when HES is asserted.

SEIE (R/W, DCLOC)
Bit ⟨6⟩

Soft Error Interrupt Enable bit. System software normally sets SEIE to enable an error interrupt when SES is asserted.

ARB (R/W, DCLOC)
Bits ⟨5:4⟩

Arbitration Control bits. ARB determines the mode of arbitration to be used by the KA820 module, as follows:

Table D–1: Arbitration Control Codes

Bit 5	Bit 4	Arbitration Scheme
0	0	Dual round-robin arbitration
0	1	Fixed-high priority (reserved)
1	0	Fixed-low priority (reserved)
1	1	Disable arbitration (reserved)

System software normally writes 0s in these bits.

NODE ID
(RO, DMW, DCLOL)
Bits ⟨3:0⟩

Node Identification field. The BIIC loads the Node ID field from the BCI I ⟨3:0⟩ H lines during the last cycle in which BCI DC LO L is asserted. Node ID reflects the encoded node ID plug that is plugged into the VAXBI backplane at the slot used by this KA820 module.

Node 2 is the normal node ID assigned to the VAX 8200 primary processor, according to DIGITAL convention.

D.3 Bus Error Register, BER (W1C, DCLOC)

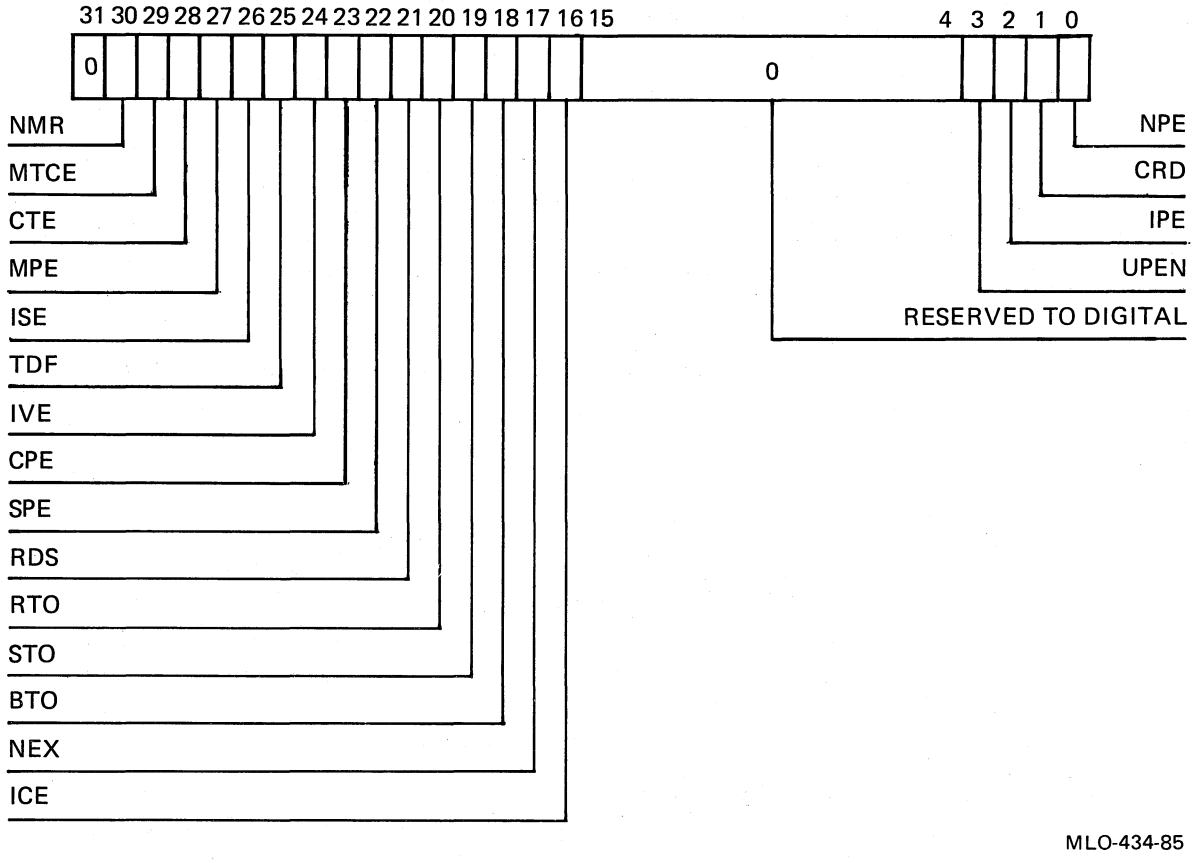


Figure D–3: Bus Error Register (BER)

When the BIIC detects an error, it sets the appropriate Bus Error Register (BER) bit and generates an interrupt, if enabled. Software can read this register to determine the cause of the error when it services a VAXBI error interrupt. System software should set the Hard and Soft Error Interrupt Enable bits of each VAXBI node in the system by setting bits ⟨6⟩ and ⟨7⟩ in the VAXBICSR of each node.

System software must clear the error bits in the BER after reading the register. After clearing the appropriate bit, software should reread the BER to determine whether another bit has been set in the meantime. Alternatively, it can read the Hard and Soft Error Summary bits in the appropriate VAXBICSR to determine if another error bit has been set. Bits ⟨31:16⟩ indicate hard errors; bits ⟨2:0⟩ indicate soft errors.

During initialization, system software can clear all error bits by writing 3FFF 0007 (hex) to the BER of each VAXBI node.

Node private space address: 2008 0008

Nodespace address: bb + 08

D.3.1 Bus Error Register Hard Error Bits

NMR (W1C, DCLOC) Bit ⟨30⟩
: NO ACK to Multi-Responder Command Received. NMR is set if the KA820 module receives a NO ACK command confirmation for an INVAL, INTR, IPINTR, STOP, BDCST, or RESERVED command.

MTCE (W1C, DCLOC) Bit ⟨29⟩
: Master Transmit Check Error. During the cycles of a transaction in which the KA820 module is the only source of data on the VAXBI D, I and P lines, the BIIC verifies that the transmitted data matches the data received from the VAXBI bus. If there is no match, the BIIC sets MTCE. The BIIC does not perform this check on the assertion of the encoded ID on the I lines during the embedded ARB cycle.

CTE (W1C, DCLOC) Bit ⟨28⟩
: Control Transmit Error. The BIIC sets CTE when there is an assertion of the NO ARB, BSY, or the CNF ⟨2:0⟩ control lines, and the BIIC detects the deasserted state. Note that the assertion of BI NO ARB L during a burst mode transaction is not checked.

MPE (W1C, DCLOC) Bit ⟨27⟩
: Master Parity Error. The BIIC sets MPE during a master transaction if it detects a parity error on the bus during a data cycle of a transaction that has an ACK confirmation on the CNF ⟨2:0⟩ lines.

ISE (W1C, DCLOC) Bit ⟨26⟩
: Interlock Sequence Error. The BIIC sets ISE if the KA820 module successfully completes a UWMCI transaction when the Unlock Write Pending (UWP) bit in the VAXBICSR is not set.

TDF (W1C, DCLOC) Bit ⟨25⟩
: Transmitter During Fault. The BIIC sets TDF if the BIIC is driving the BI D and I lines (only the I lines during the embedded ARB cycle) during a cycle with a parity error.

The BIIC also sets TDF during slave transactions if it detects a parity error on read data that it transmits. Software that reads this register should interpret a set TDF without a set MPE, CPE, or IPE as an indication that the BIIC has detected a parity error on its own transmitted read data.

TDF is not set for parity errors that occur during loopback transactions.

IVE (W1C, DCLOC) Bit ⟨24⟩

IDENT Vector Error. The BIIC sets IVE if it receives anything but an ACK confirmation from a master issuing IDENT after the KA820 module has sent out an interrupt vector. This can occur only when the KA820 module is used as an I/O controller and generates an interrupt.

CPE (W1C, DCLOC) Bit ⟨23⟩

Command Parity Error. The BIIC sets CPE when it detects a parity error during a command/address cycle. The error can occur during a VAXBI transaction or a loopback transaction.

SPE (W1C, DCLOC) Bit ⟨22⟩

Slave Parity Error. The BIIC sets SPE when it detects a parity error during a data cycle of a write transaction directed at the RXCD Register or any of the BIIC registers on the KA820 module.

RDS (W1C, DCLOC) Bit ⟨21⟩

Read Data Substitute. The BIIC sets RDS if it receives a Read Data Substitute or RESERVED status code during a read-type or IDENT (for vector status) transaction. The BIIC logic also requires a successful parity check for the data cycle that contains the RDS code, before it sets RDS. RDS gets set even if the transaction is aborted some time after the receipt of the RDS or RESERVED code.

RTO (W1C, DCLOC) Bit ⟨20⟩

RETRY Timeout. The BIIC sets RTO if the KA820 module receives 4096 consecutive RETRY responses from the slave for the same transaction.

STO (W1C, DCLOC) Bit ⟨19⟩

STALL Timeout. Not used on the KA820 module.

BTO (W1C, DCLOC) Bit ⟨18⟩

Bus Timeout. The BIIC sets BTO if it is unable to start any pending transaction (there may be several that are pending) before 4096 cycles have elapsed.

NEX (W1C, DCLOC) Bit ⟨17⟩

Nonexistent Address. The BIIC sets NEX when it receives a NO ACK response for a read-type or write-type command sent by the KA820 module. Note that this bit is set only if the master transmit check of the command/address cycle is successful.

The BIIC does not set NEX for NO ACK responses to other commands.

ICE (W1C, DCLOC) Bit ⟨16⟩ — Illegal Confirmation Error. The BIIC sets ICE when it receives a reserved or illegal CNF ⟨2:0⟩ code during a transaction. This bit can be set during either master or slave transactions. Note that NO ACK is not considered an illegal response for command confirmation.

D.3.2 Bus Error Register Parity Mode

UPEN (RO) Bit ⟨3⟩ — User Parity Enabled. UPEN indicates the parity mode (source of parity generation). When UPEN is 0, the BIIC is generating parity. When UPEN is 1, the port controller is generating parity. UPEN is normally 0.

D.3.3 Bus Error Register Soft Error Bits

IPE (W1C, DCLOC) Bit ⟨2⟩ — ID Parity Error. The BIIC sets IPE when it detects a parity error on the encoded ID asserted when the KA820 module is the VAXBI master, during an embedded arbitration cycle. This error condition also sets the TDF bit, except during loopback transactions.

CRD (W1C, DCLOC) Bit ⟨1⟩ — Corrected Read Data. The BIIC has received a corrected read data status code during a read transaction. The BIIC sets CRD even if the transaction is aborted after the BIIC receives the CRD status code.

NPE (W1C, DCLOC) Bit ⟨0⟩ — Null Bus Parity Error. The BIIC has detected odd parity on the VAXBI bus in the second cycle of a two-cycle sequence during which BI NO ARB L and BI BSY L were not asserted.

D.4 Error Interrupt Control Register, EINTRCSR

When the BIIC on the KA820 module detects an error on the VAXBI bus, it normally interrupts the processor. System software can use the Error Interrupt Control Register to control this function.

Node private space address: 2008 000C

Nodespace address: bb + 0C

System software should load the Error Interrupt Control Register to control interrupts from the BIIC following detection of a VAXBI error or the setting of the INTR FORCE bit in this register. The BIIC interrupts the KA820 CPU,

if the INTR FORCE bit or any of the Bus Error Register bits is set, and the error interrupt enable bits in the VAXBI Control and Status Register are set.

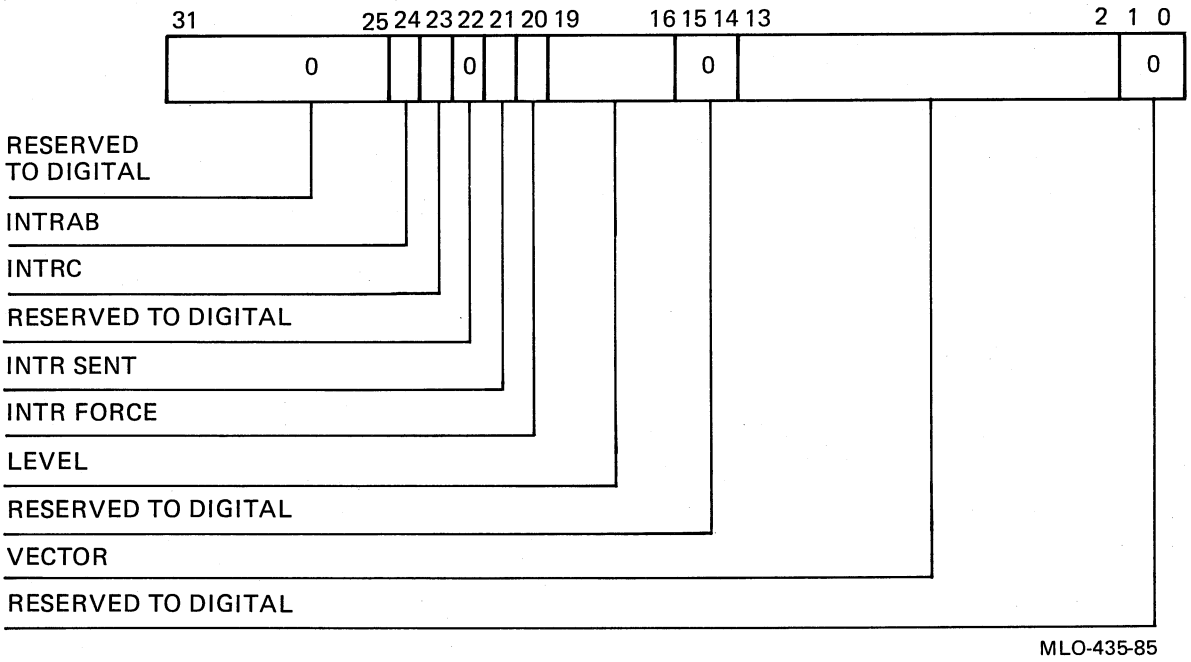


Figure D–4: Error Interrupt Control Register

INTRAB
(W1C, DCLOC, SC)
Bit ⟨24⟩

Interrupt Abort. The BIIC sets this status bit if an INTR command sent under the control of this register is aborted (a NO ACK or illegal confirmation code is received). System software should reset INTRAB after reading it. INTRAB has no effect on the ability of the BIIC to send or respond to further INTR or IDENT transactions.

INTRC
(W1C, DCLOC, SC)
Bit ⟨23⟩

Interrupt Complete. The BIIC sets INTRC when the BIIC has successfully transmitted the vector for an error interrupt or when an INTR command sent under the control of this register has been aborted. Removal of the error interrupt request automatically clears this bit. When set, INTRC inhibits the generation of new error interrupts by the BIIC.

INTR SENT
(W1C, DCLOC, SC)
Bit ⟨21⟩

Interrupt Sent. The BIIC sets INTR SENT when it has sent an INTR command and it expects an IDENT command to follow. The BIIC clears INTR SENT during an IDENT command following the detection of a level and master ID match. This allows the KA820 module to resend the interrupt re-

quest if it loses the interrupt arbitration or if it wins but the vector transmission fails. If the KA820 module deasserts the error interrupt request, the BIIC clears the INTR SENT bit.

INTR FORCE (R/W, DCLOC) Bit ⟨20⟩

Force Interrupt. System software can set INTR FORCE to force an error interrupt request when no error has been detected.

LEVEL ⟨7:4⟩ (R/W, DCLOC) Bits ⟨19:16⟩

Level ⟨7:4⟩. System software can set the Level field to define the level at which the BIIC transmits interrupt commands.

The Level field also helps determine whether this control register will respond to IDENT commands. If any level bits of a received IDENT command match the Level field in this register and there is a match in the destination mask, the BIIC will arbitrate for the IDENT. This lets you program the BIIC to respond to IDENT commands that match the level exactly, or to IDENT commands that match the level on a greater-than-or-equal-to basis.

The BIIC does not transmit an error interrupt request if none of the Level bits is set.

VECTOR (R/W, DCLOC) Bits ⟨13:2⟩

Vector. The BIIC uses the vector in this field during error interrupt sequences. It transmits the vector when this node wins an IDENT arbitration cycle on an IDENT transaction that matches the conditions in the Error Interrupt Control Register. The KA820 module uses the Vector field only when it functions as an I/O controller.

D.5 BCI Control and Status Register, BCICSR

Processor initialization microcode loads the BCI Control and Status Register, and software should leave it unchanged.

Node private space address: 2008 0028

Nodespace address: bb + 28

BURSTEN (R/W, DCLOC, NA) Bit ⟨17⟩

Burst Enable. When BURSTEN is set, the BIIC asserts BI NO ARB L continuously after the next successful arbitration, until the BURSTEN bit is reset or the BCI MAB L signal is asserted. The assertion of BCI MAB L does not reset the BURSTEN bit. It merely clears the burst mode state in the BIIC that is holding BI NO ARB L. Unless a subsequent transaction clears this bit, the next

successful arbitration causes the BIIC to again assert BI NO ARB L continuously. Note that loopback transactions must not use the burst mode.

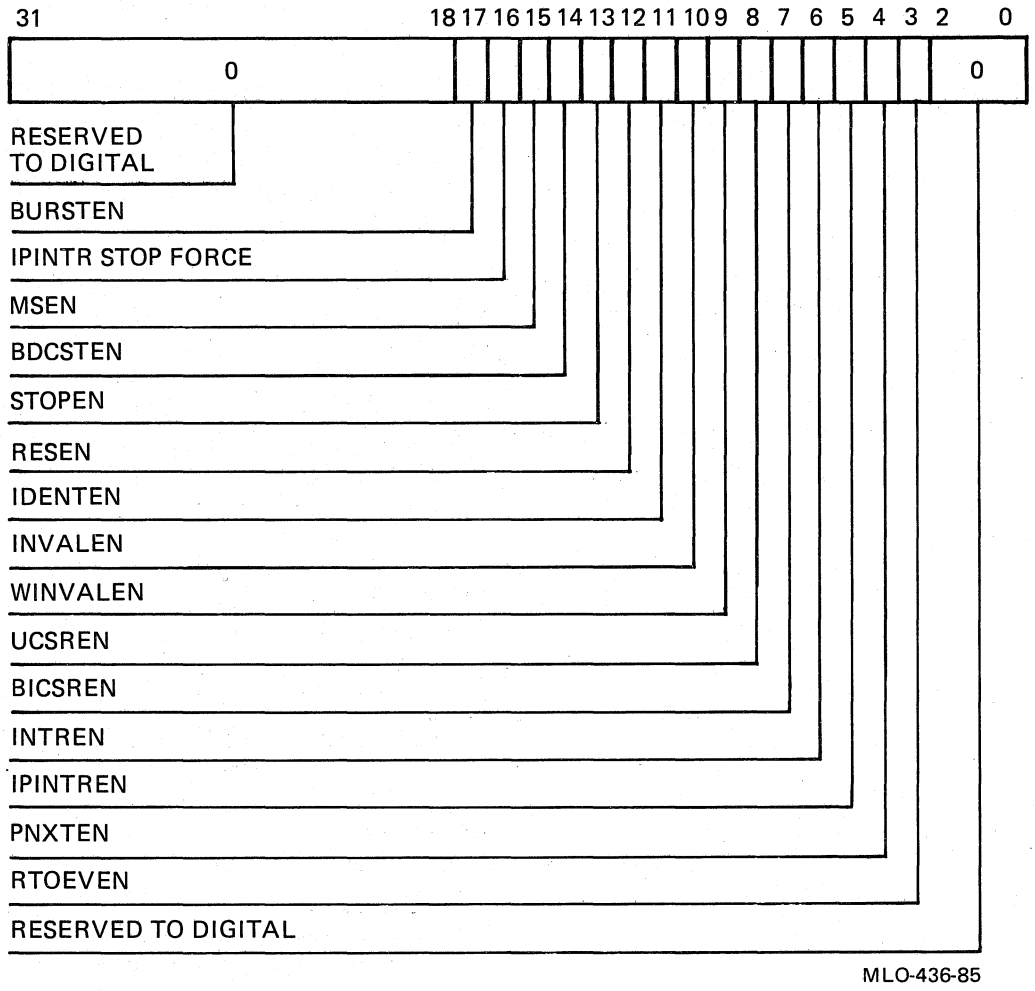


Figure D–5: BCI Control and Status Register

IPINTR/STOP FORCE (R/W, DCLOC, SC, NA) Bit ⟨16⟩

IP Interrupt/Stop Force. When IPINTR/STOP FORCE is set, it forces the BIIC to arbitrate for the bus and transmit an IPINTR command or STOP command, using the IPINTR Destination Register for the destination field. The BIIC clears the IPINTR/STOP FORCE bit when it transmits the IPINTR command, regardless of whether the transaction is completed successfully. Software can monitor the EV ⟨4:0⟩ lines to determine whether the transaction is completed normally.

MSEN
(R/W, DCLOC, DS)
Bit ⟨15⟩

Multicast Space Enable. When MSEN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of a read-type or write-type command directed at broadcast space. This bit is normally left cleared, so the KA820 module does not respond to multicast space commands.

BDCSTEN
(R/W, DCLOC, DS)
Bit ⟨14⟩

BDCST Enable. When BDCSTEN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of a BDCST command. This bit is normally left cleared, so the KA820 module does not respond to BDCST commands.

STOPEN
(R/W, DCLOC, DS)
Bit ⟨13⟩

STOP Enable. The KA820 module does not respond to the STOP command; this bit should remain cleared.

RESEN
(R/W, DCLOC, SC)
Bit ⟨12⟩

RESERVED Enable. When RESEN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of a RESERVED command code. RESEN is normally clear.

IDENTEN
(R/W, DCLOC, SC)
Bit ⟨11⟩

IDENT Enable. When IDENTEN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of an IDENT command. This bit affects only the output of SEL and the IDENT SC code. Therefore, the BIIC always participates in IDENT transactions that select this node, even if IDENTEN is cleared.

INVALEN
(R/W, DCLOC, DS)
Bit ⟨10⟩

INVAL Enable. When INVALEN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of an INVAL command. Processor initialization microcode sets INVALEN to enable the port controller to forward VAXBI invalidate transactions to the cache tag array on the M chip.

WINVALEN
(R/W, DCLOC, SC)
Bit ⟨9⟩

Write Invalidate Enable. When WINVALEN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of a write-type command for which the address is not in I/O space. Processor initialization microcode sets WINVALEN to enable the BIIC to forward VAXBI write addresses to the port controller, so it, in turn, can send invalidate requests to the cache tag array in the M chip.

UCSREN
(R/W, DCLOC, DS)
Bit ⟨8⟩

User CSR Space Enable. When set, this bit causes the BIIC to assert BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of a read-type

or write-type command directed at the RXCD Register on the KA820 module. Processor initialization microcode sets UCSREN to enable VAXBI access to the RXCD Register.

BICSREN
(R/W, DCLOC, SC)
Bit ⟨7⟩

BIIC CSR Space Enable. When BICSREN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of a read-type or write-type command to its BIIC CSR space (the first 256 bytes of this node's node space). Processor initialization microcode sets BICSREN to enable VAXBI access to the BIIC CSR space on the KA820 module.

INTREN
(R/W, DCLOC, DS)
Bit ⟨6⟩

INTR Enable. When INTREN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of an INTR command. Processor initialization microcode sets INTREN.

IPINTREN
(R/W, DCLOC, SC)
Bit ⟨5⟩

IPINTR Enable. When IPINTREN is set, the BIIC asserts BCI SEL L and the appropriate BCI SC ⟨2:0⟩ code following receipt of an IPINTR command that matches the IPINTR Mask Register.

However, IPINTREN enables only the IPINTR SEL/SC code. The state of IPINTREN does not determine whether the KA820 module receives IPINTR commands. Software should clear the IPINTR Mask Register to disable receipt of IPINTR commands. Processor initialization microcode sets IPINTREN.

PNXTEN
(R/W, DCLOC, NA)
Bit ⟨4⟩

Pipeline NXT Enable. When PNXTEN is set, the BIIC asserts BCI NXT L for an extra cycle (one more than the number of longwords transferred) during write-type transactions. This extra BCI NXT L cycle occurs after the last NXT L cycle for write data, and is used to increment the write-silo address counter to the address of the command/address data for the next transaction. Processor initialization microcode sets PNXTEN when it sets the pipeline enable bit in the port controller CSR, to enable the pipeline mode for memory write transactions, increasing the efficiency of the processor.

NOTE

If ENBL PIPE (PCntl CSR bit ⟨13⟩) is set, PNXTEN must be set as well. If ENBL PIPE is clear, PNXTEN must be clear.

RTOEVEN
(R/W, DCLOC, NA)
Bit ⟨3⟩

When RTOEVEN is set, the BIIC asserts the RETRY Timeout (RTO) Event code instead of the RETRY CNF Received for Master Port Command (RCR) Event code following the occurrence of a RETRY timeout. If RTOEVEN is cleared, the BIIC will not assert the RTO EV code in place of the RCR EV code following a retry timeout. However, the RTO bit in the BER will be set, and the BIIC will generate an error interrupt if error interrupts are enabled. Processor initialization microcode sets RTOEVEN.

D.6 Receive Console Data Register, RXCD

The KA820 module receives data from other processors in the RXCD Register, one byte at a time. This register is implemented in the port controller, not in the BIIC.

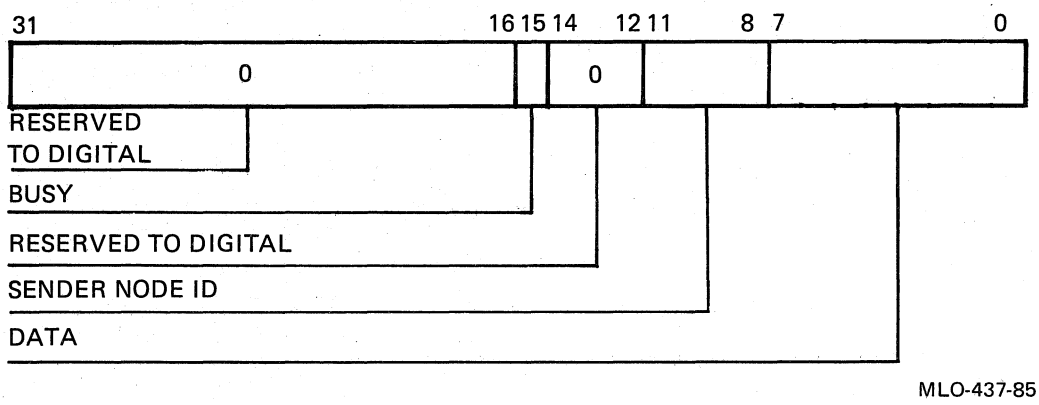


Figure D–6: RXCD Register

Node private space address: 2008 0200

Nodespace address: bb + 200

BUSY
Bit ⟨15⟩

Busy bit. The receiving (local) KA820 module changes BUSY from 1 to 0 after reading the RXCD Register. The sending processor changes BUSY from 0 to 1 when it writes data in the RXCD Register.

SENDER NODE ID
Bit ⟨11:8⟩

Identifies the sender's node number.

DATA
Bits ⟨7:0⟩

Data received. In communications between the consoles of two processors, the data is an ASCII character.

D.6.1 MFPR Instruction for the RXCD Register

Software can read the RXCD Register on the local KA820 module with an MFPR instruction.

If the BUSY bit is cleared, no character has been received. The MFPR instruction signals this condition by setting the V bit in the PSL.

If the BUSY bit is set, a character has been received. The MFPR instruction reads the Data and Sender Node ID fields and clears the BUSY bit to indicate that the RXCD Register is free to receive another character. The MFPR instruction signals this condition by clearing the V bit in the PSL.

D.6.2 MTPR Instruction for the RXCD Register

Software can write the RXCD Register on a remote processor on the VAXBI bus with an MTPR instruction.

If the BUSY bit is set, no data is transferred. The MTPR instruction signals this condition by setting the V bit in the PSL.

If the Busy bit is cleared, the MTPR writes the Data and Sender Node ID fields in the remote RXCD Register and sets the BUSY bit to indicate that new data is in the register. The MTPR instruction signals this condition by clearing the V bit in the PSL.

Appendix E
Port Controller Control and Status Register

The port controller CSR performs three functions that are central to operation of the KA820 module:

1. Provides a read-only interface between the module and the control panel that lets software and microcode examine switch positions and off-module signals.
2. Enables software control of some module functions.
3. Stores module and VAXBI bus status information.

This register is accessible in the node private address space at address 2008 8000. No other node on the VAXBI bus can read or write it.

When a VAXBI error occurs, machine-check microcode copies bits ⟨22:16, 14⟩ to the MTEMPC Register in the M chip and to the status word on the stack at SP + 18, where they are available to exception handler software (see Section 5.2 and Table 5–3 in Chapter 5).

RSTRT HLT
Bit ⟨31⟩

(RO)
Auto Start/Halt Power-up Option Bit

1 = Halt
0 = Auto Start
Related signal name: PNL RSTRT HLT H
Module I/O pin: D51

RSTRT HLT shows the position of the lower key switch on the control panel. Microcode checks the status of this bit to determine whether to restart (warm or cold) or go to console mode at power-up and following an error halt condition.

RSTRT HLT is 1 on all KA820 attached processors, since the signal from the control panel switch runs only to the primary processor.

CNSL LOG
Bit ⟨30⟩

(RO)
Physical/Logical Console Selection Bit

1 = Logical
0 = Physical
Related signal name: PNL CNSL LOG H
Module I/O pin: D52

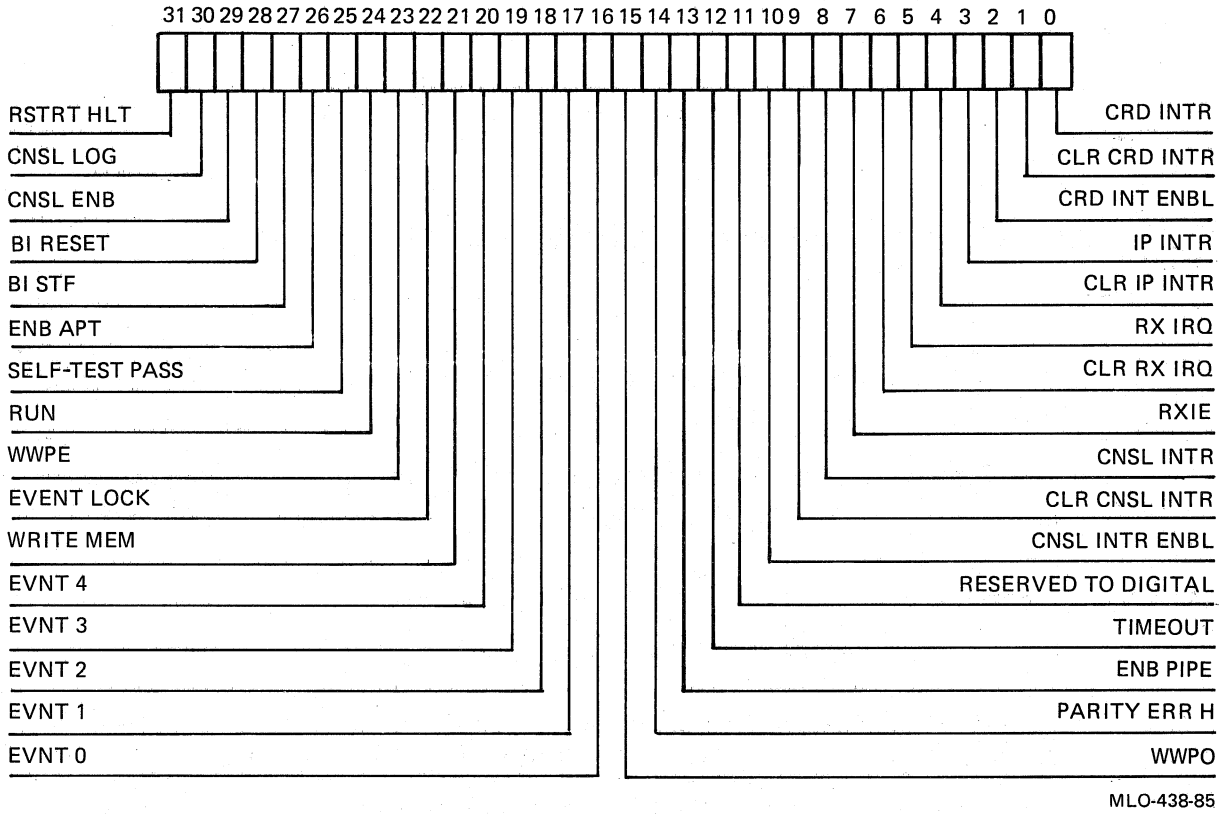


Figure E–1: Port Controller Control and Status Register

Microcode reads this bit to determine the console source. The physical console is serial-line unit 0, which is normally attached to a terminal. The logical console is another processor on the VAXBI bus.

CNSL ENB
Bit ⟨29⟩

(RO)
Console Secure/Enabled Selection Bit

1 = Console Enabled
0 = Console Secure
Related signal name: PNL CNSL ENB H
Module I/O pin: D53

CNSL ENB shows the position of the upper key switch on the control panel. Microcode checks this bit when the console sends CTRL/P, to determine whether console functions are enabled.

CNSL ENB is 1 on all KA820 attached processors, since the signal from the upper key switch on the control panel runs only to the primary processor.

BI RESET
Bit ⟨28⟩

(R/W)
System Reset Control Bit

Related signal name: BI RESET L
Module I/O pin: D54

Software can completely reset the VAX 8200 system, including VAXBI memory, by setting BI RESET. Setting this bit simulates a power-up sequence by cycling the BI AC LO L and BI DC LO L signals. Writing 0 clears the bit, but this has no useful effect, because whenever it is set, the resulting BI DC LO L signal clears it.

BI STF
Bit ⟨27⟩

(RO)
Self-Test Fast/Slow Selection Bit

1 = Slow Self-Test
0 = Fast Self-Test
Related signal name: BI STF L
Module I/O pin: D55

Microcode reads this bit to determine whether to run the slow self-test or the fast self-test.

ENB APT
Bit ⟨26⟩

(RO)
APT Connection Status Bit

1 = APT line not connected
0 = APT line connected
Related signal name: EXT ENB APT L
Module I/O pin: C45

Software can read this bit to determine whether an APT line is connected to serial-line unit 0. APT (Automated Product Test) is a testing system used by DIGITAL manufacturing.

SELF-TEST PASS
Bit ⟨25⟩

(Read/Write by microcode)
Self-Test Status Bit

1 = Successful self-test
0 = Failed self-test

Self-test microcode writes 1 to this bit at the end of a successful self-test, either at power-up or from the console T (Test) command. The BI DC LO L signal clears the Self-Test Status bit at power-up.

When the Self-Test Status bit is cleared, the KA820 module asserts the BI BAD L signal on module I/O pin D51 and it turns off the two yellow self-test-passed LEDs on the module.

When the Self-Test Status bit is set, the KA820 module deasserts BI BAD L and turns on the two yellow self-test-passed LEDs on the module.

RUN
Bit ⟨24⟩

(R/W)
Program Mode Run Bit

Related signal name: PNL RUN LED L
Module I/O pin: D49

Microcode sets this bit to indicate that it is in the program I/O mode; microcode clears the bit when changing to console mode.

The console microcode toggles this bit each time it recognizes a ⟨BREAK⟩ character on serial-line unit 0. This is an aid in troubleshooting a dead console terminal. When RUN L toggles, the Run light on the control panel flashes on and off to indicate that the CPU recognizes the characters typed on the console terminal. If the control-panel Run light flashes and the terminal is dead, it indicates that the problem is in the output path to the console terminal or in the terminal itself.

WWPE
Bit ⟨23⟩

(R/W)
Write Wrong Parity, Even Bytes.

1 = Force wrong parity generation and disable parity checking of even data bytes on the DAL bus
0 = Normal parity generation and parity checking of the even bytes on the DAL bus

Self-test microcode and diagnostic software can set WWPE to force wrong parity on the even bytes of the DAL bus to, in turn, force wrong parity on either the cache or BTB parity RAMs on write operations. This function also prevents the port controller from detecting a cache or BTB parity error on even bytes.

Microcode and diagnostic software can clear WWPE to resume normal parity operation and checking of even bytes of the cache and BTB parity RAMs. Note that bit ⟨15⟩ performs an equivalent function for odd bytes.

EVENT LOCK
Bit ⟨22⟩

(Read/Write 1 to Clear)
Event Code Error Bit

1 = Event code is locked
0 = Event code is unlocked

The port controller sets this bit when it detects an error condition on the Event lines from the BIIC. Once Event Lock is set, bits ⟨21:16⟩ are latched and no longer reflect the state of the BIIC Event lines.

Software can clear the bit and unlock the latched bits ⟨21:16⟩ by writing 1 to Event Lock. Writing 0 to Event Lock has no effect.

WRITE MEMORY
Bit ⟨21⟩

(RO)
Memory Write Transaction Status Bit

1 = Error in Write Operation
0 = Error in Nonwrite Operation

The port controller sets this bit during memory write transactions, two cycles after the BIIC asserts RAK (Request Acknowledge). Write Memory remains set as long as memory write transactions continue. Any other type of VAXBI transaction clears the bit. If the BIIC signals the occurrence of a VAXBI error condition, the port controller latches Write Memory with the event code bits when it sets Event Lock.

Write Memory lets software determine when an error is associated with a pipelined VAXBI memory write transaction, since errors may go undetected until the port controller begins a subsequent transaction.

EVENT4 — 0
Bits ⟨20:16⟩

(RO)
VAXBI Transaction Event Code Bits.

The port controller checks these bits for error codes. If it finds an error code, it latches bits ⟨20:16⟩ and Write Memory, sets Event Lock, and invokes machine check microcode.

Event codes (hex) latched by the port controller:

3	Bus Timeout (BTO)
18	Read Data Substitute or RESERVED Status Code Received (RDSR)
19	Illegal CNF Received for Master Port Command (ICRMC) Command
1A	NO ACK CNF Received for Master Port Command (NCRMC)
1C	Illegal CNF Received by Master Port for Data Cycle (ICRMD)
1D	Retry Timeout (RTO)
1E	Bad Parity Received During Master Port Transaction (BPM)
1F	Master Transmit Check Error (MTCE)

After the machine-check exception handler deals with the error, it should write 1 to PCntl CSR bit ⟨22⟩ to clear Event Lock and release the latch on bits ⟨20:16⟩ and Write Memory. See Table 5–4 in Chapter 5 for a full list of the event codes.

WWPO
Bit ⟨15⟩

(R/W)
Write Wrong Parity, Odd Bytes.

1 = Force wrong parity generation and disable parity checking of the odd bytes on the DAL bus
0 = Normal parity generation and parity checking of the odd bytes on the DAL bus

Self-test microcode and diagnostic software can set WWPO to force wrong parity on the odd bytes of the DAL bus to, in turn, force wrong parity on either the cache or BTB parity RAMs on write operations. This function also prevents the port controller from detecting a cache or BTB parity error on odd bytes.

Microcode and diagnostic software can clear WWPO to resume normal parity operation and checking of odd bytes on the cache and BTB parity RAMs. Note that PCntl CSR bit ⟨23⟩ performs an equivalent function for even bytes.

PARITY ERROR
Bit ⟨14⟩

(Read/Write 1 to clear)
Parity Error Status Bit

1 = Parity error detected by the port controller
0 = No parity error detected

Cleared by BI DC LO L

The port controller sets this bit when it detects a parity error on data read from the cache RAMs or BTB RAMs. The port controller asserts the PCNTL ERROR L signal at the same time to notify the M chip of the error.

ENBL PIPE
Bit ⟨13⟩

(R/W)
Enable VAXBI Pipeline Mode Control Bit

1 = Enable pipeline mode
0 = Disable pipeline mode

Cleared by BI DC LO L

Pipeline mode lets the port controller post the next VAXBI transaction to the BIIC without waiting until the current transaction is completed and confirmation is received.

However, to aid error recovery, the port controller disables pipelining during write transactions to I/O space, regardless of the state of this bit. This lets microcode make sure that the transaction is completed, or an error is detected, before it starts the next transaction.

NOTE

If ENBL PIPE is set, PNXTEN (BCI Control Register bit ⟨4⟩) must be set as well. If ENBL PIPE is clear, PNXTEN must be clear.

TIMEOUT
Bit ⟨12⟩

(Read/Write 1 to Clear)
Port Controller Timeout Bit

1 = Port controller timeout error
0 = No port controller timeout error

Cleared by BI DC LO L

The port controller starts a timer when it receives a command from the CPU to perform a VAXBI transaction. If the transaction has not been completed 12.8 milliseconds later, the port controller sets this bit and asserts the PCNTL ERROR L signal to notify the M chip of the error.

Reserved to DIGITAL
Bit ⟨11⟩

CNSL INTR ENBL
Bit ⟨10⟩

(R/W)
RXCD Logical Console Interrupt Enable Control Bit

1 = Enable interrupts from the RXCD Register
0 = Disable interrupts from the RXCD Register

Cleared by BI DC LO L

Processor initialization microcode sets this bit following self-test to enable interrupts from the RXCD Register. With this bit set, an interrupt occurs when the BUSY bit in the RXCD Register is set.

Software can disable RXCD Register interrupts by writing 0 to this bit.

CLEAR CNSL INTR
Bit ⟨9⟩

(Write Only)
Clear RXCD Console Interrupt Bit

Interrupt handling microcode writes 1 to this bit to clear PCntl CSR bit ⟨8⟩, CNSL INTR.

CNSL INTR
Bit ⟨8⟩

(RO)
RXCD Console Interrupt Bit

1 = RXCD interrupt pending
0 = No RXCD interrupt pending

The port controller sets this bit when the logical console is enabled and the RXCD BUSY bit changes from 0 to 1, indicating that new data is in the register.

RXIE
Bit ⟨7⟩

(R/W)
RCX50 Interrupt Enable Bit

1 = RCX50 interrupt enabled
0 = RCX50 interrupt disabled

Cleared by BI DC LO L

Software can enable or disable interrupts from the RCX50 controller by setting or clearing this bit. You must set RXIE before performing RX50 I/O functions.

CLR RX IRQ
Bit ⟨6⟩

(Write Only)
Clear RCX50 Interrupt Request Bit

Interrupt handling microcode writes 1 to this bit to clear the RCX50 interrupt request bit, PCntl CSR bit ⟨5⟩, after reading it.

RX IRQ
Bit ⟨5⟩

(RO)
RCX50 Interrupt Request Bit

1 = RCX50 interrupt request pending
0 = NO RCX50 interrupt request pending

Cleared by BI DC LO L
Cleared by writing 1 to PCntl CSR bit ⟨6⟩

The port controller sets RX IRQ whenever it receives RX INTRA or RX INTRB pulses from the RCX50 controller. RX INTRA signals the completion of each operation, and RX INTRB signals a change in the RX50 drive status.

The port controller ORs RX IRQ with IP INTR and BI INTR4 to produce BI INTR4 at IPL14. Therefore interrupt handling microcode reads RX IRQ, when it responds to a BI INTR4 request, to determine if the request is coming from the RCX50 controller. Microcode clears RX IRQ after reading it, by writing 1 to PCntl CSR bit ⟨6⟩.

CLEAR IP INTR
Bit ⟨4⟩

(Write Only)
Clear Interprocessor Interrupt Request Bit

Interrupt handling microcode writes 1 to this bit to clear the IP INTR bit, PCntl CSR bit ⟨3⟩.

IP INTR
Bit ⟨3⟩

(RO)
Interprocessor Interrupt Request Bit

1 = Interprocessor interrupt request pending
0 = No interprocessor interrupt request pending

Cleared by BI DC LO L
Cleared by writing 1 to PCntl CSR bit ⟨4⟩

The port controller sets this bit when it receives an IP INTR request from the VAXBI bus. The port controller ORs IP INTR with RX IRQ and BI INTR4 at IPL 14. Interrupt handling microcode clears IP INTR as it handles interrupt requests, by writing 1 to PCntl CSR bit ⟨4⟩.

CRD INTR ENBL
Bit ⟨2⟩

(R/W)
Corrected Read Data Interrupt Enable Control Bit

1 = CRD interrupt enabled
0 = CRD interrupt disabled

Cleared by BI DC LO L

The BIIC on the KA820 module generates a corrected read data interrupt when a single-bit error occurs in the VAXBI memory array and the memory has corrected the data. Error logging software uses this information to keep track of soft errors.

CLEAR CRD INTR
Bit ⟨1⟩

(Write Only)
Clear CRD Interrupt Bit

Interrupt handling microcode writes 1 to this bit to clear the CRD INTR bit, PCntl CSR bit ⟨0⟩.

CRD INTR
Bit ⟨0⟩

(RO)
CRD Interrupt Bit

Cleared by writing 1 to PCntl CSR bit ⟨1⟩.

The port controller sets CRD INTR when the BIIC indicates that it has received corrected read data and CRD INTR ENBL (PCntl CSR bit ⟨2⟩) is set. Interrupt handling microcode clears the bit by writing 1 to PCntl CSR bit ⟨1⟩.

Appendix F
Internal Processor Registers on the KA820 Module

You can use the console E/I command or macrocode MTPR and MFPR instructions to access these registers.

The following table lists the IPRs and their addresses.

IPR Address (hex)	Mnemonic	Name
0	KSP	Kernel Stack Pointer
1	ESP	Executive Stack Pointer
2	SSP	Supervisor Stack Pointer
3	USP	User Stack Pointer
4	ISP	Interrupt Stack Pointer
8	P0BR	P0 Base Register
9	P0LR	P0 Length Register
A	P1BR	P1 Base Register
B	P1LR	P1 Length Register
C	SBR	System Base Register
D	SLR	System Length Register
10	PCBB	Process Control Block Base
11	SCBB	System Control Block Base
12	IPLR	Interrupt Priority Level Register
13	ASTLVL	Asynchronous System Trap Level
14	SIRR	Software Interrupt Request Register
15	SISR	Software Interrupt Summary Register
16	IPIR	Interprocessor Interrupt Register
18	ICCS	Interval Clock Control Register
19	NICR	Next Interval Count Register
1A	ICR	Interval Count Register
1B	TODR	Time of Day Counter Register
20	RXCS	Console Transmit Buffer
21	RXDB	Console Receive Data Buffer
22	TXCS	Console Transmit CSR
23	TXDB	Console Transmit Data Buffer
24	TBDR	Translation Buffer Disable Register
25	CADR	Cache Disable Register
26	MCESR	Machine Check Error Summary Register
28	ACCS	Floating-Point Accelerator CSR
2C	WCSA	Writeable Control Store Address Register
2D	WCSD	Writeable Control Store Data Register
2E	WCSL	Writeable Control Store Load Register

(Continued on next page)

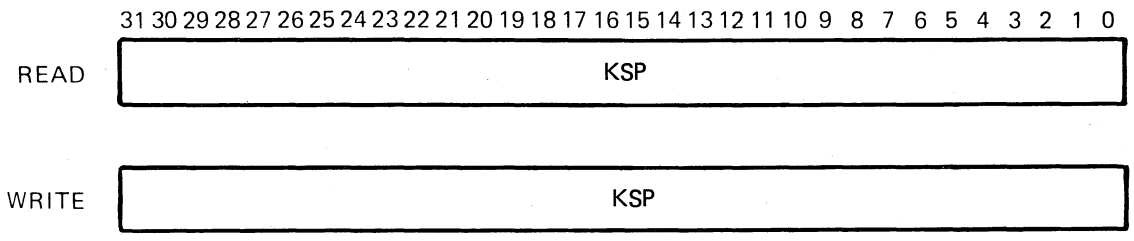
IPR Address (hex)	Mnemonic	Name
38	MAPEN	Memory Management Enable Register
39	TBIA	Translation Buffer Invalidate All Register
3A	TBIS	Translation Buffer Invalidate Single Register
3D	PMR	Performance Monitor Enable Register
3E	SID	System Identification Register
3F	TBCHK	Translation Buffer Check Register
50	RXCS1	Serial-Line Unit 1 Receive CSR
51	RXDB1	Serial-Line Unit 1 Receive Data Buffer
52	TXCS1	Serial-Line Unit 1 Transmit CSR
53	TXDB1	Serial-Line Unit 1 Transmit Data Buffer
54	RXCS2	Serial-Line Unit 2 Receive CSR
55	RXDB2	Serial-Line Unit 2 Receive Data Buffer
56	TXCS2	Serial-Line Unit 2 Transmit CSR
57	TXDB2	Serial-Line Unit 2 Transmit Data Buffer
58	RXCS3	Serial-Line Unit 3 Receive CSR
59	RXDB3	Serial-Line Unit 3 Receive Data Buffer
5A	TXCS3	Serial-Line Unit 3 Transmit CSR
5B	TXDB3	Serial-Line Unit 3 Transmit Data Buffer
5C	RXCD	Receive Console Data Register
5D	CACHEX	Cache Invalidate Register
5E	BINID	VAXBI Node ID Register
5F	BISTOP	VAXBI Stop Register

As you examine the formats of these registers, keep the following three conventions in mind:

0 on read	=	Read as zero.
0 on write	=	Software must supply zeros. Microcode does not force zeros or check for errors. These bits are read back as written.
x	=	Ignored on write.

Hex Dec. Name Format

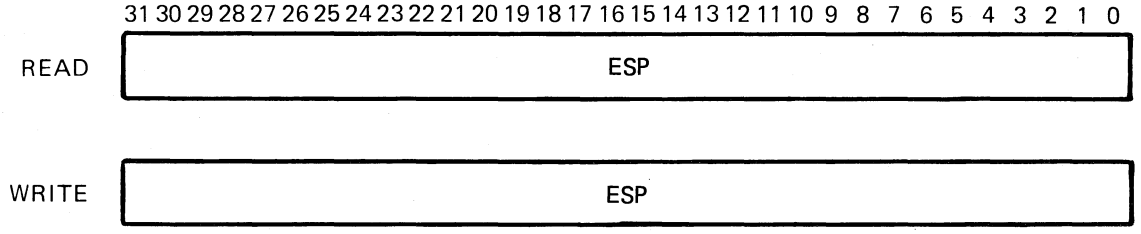
00 0. KSP Kernel Stack Pointer



MLO-439-85

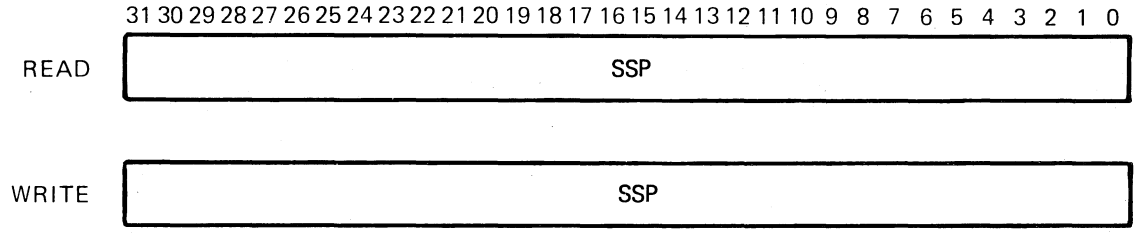
Hex	Dec.	Name	Format

01 1. ESP Executive Stack Pointer



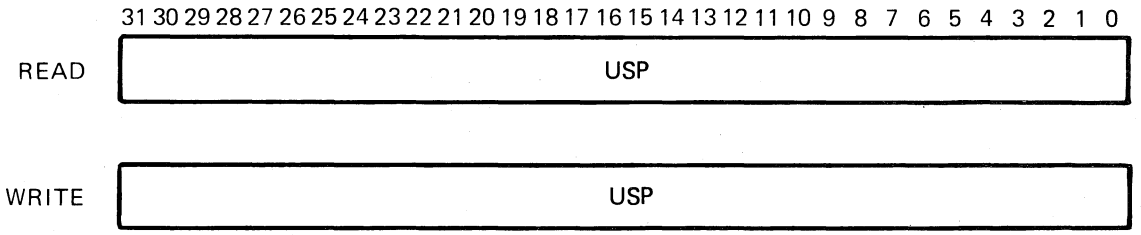
MLO-440-85

02 2. SSP Supervisor Stack Pointer



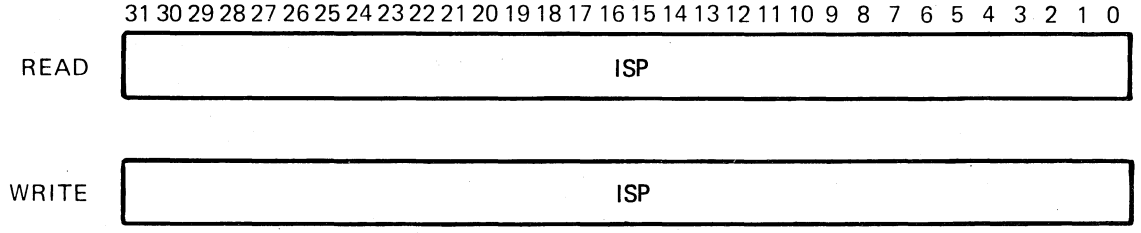
MLO-441-85

03 3. USP User Stack Pointer



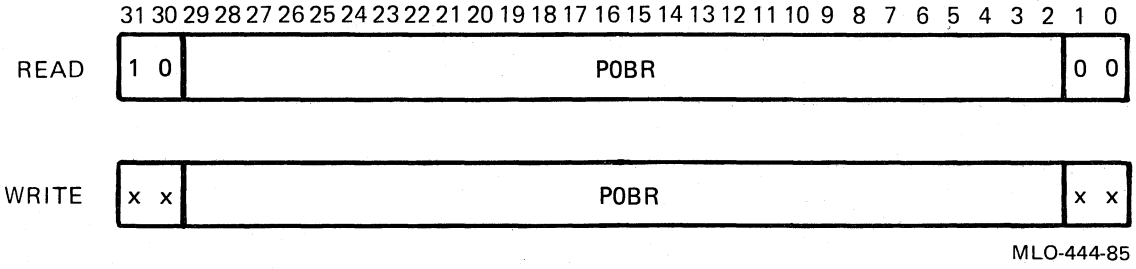
MLO-442-85

04 4. ISP Interrupt Stack Pointer



MLO-443-85

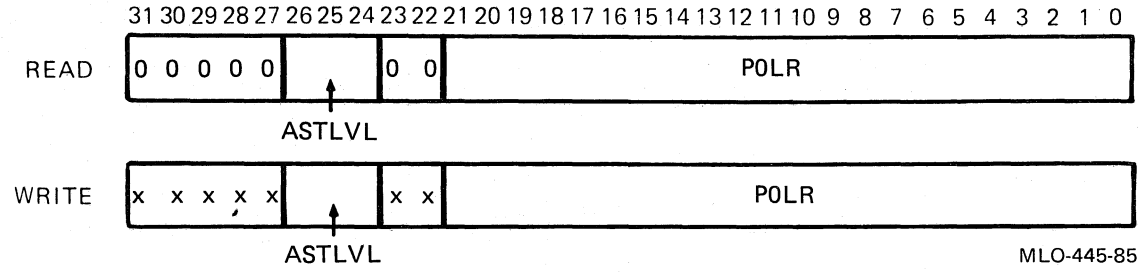
Hex	Dec.	Name	Format
08	8.	P0BR	P0 Base Register



MLO-444-85

P0BR contains the virtual address of the beginning of the P0 page table.

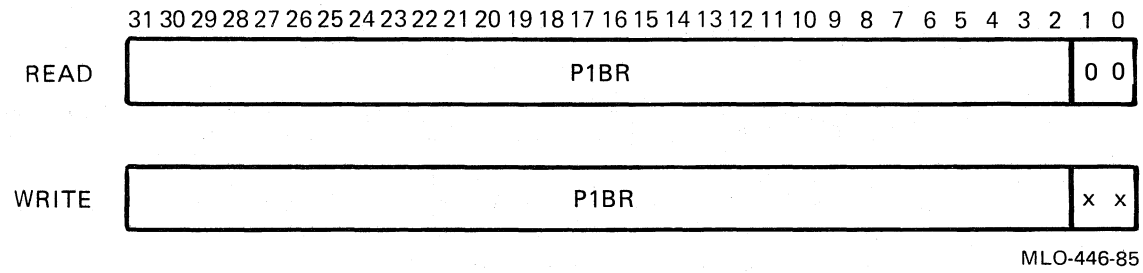
Hex	Dec.	Name	Format
09	9.	P0LR	P0 Length Register



MLO-445-85

P0LR tells the size of the P0 page table in longwords. ASTLVL stands for asynchronous system trap level.

Hex	Dec.	Name	Format
0A	10.	P1BR	P1 Base Register

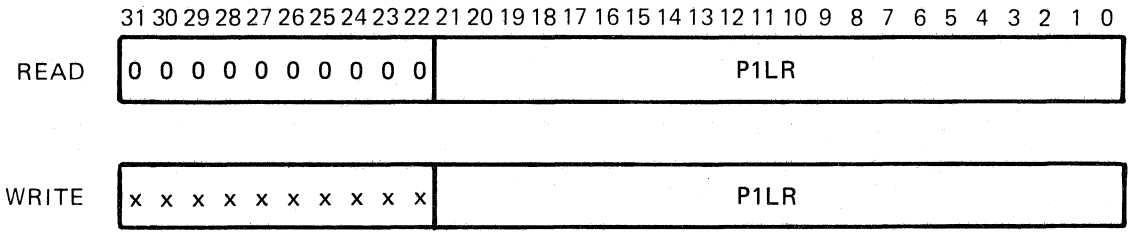


MLO-446-85

Hex	Dec.	Name	Format

P1BR contains the virtual address of the page table entry for the next unused page of P1 space. Initially this address is 4000 0000.

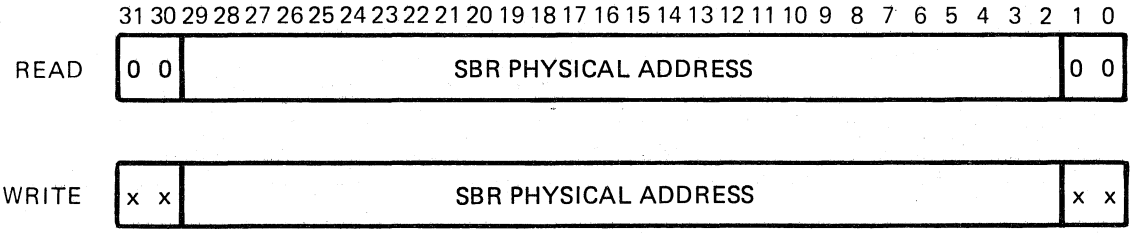
0B 11. P1LR P1 Length Register



MLO-447-85

P1LR tells the number of nonexistent page table entries in P1 space.

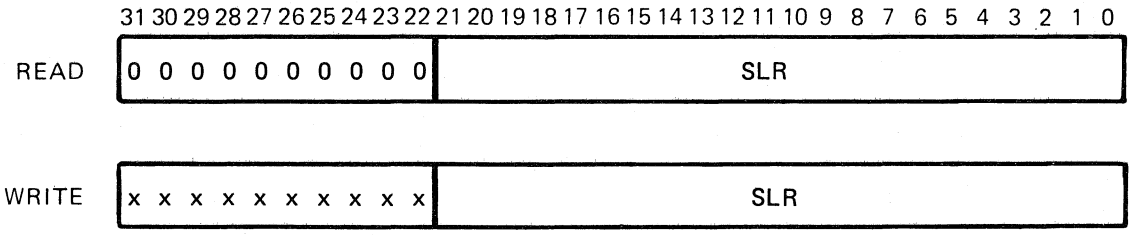
0C 12. SBR System Base Register



MLO-448-85

SBR contains the physical base address of the system page table.

0D 13. SLR System Length Register

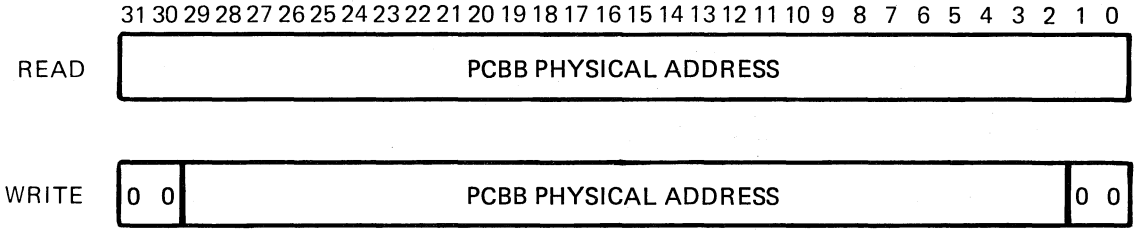


MLO-449-85

Hex	Dec.	Name	Format

SLR tells the size of the system page table in longwords.

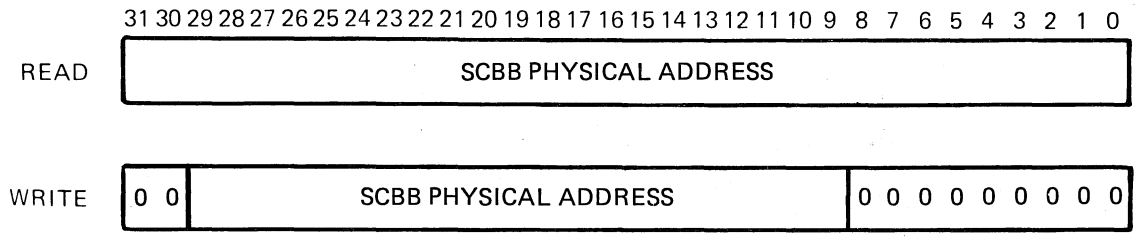
10 16. PCBB Process Control Block Base



MLO-450-85

PCBB contains the physical address of the process control block.

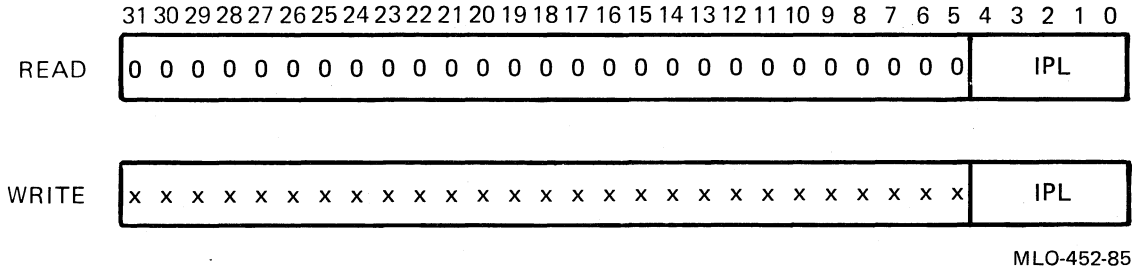
11 17. SCBB System Control Block Base



MLO-451-85

SCBB contains the physical address of the system control block (SCB). The SCB is a table of vectors used to service exceptions and interrupts (see Chapter 5).

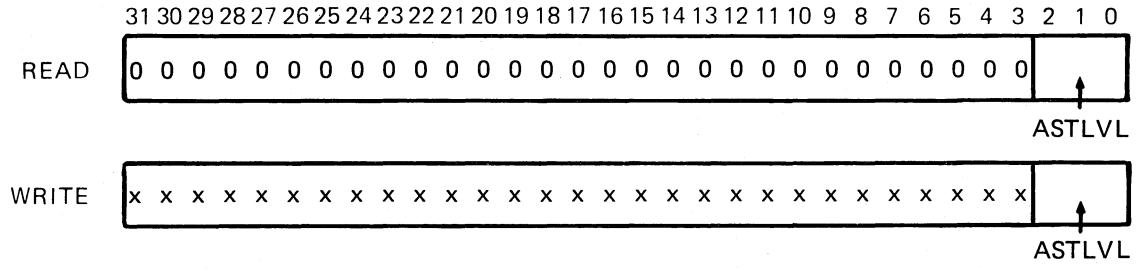
Hex	Dec.	Name	Format
12	18.	IPLR	Interrupt Priority Level Register



MLO-452-85

Writing to IPLR with the MTPR instruction loads the processor priority field of the PSL. PSL bits ⟨20:16⟩ are loaded from IPLR bits ⟨4:0⟩. Reading IPLR with the MFPR instruction reads the processor priority field of the PSL.

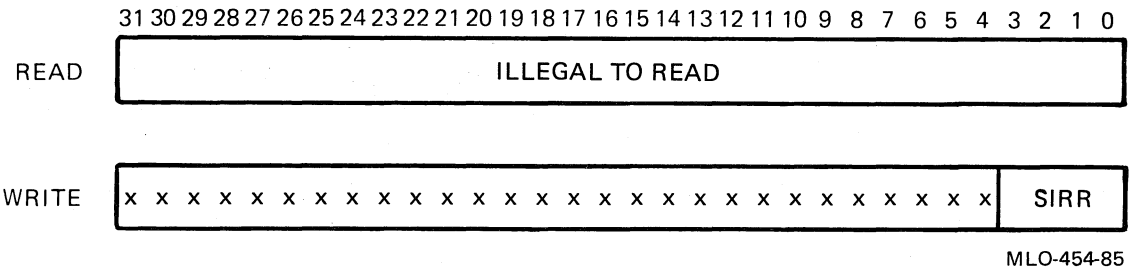
13	19.	ASTLVL	Asynchronous System Trap Level



MLO-453-85

ASTLVL tells the most privileged access mode number for which an asynchronous system trap is pending.

14	20.	SIRR	Software Interrupt Request Register

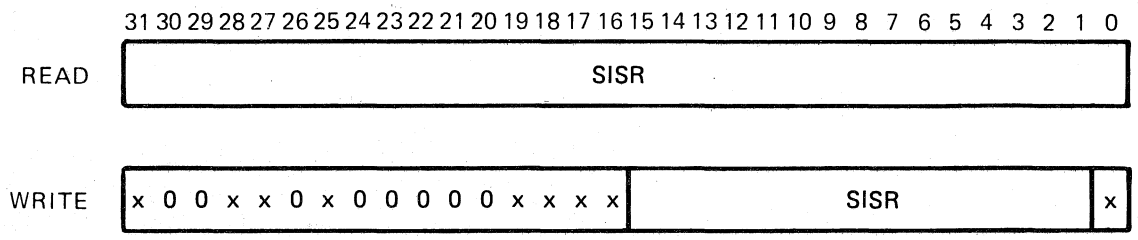


MLO-454-85

Hex	Dec.	Name	Format

Software can execute an MTPR instruction to SIRR to request an interrupt at the level specified in bits ⟨3:0⟩.

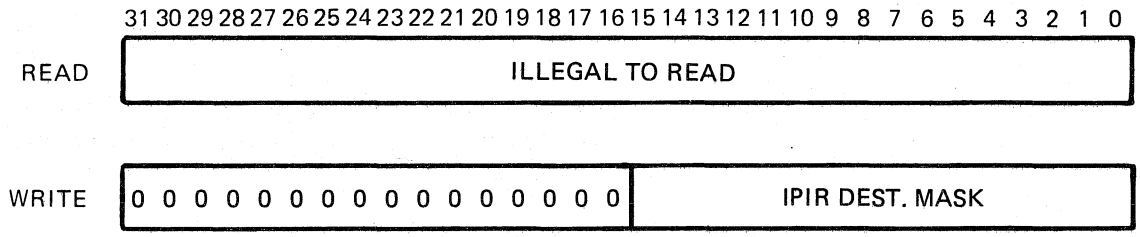
15 21. SISR Software Interrupt Summary Register



MLO-455-85

When software reads SISR, bits⟨15:1⟩ contain 1s in bit positions corresponding to levels at which software interrupts are pending. In addition software can post interrupts at several levels at once by writing to SISR, setting any of the bits in the field ⟨15:1⟩. Software should not set bits ⟨31:16⟩.

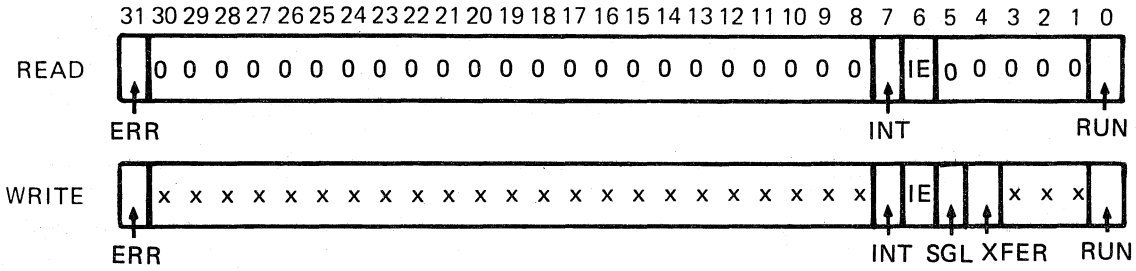
16 22. IPIR Interprocessor Interrupt Request Register



MLO-456-85

To send an interprocessor interrupt (IPINTR), software should write this register with the destination ID mask. Bit ⟨0⟩ of the mask corresponds to VAXBI node 0, and bit ⟨15⟩ corresponds to VAXBI node 15, and so on.

Hex	Dec.	Name	Format
18	24.	ICCS	Interval Clock Control Register



MLO-457-85

ERR (Read, write-1-to-clear)
Bit ⟨31⟩ Error
Hardware sets ERR if INT is already set and the Interval Count Register (ICR) overflows. ERR indicates one or more missed clock ticks.

INT (Read, write-1-to-clear)
Bit ⟨7⟩ Interrupt
Hardware sets INT every time ICR overflows. If IE is set, then the overflow generates an interrupt.

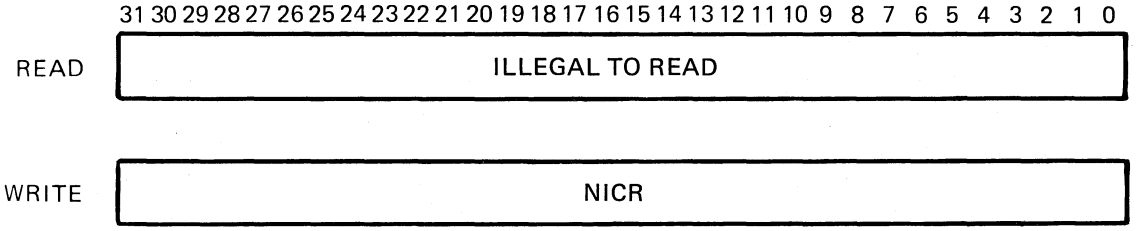
IE (Read, write)
Bit ⟨6⟩ Interrupt Enable
When set, IE enables the hardware to interrupt the processor every time ICR overflows.

SGL (Write only)
Bit ⟨5⟩ Signal
Each time this bit is set, it increments ICR by 1, if RUN is set.

XFER (Write only)
Bit ⟨4⟩ Transfer
Each time this bit is set, the contents of NICR are transferred to ICR.

RUN (Read/write)
Bit ⟨0⟩ Run
ICR increments every microsecond when RUN is set. When RUN is clear, ICR does not increment automatically.

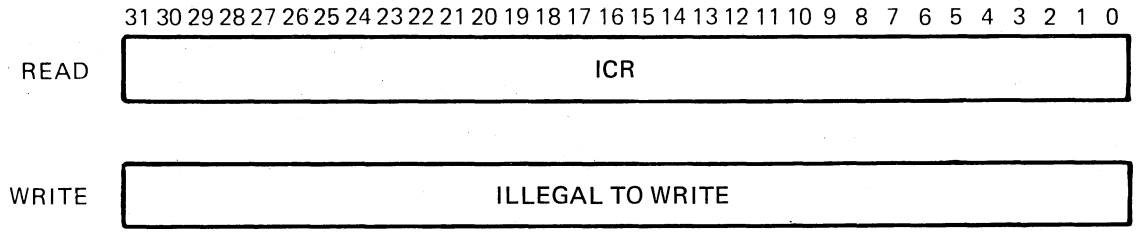
Hex	Dec.	Name	Format
19	25.	NICR	Next Interval Count Register (Write only)



MLO-458-85

NICR is a reload register that holds the value to be loaded into ICR when ICR overflows.

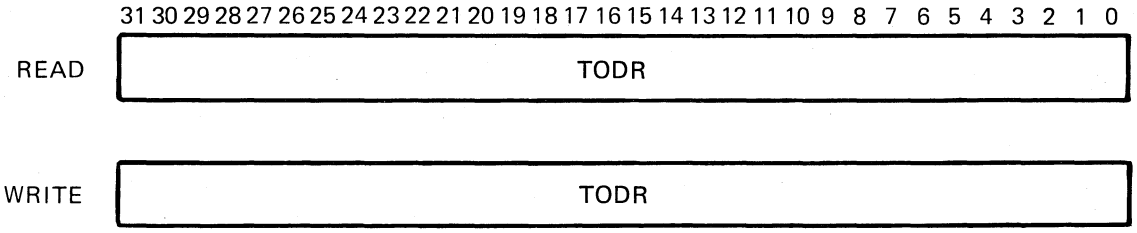
Hex	Dec.	Name	Format
1A	26.	ICR	Interval Count Register (Read/write)



MLO-459-85

Hardware increments this register every microsecond if ICCS bit ⟨0⟩ (RUN) is set.

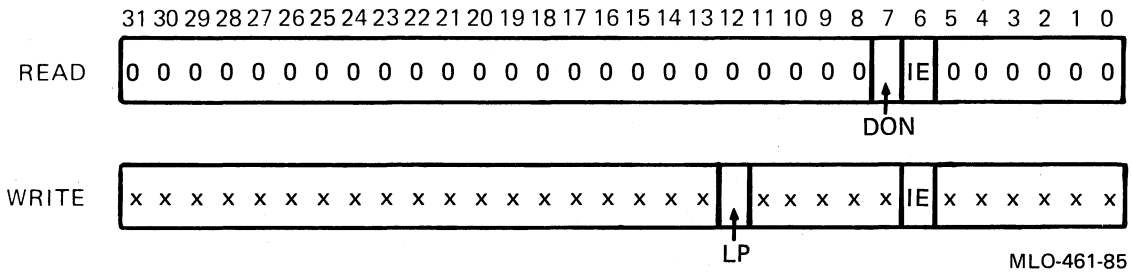
Hex	Dec.	Name	Format
1B	27.	TODR	Time of Day Register



MLO-460-85

TODR is a 32-bit binary counter driven by a precision clock source and incremented every 10 milliseconds. The counter cycles to zero after 497 days.

Hex	Dec.	Name	Format
20	32.	RXCS	Console Receive Control and Status Register

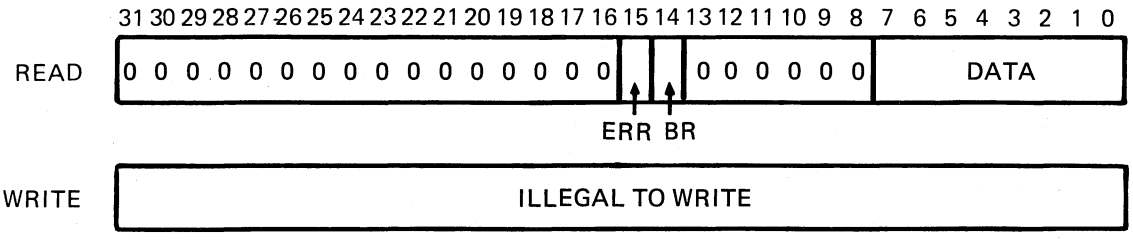


MLO-461-85

RXCS is the control and status register for receiving data from the console via serial-line unit 0. The IE bit should be set to allow interrupts when the RXDB Register receives a character. If the IE bit is set and a character is received, or if a character is received, and then the IE bit is set, an interrupt occurs at IPL 20 and the Done bit is set.

If bit ⟨12⟩ (Loopback — LP) is set, then the serial-line unit's receiver is connected to the M chip's internal loopback bus and disconnected from the external serial line. Any number of the four receiver loopback bits may be set at once, but only one of the four transmitter loopback bits should be set.

21	33.	RXDB	Console Receive Data Buffer Register



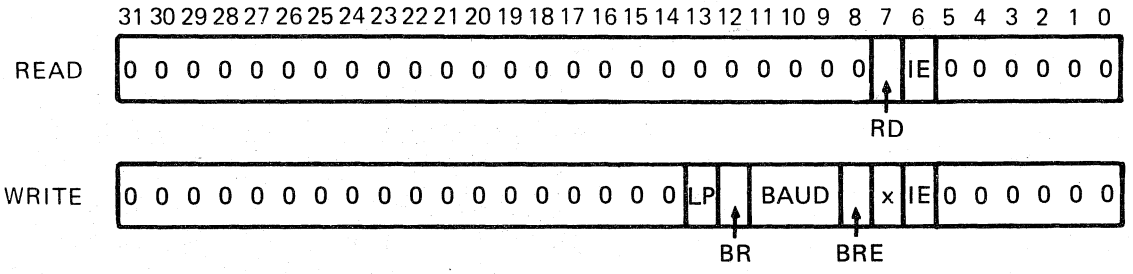
MLO-462-85

RXDB is the data buffer register for serial-line unit 0. Bit ⟨15⟩, the Error bit, is set if the received data has an overrun, or framing error.

Hex	Dec.	Name	Format

Bit ⟨14⟩, the Break bit is set, and the Error bit is not set, if a ⟨BREAK⟩ character is received. When software reads RXDB with an MFPR instruction, microcode clears the Done bit in the RXCS Register.

22 34. TXCS Console Transmit Control and Status Register



MLO-463-85

TXCS is the control and status register for transmitting data to the console on serial-line unit 0. When the console transmitter is not busy, the Ready bit is set, indicating that the TXDB Register is ready to receive a character for transmission. If the IE bit is set when the Ready bit gets set, or if the Ready bit is set and the IE bit subsequently gets set, an interrupt occurs at IPL 20. Software can then write to the TXDB Register with an MTPR instruction to transmit the next character. The MTPR instruction automatically clears the Ready bit. Ready stays cleared until the character is transmitted.

If bit ⟨8⟩ (Baud rate enable — BRE) is set, the contents of bits ⟨11:9⟩ are used to set the new baud rate, both for the transmitter and receiver of serial-line unit 0. If bit ⟨12⟩ (Send BREAK — BR) is set, the serial line sends a continuous BREAK signal until bit ⟨12⟩ is cleared. Software can generate breaks of any duration.

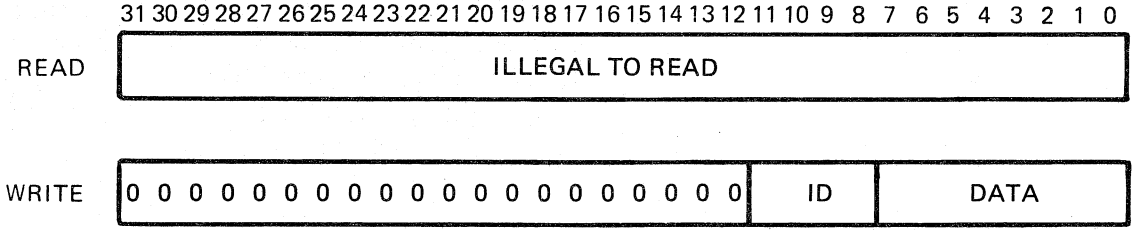
Hex	Dec.	Name	Format

If bit ⟨13⟩ (Loopback — LP) is set, the transmitter for serial-line unit 0 is connected to the M Chip's internal loopback bus, and disconnected from the external serial line (which is held idle). Only one of the four transmitter loopback bits should be set, but any number of the four receiver loopback bits may be set at once.

The following is a table of the baud rates that may be selected in bits ⟨11:9⟩:

Bits ⟨11:9⟩	Baud rate
000	150
001	300
010	600
011	1200
100	2400
101	4800
110	9600
111	19200

23 35. TXDB Console Transmit Data Buffer Register

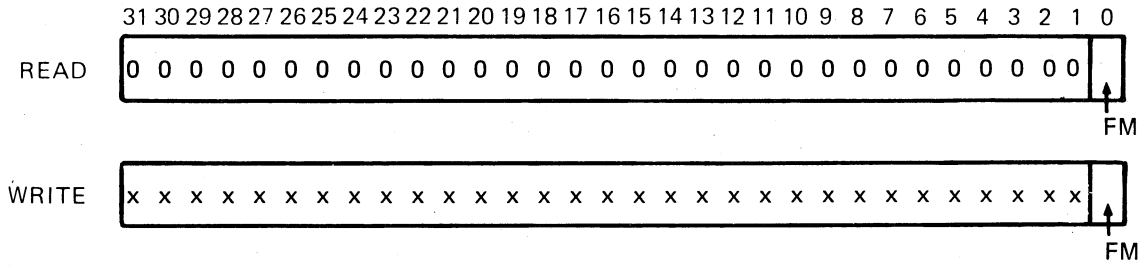


MLO-464-85

TXDB is the data buffer register used to transmit console data on serial-line unit 0. Bits ⟨11:8⟩ form the ID field, which the CPU uses to distinguish between console commands and data. For writing data, bits ⟨11:8⟩ are written with 0000. For sending a command, they are written with 1111. If the ID field is set to 1111, the value in the data field is interpreted as follows:

2 = Boot CPU.
3 = Clear Warm-Start flag.
4 = Clear Cold-Start flag.

Hex	Dec.	Name	Format
24	36.	TBDR	Translation Buffer Disable Register



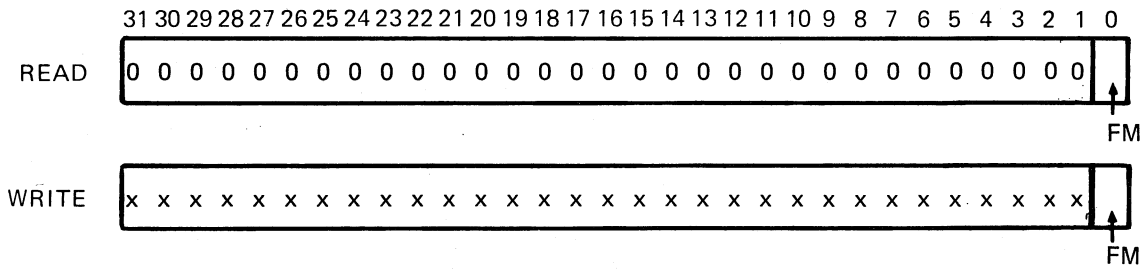
MLO-465-85

Processor initialization microcode sets or clears bit ⟨0⟩ according to the BTB Enable bit in the EEPROM (location 2009 817E). For normal operation software should ensure that bit 0 is clear at power-up.

Software can write TBDR to disable the backup translation buffer (BTB); however, this is only appropriate for diagnostics. TBDR has no effect on the MTB (mini-translation buffer).

Disabling the BTB does not work properly with I-stream (instruction stream) addresses, because of MTB and prefetcher interaction. Therefore software should disable the BTB only when it is operating from physical addresses. Note that bits ⟨31:1⟩ are undefined on a read.

25	37.	CADR	Cache Disable Register



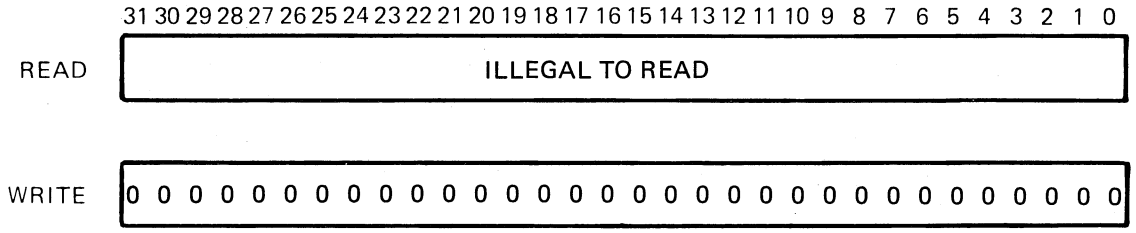
MLO-466-85

Hex	Dec.	Name	Format

Processor initialization microcode sets or clears bit ⟨0⟩ according to the Cache Enable bit in the EEPROM (location 2009 817E). For normal operation software should ensure that bit 0 is clear at power-up.

Software can write CADR to disable the cache; however, this is only appropriate for diagnostics. Microcode interprets data written to CADR and then writes to the cache control/status register within the M chip. Note that bits ⟨31:1⟩ are undefined on a read, but bit ⟨3⟩ is normally read as 1.

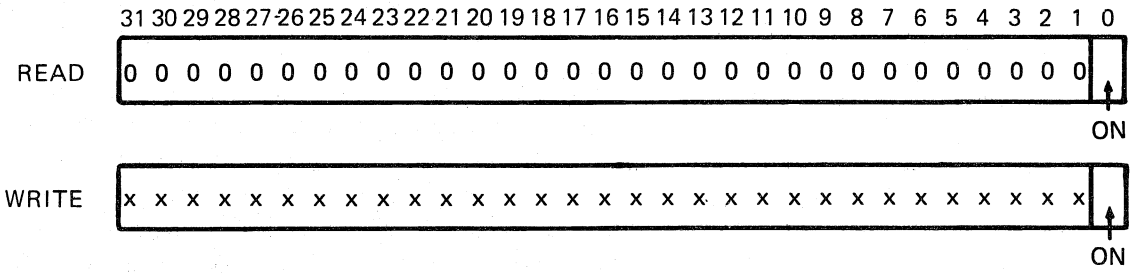
26 38. MCESR Machine Check Error Summary Register



MLO-467-85

Software can use MCESR to clear the machine-check condition flag within the CPU. When the CPU begins to execute the machine-check handling software, and a machine-check condition is detected before this flag is cleared with an MTPR instruction, a CPU double error halt will result. The data written to this register is ignored, but the condition codes will reflect the data. Machine-check software must write to MCESR before returning control to normal VAX code.

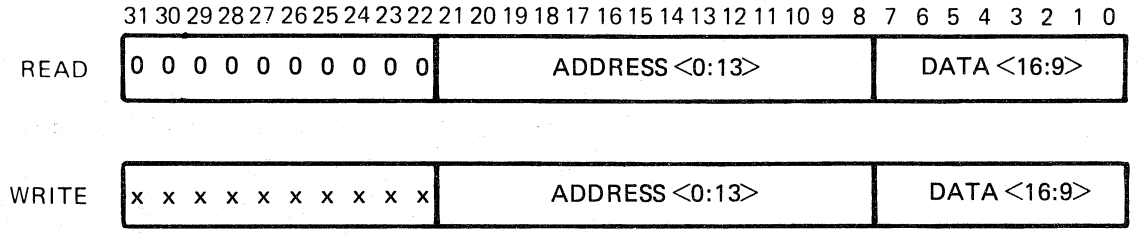
Hex	Dec.	Name	Format
28	40.	ACCS	Accelerator Control and Status Register (floating point)



MLO-468-85

Software can disable the floating-point accelerator chip (F chip) by writing 0 to ACCS. Power-up initialization microcode sets bit ⟨0⟩ to enable the F chip after reading the enable bit in the EEPROM (location 2009 817E).

Hex	Dec.	Name	Format
2C	44.	WCSA	Writeable Control Store (Patch) Address Register



MLO-469-85

This register is used with the WCSD Register for reading the contents of the control store. The order of the address bits is reversed from the normal VAX order, and the data bits are permuted in an unusual way.

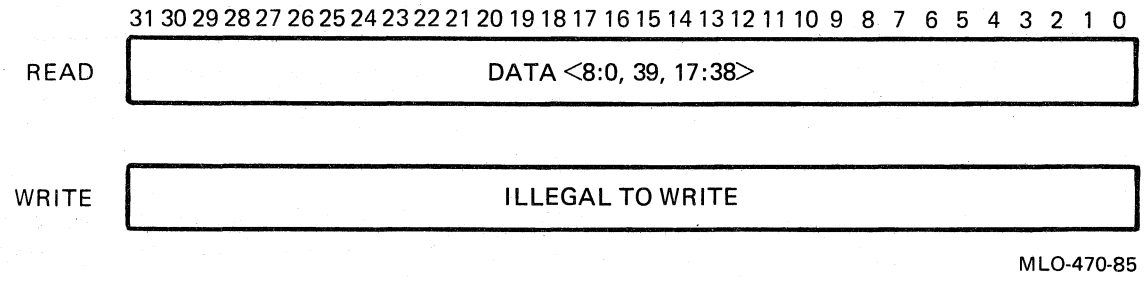
Software can read a control store location, by executing a three-instruction sequence. First use an MTPR instruction to write WCSA with a valid control-store ROM address in bits ⟨21:8⟩ and don't-care data. Then use an MFPR instruction to read WCSD. Then use an MFPR instruction to read WCSA. No interrupts or context changes should be taken within this three-instruction

Hex	Dec.	Name	Format

sequence, or the internal address and data state may be lost. Addresses 0000 to 3BFF refer to control-store ROM locations.

Addresses 3C00 to 3FFB refer to patch RAM locations. Addresses 3FFC to 3FFF refer to the CAM Match Register. When software reads a ROM location that has been patched, the data returned is the patch, not the original ROM word. See Section 3.4.2.3 in Chapter 3 for details.

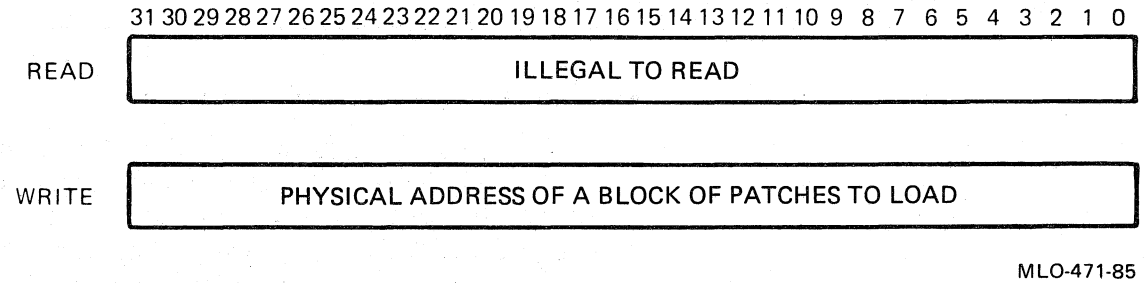
2D 45. WCSD Writeable Control Store (Patch) Data Register



MLO-470-85

This register is used with the WCSA Register for reading the patch RAM portion of the control-store ROM/RAM chips. It contains 32 bits of the 40 bit microword. The data bits are permuted in an unusual way.

2E 46. WCSL Writeable Control Store Load Register



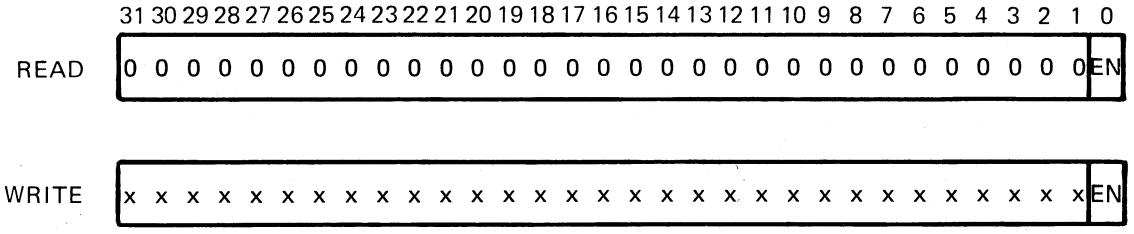
MLO-471-85

Software can load control store patches by writing WCSL with the physical memory address of the block of patches to be loaded, using the MTPR instruction. Microcode calculates a checksum for

Hex	Dec.	Name	Format

the block of patches and compares it with the checksum stored with the block before loading. If the checksum fails, the patches are not loaded, and the V bit in the PSL is set to 1. Otherwise, microcode loads all the patches and clears the PSL V bit. See Section 3.4.2.3 in Chapter 3 for details.

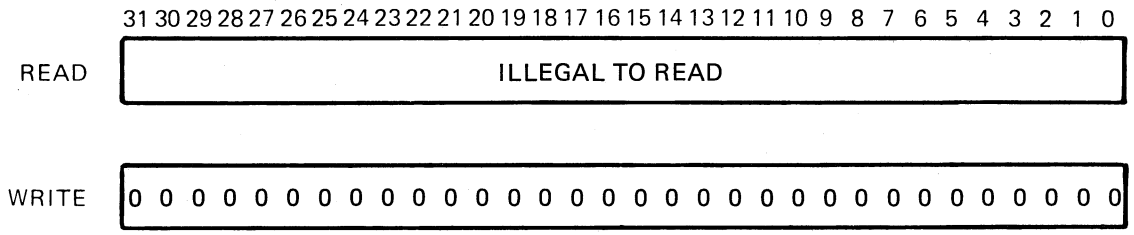
38 56. MAPEN Memory Management Enable Register



MLO-472-85

Setting bit ⟨0⟩ enables memory management. When bit ⟨0⟩ is clear, memory management is disabled.

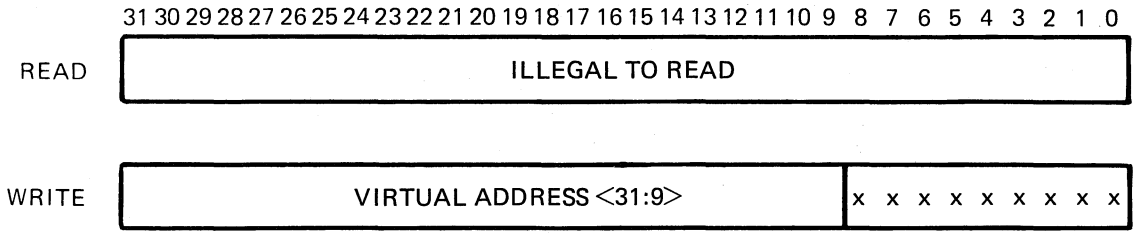
39 57. TBIA Translation Buffer Invalidate All Register



MLO-473-85

Software can invalidate the MTB, BTB, and instruction buffer by writing to TBIA. Microcode responds by clearing all of the valid bits in the translation buffers. The data written to this register is ignored, but the condition codes will reflect the data.

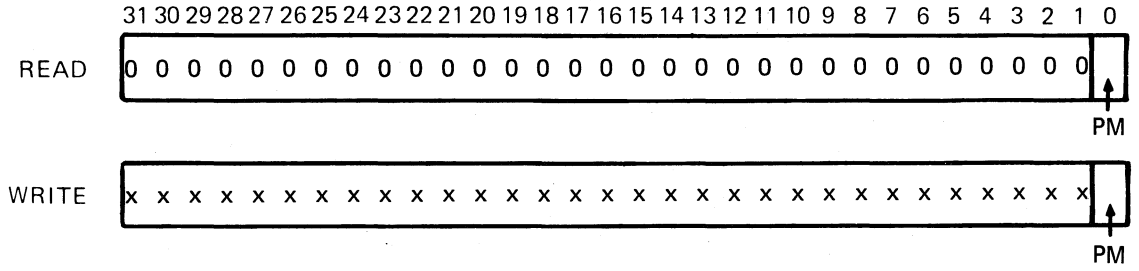
Hex	Dec.	Name	Format
3A	58.	TBIS	Translation Buffer Invalidate Single Register



MLO-474-85

Software can write TBIS to invalidate a specific mapping entry in the BTB, the instruction buffer, or the entire MTB.

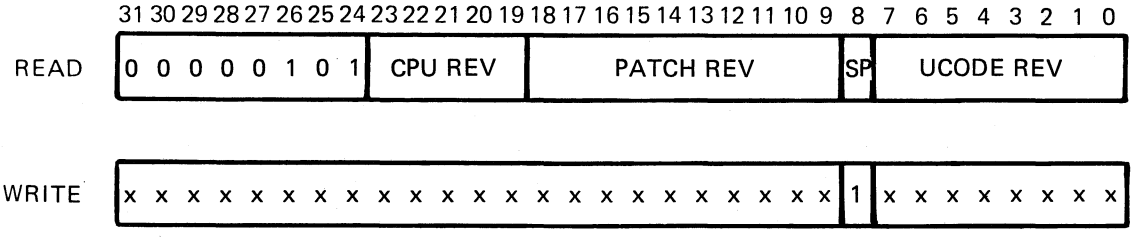
3D	61.	PMR	Performance Monitor Enable Register



MLO-475-85

Bit ⟨0⟩ in PMR controls a signal visible to an external hardware performance monitor. This bit refers only to the current process.

3E	62.	SID	System Identification Register



MLO-476-85

Hex	Dec.	Name	Format

Software uses the SID Register for system identification and revision control. Software should interpret the bits as follows:

Bits ⟨31:24⟩ identify the processor by a unique type number. Type number for KA820 is 5.

Bit ⟨23⟩ identifies the CPU module as a KA825 when set.

Bits ⟨22:19⟩ contain a binary number, which when converted to decimal, indicates the module revision. For example, 1 = rev A; 2 = rev B.

Bits ⟨18:9⟩ contain the revision level of the microcode patches.

Bit ⟨8⟩ is the secondary patch bit (SP), which is written by processor initialization microcode according to data in location 2009 816C in the EEPROM.

SP = 1 Either secondary patches are not needed or they are needed and have been loaded.

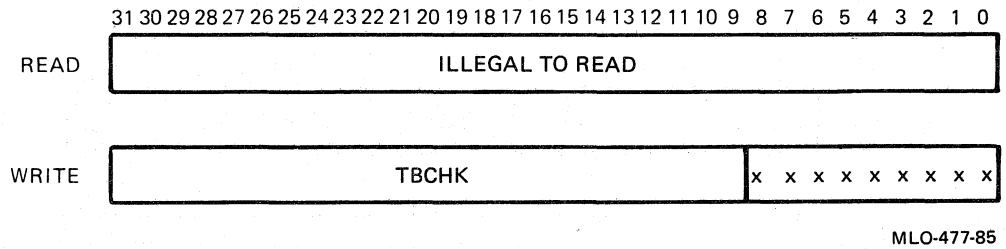
SP = 0 Secondary patches are needed and have not been loaded.

The code that loads secondary patches should set bit ⟨8⟩ when it finishes loading the patches.

Bits ⟨7:0⟩ contain the revision level of the control-store ROM/RAM chips.

Software can write only bit ⟨8⟩, and should write it after loading the secondary patches.

3F 63. TBCHK Translation Buffer Check Register

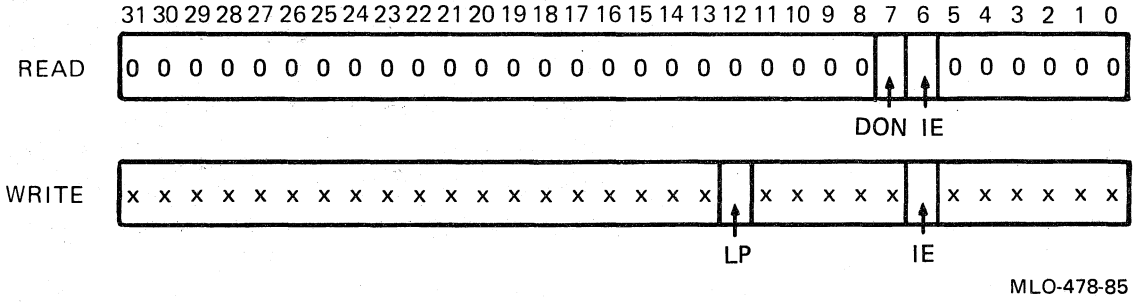


Hex	Dec.	Name	Format

Software can check for the presence of a valid PTE in the BTB by writing the corresponding virtual address in TBCHK. The V bit in the PSL is set if the PTE is valid.

The next 12 items describe the transmit and receive status and data registers for serial-line units 1, 2, and 3.

50 80. RXCS1 Serial-Line Unit 1 Receive Control and Status Register



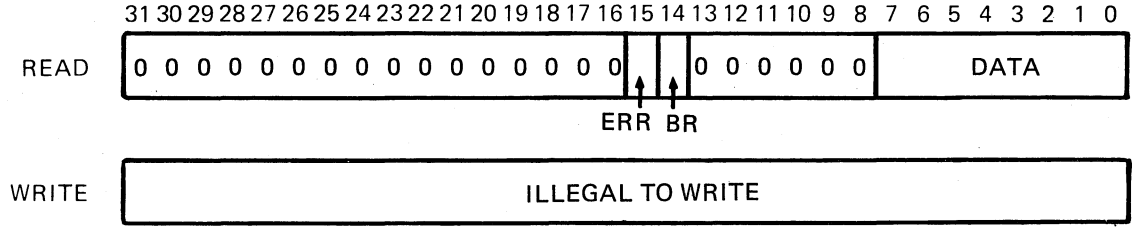
MLO-478-85

RXCS1 is the control and status register for receiving data from serial-line unit 1. The IE bit should be set to allow interrupts when the RXDB1 Register receives a character. If the IE bit is set and a character is received, or if a character is received, and then the IE bit is set, an interrupt occurs at IPL 20 and the Done bit is set.

If bit ⟨12⟩ (Loopback — LP) is set, then the serial-line unit's receiver is connected to the M chip's internal loopback bus and disconnected from the external serial line. Any number of the four receiver loopback bits may be set at once, but only one of the four transmitter loopback bits should be set.

Hex	Dec.	Name	Format

51 81. RXDB1 Serial-Line Unit 1 Receive Data Buffer Register

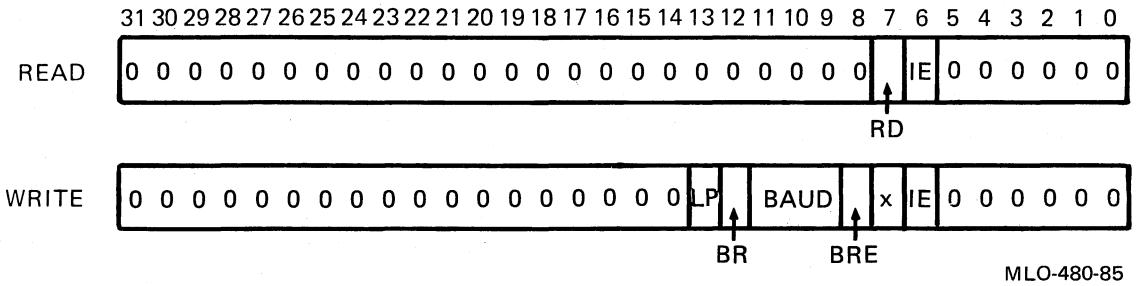


MLO-479-85

RXDB1 is the data buffer register for serial-line unit 1. Bit ⟨15⟩, the Error bit, is set if the received data has an overrun, or framing error.

Bit ⟨14⟩, the Break bit is set, and the Error bit is not set, if a ⟨BREAK⟩ character is received. When software reads RXDB1 with an MFPR instruction, microcode clears the Done bit in the RXCS1 Register.

52 82. TXCS1 Serial-Line Unit 1 Transmit Control and Status Register



MLO-480-85

TXCS1 is the control and status register for transmitting data on serial-line unit 1. When the console transmitter is not busy, the Ready bit is set, indicating that the TXDB1 Register is ready to receive a character for transmission. If the IE bit is set when the Ready bit gets set, or if the Ready bit is set and the IE bit subsequently gets set, an interrupt occurs at IPL 20. Software can then write to the TXDB1 Register with an MTPR instruction to transmit the next character. The MTPR instruction automatically clears the Ready bit. Ready stays cleared until the character is transmitted.

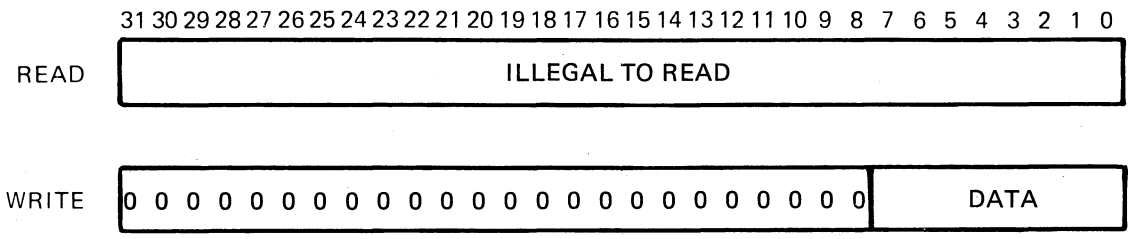
If bit ⟨8⟩ (Baud rate enable — BRE) is set, the contents of bits ⟨11:9⟩ are used to set the new baud rate, both for the transmitter and receiver of serial-line unit 1. If bit ⟨12⟩ (Send BREAK — BR) is set, the serial line sends a continuous BREAK signal until bit ⟨12⟩ is reset. Software can generate breaks of any duration.

If bit ⟨13⟩ (Loopback — LP) is set, the transmitter for serial-line unit 1 is connected to the M Chip's internal loopback bus, and disconnected from the external serial line (which is held idle). Only one of the four transmitter loopback bits should be set, but any number of the four receiver loopback bits may be set at once.

The following is a table of the baud rates that may be selected in bits ⟨11:9⟩:

Bits ⟨11:9⟩	Baud rate
000	150
001	300
010	600
011	1200
100	2400
101	4800
110	9600
111	19200

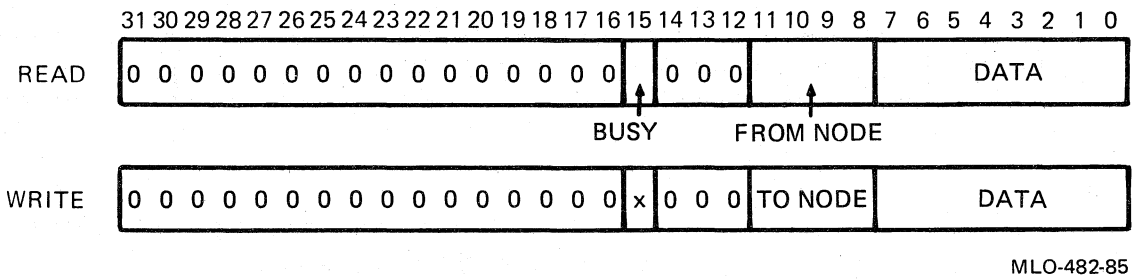
53 83. TXDB1 Serial-Line Unit 1 Transmit Data Buffer Register



MLO-481-85

TXDB1 is the data buffer register used to transmit data on serial-line unit 1.

Hex	Dec.	Name	Format
			NOTE The operation of the registers for serial-line units 2 and 3 is exactly the same as the operation of the registers for serial-line unit 1.
54	84.	RXCS2	Serial-Line Unit 2 Receive Control and Status Register
55	85.	RXDB2	Serial-Line Unit 2 Receive Data Buffer Register
56	86.	TXCS2	Serial-Line Unit 2 Transmit Control and Status Register
57	87.	TXDB2	Serial-Line Unit 2 Transmit Data Buffer Register
58	88.	RXCS3	Serial-Line Unit 3 Receive Control and Status Register
59	89.	RXDB3	Serial-Line Unit 3 Receive Data Buffer Register
5A	90.	TXCS3	Serial-Line Unit 3 Transmit Control and Status Register
5B	91.	TXDB3	Serial-Line Unit 3 Transmit Data Buffer Register
5C	92.	RXCD	Receive Console Data Register



MLO-482-85

Hex	Dec.	Name	Format

The RXCD Register provides for interprocessor console communications on the VAXBI bus. Reading this register picks up a console character (if any) that has been sent to this processor from a processor on another VAXBI node. Writing the RXCD on another processor sends a console character to that processor.

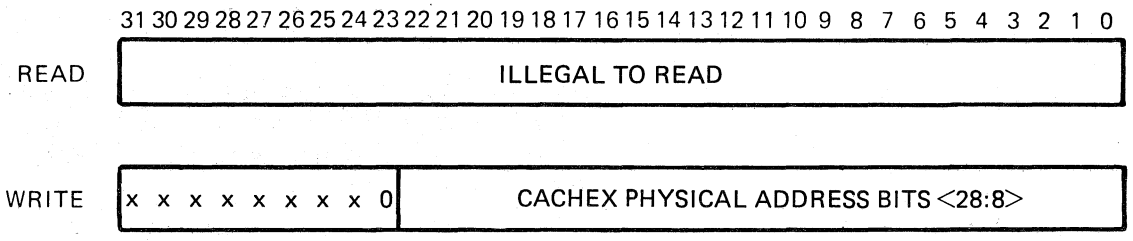
Bit ⟨15⟩ is the Busy bit. When software sends a character, this bit is sent as 1, to indicate that the receiving register is busy. When the receiving processor executes an MFPR instruction to read the character, its microcode resets the receiving processor's RXCD bit ⟨15⟩ to 0.

Bits ⟨11:8⟩ identify the sender's VAXBI node number.

When software reads a not-busy RXCD Register (bit ⟨15⟩ = 0), the MFPR microcode sets the PSW V-bit. When software reads a busy RXCD Register, the MFPR microcode clears the PSW V-bit.

When software writes to a busy RXCD Register on a remote node (bit ⟨15⟩ = 1), no data is sent and the sending MFPR sets the PSW V-bit.

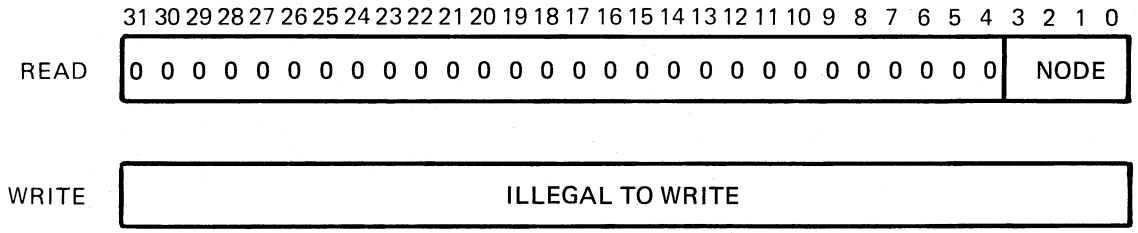
5D 93. CACHEX Cache Invalidate Register



MLO-483-85

Software can invalidate a page in the cache by writing physical address bits ⟨28:8⟩ in bits ⟨20:0⟩ of CACHEX. The CPU clears the four valid bits of up to eight cache tags at one time. Use CACHEX when you use the KA820 module as an I/O processor.

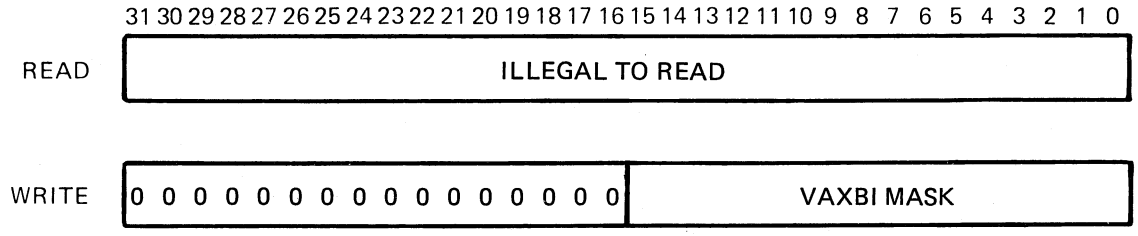
Hex	Dec.	Name	Format
5E	94.	BINID	VAXBI Node Identification Register



MLO-484-85

Software can read BINID to determine the VAXBI node ID of the processor. Bits ⟨3:0⟩ contain the VAXBI node address. Microcode obtains the address from the BIIC Control/Status Register during processor initialization.

Hex	Dec.	Name	Format
5F	95.	BISTOP	VAXBI STOP Register



MLO-485-85

Software uses BISTOP to initiate a VAXBI STOP transaction. BISTOP is a write-only register that software can write with a mask of VAXBI nodes to be stopped.

Appendix G
Register Contents at Power-Up and Boot Entry

Register	Address	Contents at Console Mode Entry	Contents at Change to Macrocode	Comments
PCntl CSR	2008 8000	Varies	Varies	—
DTYPE	2008 0000 bb+00	xxxx 0105	xxxx 0105	xxxx refers to the CPU revision code, microcode patch revision, and secondary patch loading
VAXBI CSR	2008 0004 bb+04	rrtt 0803	rrtt 0803	rrtt refers to the BIIC revision and type
Bus Error Register	2008 0008 bb+08	0000 0000	0000 0000	—
Error Interrupt Control Register	2008 000C bb+0C	0000 0000	0000 0000	—
BCI Control Register	2008 0028 bb+28	0000 0778	0000 0778	—
RXCD	2008 0200 bb+200	0000 0000	0000 0000	—
R0	R0	Random	Random	—
R1	R1	Random	Random	—
R2	R2	Random	Random	—
R3	R3	Random	Boot device ⟨ddxn⟩	See Chapter 4

(Continued on next page)

Register	Address	Contents at Console Mode Entry	Contents at Change to Macrocode	Comments
R4	R4	Random	Random	—
R5	R5	Random	Boot control flags	See Appendix I
R6	R6	Random	Random	—
R7	R7	Random	Random	—
R8	R8	Random	Random	—
R9	R9	Random	Random	—
R10	R10	Random	Halt PC	—
R11	R11	Random	Halt PSL	—
AP	R12	Random	Halt code	—
FP	R13	Random	Random	—
SP	R14	Random	Starting address of primary bootstrap	—
PC	R15	Random	Random	—
TBDR	IPR 24	0000 0000	0000 0000	Affected by EEPROM contents
CADR	IPR 25	0000 0000	0000 0000	Affected by EEPROM contents
ACCS	IPR 28	0000 0000	0000 0000	Affected by EEPROM contents
MAPEN	IPR 38	0000 0000	0000 0000	—

Appendix H
EEPROM Contents

Address	Contents
FIRST EEPROM CHIP	
2009 8000	FF constant used in EEPROM test
2009 8002	55 constant used in EEPROM test
2009 8004 to 2009 8020	14 bytes must be zero
2009 8024	10 bytes, module serial Number 2 characters = plant code 3 characters = numeric date code YWW (Y = year, W = week) 5 characters = numeric module serial number
2009 8038	6 bytes, load server address (for the AIE module)
2009 8044	70 bytes unused and = 0
2009 80D0	8 bytes unused; reserved for DIGITAL CSS
2009 80E0 to 2009 812E	48 bytes reserved to DIGITAL
2009 8130	8 bytes unused; reserved for users
2009 8140 to 2009 8152	17 bytes reserved to DIGITAL
2009 8154	1 byte = AA, the constant that is used in the EEPROM test
2009 8156	1 byte unused
2009 8158	4 bytes for the VAXBI self-test timeout constant incremental value = 0.2 microsecond increment
2009 8160	4 bytes unused

(Continued on next page)

Address	Contents
VAXBI Device Type Data	
2009 8168	2 bytes, VAXBI device type for module = 0105 (hex) for KA820
2009 816C	2 bytes, VAXBI revision level for module bits ⟨15:11⟩ = CPU revision bits ⟨10:1⟩ = patch revision bit ⟨0⟩ = secondary patches not needed, default = 1
2009 8170	3 bytes unused
2009 8176	1 byte for RCX50 self-test disable bit ⟨4⟩: RCX50 self-test disable (0 = enable, 1 = disable), default = 1 bit ⟨3⟩ must be zero bits ⟨7:5,2:0⟩ = 0
Console Source Data	
2009 8178	1 byte bits ⟨3:0⟩ = VAXBI node number of logical console (default = 2) bits ⟨7:4⟩ must be zero
2009 817A	1 byte must be zero
2009 817C	1 byte, UART0 baud rate — default = 1200 baud bits ⟨7:0⟩ = Baud rate as follows: 30 — 150 baud 34 — 2400 baud 31 — 300 baud 35 — 4800 baud 32 — 600 baud 36 — 9600 baud 33 — 1200 baud 37 — 19200 baud
2009 817E	F chip, BTB, and cache disable bits bit ⟨0⟩ (1 = F chip disabled; 0 = F chip enabled), default = 0 bit ⟨1⟩ (1 = BTB disabled; 0 = BTB enabled), must be zero bit ⟨2⟩ (1 = cache disabled; 0 = cache enabled), default = 0 bits ⟨7:3⟩ must be zero
2009 8180 to 2009 81CE	40 bytes unused
2009 81D0	8 bytes: 40-bit checksum of control store with primary patches installed
2009 81E0	2 bytes reserved to DIGITAL
2009 81FC	1 byte: constant 33 used in EEPROM test
2009 81FE	1 byte: constant 01 used in EEPROM test
EEPROM Boot-Code Section	
2009 8200	4 bytes of checksum for 1020 bytes (decimal) of boot code
2009 8208 to 2009 87FE	Boot dispatcher

(Continued on next page)

Address	Contents
EEPROM Patch Section	
2009 8A00	4 bytes of checksum for all the patches
2009 8A08	1 byte for the starting CAM address
2009 8A0A	2 bytes showing the number of patches
2009 8A0E	Start of patches,
to	7 bytes per patch...
2009 BFFE	up to 984 patches (decimal)
SECOND EEPROM CHIP	
2009 C000	Default boot device designation of form ddnu (4 bytes)
2009 C008	Default T/M boot device designation of form ddnu (4 bytes)
2009 C010	Boot-code descriptor A (8 bytes)
2009 C020	Boot-code descriptor B (8 bytes)
2009 C030	Boot-code descriptor C (8 bytes)
2009 C040	Boot-code descriptor D (8 bytes)
2009 C050	Boot-code descriptor E (8 bytes)
2009 C060	Boot-code descriptor F (8 bytes)
2009 C070	Boot-code descriptor G (8 bytes)
2009 C080	Boot-code descriptor H (8 bytes)
2009 C090	Boot-code descriptor I (8 bytes)
2009 C0A0	Boot-code descriptor J (8 bytes)
2009 C0B0	Boot code — including chksum
:	: :
:	: : 8104 bytes (decimal)
:	: :
2009 FFFE	Boot code

LOCATIONS THAT CAN BE READ AND WRITTEN USING CONSOLE COMMANDS E/E AND D/E

EEPROM Address	Typed Location	Configuration Use
2009 8170		
to		
2009 8174	00–02	Reserved for future use
2009 8176	03	RCX50 self-test enable
2009 8178	04	Bits ⟨3:0⟩: VAXBI node number of logical console
2009 817A	05	Unused
2009 817C	06	UART0 baud rate
2009 817E	07	F chip, BTB, and cache disables
2009 8180	08–23	16 bytes unused
to		
2009 81B8		

Appendix I
Software Boot Control Flags

You can control various phases of the boot procedure by setting bits in General Purpose Register R5 with the console command B/R5:⟨data⟩ (see Chapters 3 and 4). These bit functions are defined by the VMB primary boot routine and by VMS. Note that the value -1 in R5 is reserved to DIGITAL.

R5 Bit	Symbol	Function
⟨0⟩	RPB$V_CONV	Conversational boot. At various points in the system boot procedure, the bootstrap code solicits parameters and other input from the console terminal. If bit ⟨4⟩ is also set, the VAX Diagnostic Supervisor should start, enter menu mode, and prompt you for devices to test.
⟨1⟩	RPB$V_DEBUG	Debug. If this flag is set, VMS maps the code for the XDELTA debugger into the system page tables of the running VMS system.
⟨2⟩	RPB$V_INIBPT	Initial breakpoint. If RPB$V_ DEBUG is set, VMS executes a breakpoint (BPT) instruction immediately after enabling mapping.
⟨3⟩	RPB$V_BBLOCK	Secondary boot from boot block. The secondary bootstrap is a single 512-byte block whose logical block number is specified in R4.
⟨4⟩	RPB$V_DIAG	Diagnostic boot. The secondary bootstrap is an image called SYSMAINT DIAGBOOT.EXE.
⟨5⟩	RPB$V_BOOBPT	Bootstrap breakpoint. This stops the primary and secondary bootstraps with a breakpoint (BPT) instruction before testing memory.
⟨6⟩	RPB$V_HEADER	Image header. The transfer address of the secondary bootstrap image comes from the image header for that file. If RPB$VHEADER is not set, control shifts to the first byte of the secondary boot file.

(Continued on next page)

R5 Bit	Symbol	Function
⟨7⟩	RPB$V_NOTEST	Memory test inhibit. This function sets a bit in the PFN bit map for each page of memory present, inhibiting the memory test.
⟨8⟩	RPB$V_SOLICT	File name. VMB prompts for the name of a secondary bootstrap file.
⟨9⟩	RPB$V_HALT	Halt before transfer. VMB executes a HALT instruction before transferring control to the secondary bootstrap.
⟨13⟩	RPB$V_MEMTEST	Specifies that a more extensive algorithm be used when testing main memory for uncorrectable hardware (RDS) errors.
⟨15⟩	RPB$V_AUTOTEST	Used by the VAX Diagnostic Supervisor.
⟨16⟩	RPB$V_CRDTEST	Specifies that memory pages with correctable (CRD) errors not be discarded at bootstrap time. By default, pages with CRD errors are removed from use during the bootstrap memory test.
⟨31:28⟩	RPB$V_TOPSYS	Specifies the top level directory number for system disks in multiple systems.

Appendix J
Sample Bootstrap Code

J.1 EEPROM Bootstrap Dispatcher

```
.TITLE KA820 EEPROM BOOT DISPATCHER
.IDENT /V1.05/

;++
;
;                          COPYRIGHT (c) 1985 BY
;                  DIGITAL EQUIPMENT CORPORATION, MAYNARD,
;                    MASSACHUSETTS. ALL RIGHTS RESERVED.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION
; OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER COPIES THEREOF
; MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO
; TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE, AND
; SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
;
;--
.PAGE
;++
;
; FACILITY:
;
;      KA820 EEPROM BOOT CODE DISPATCHER
;
; AUTHOR:
;
;
;
; ABSTRACT:
;
; This routine interprets the boot command device specification in R3.
; R3 should contain a 4-byte ASCII device specification of the form
; ddxn, where dd is the mnemonic for the generic device type, x is
; the VAXBI node number and n is the unit number. When R3 is set to 0, it
; indicates that the default boot device is being selected. If the
; string is valid, an attempt is made to match the 2-character
; device code with one of the boot device descriptors in the EEPROM.
; If a match is found, control is passed to the appropriate boot
; device. If no match is found, or if an invalid string was
; detected, the message "?44" is printed on the console and the CPU
; halts. All qualifiers to the boot command and the handling of the
; cold and warm restart flags are handled by the CPU microcode.
;
```

```
; This routine resides at offset 104 in the EEPROM.
;
;
; INPUTS:
;       R3 preloaded with boot device specification of form ddxn.
;       R5 set to 11 hex if a CRD boot is to be performed.
;
; Outputs:
;       R0 - destroyed
;       R1 - VAXBI node number
;       R2 - destroyed
;       R3 - unit number
;       R8 - destroyed
;
;--
.PAGE
;
; REVISION HISTORY
;
;
;
.PAGE
        BOOT_DSC = ^X0C0
        PAKET_RAM = ^X20090000
        TXDB = 35
        TXCS = 34
        RDY = 7

;*****************************************************************************
.LONG   ^XF63F4A15                                              ; CHKSUM FOR
                                                                ; FILE HANDLING
;*****************************************************************************

STARTKA8::
        BRB     STARTKA8B               ; Br to start of code.
        .WORD   ENDKA8 - STARTKA8       ; Size of file
        .WORD   105                     ; Version number
        .ASCII  /AK/                    ; Device designation - reversed


STARTKA8B:
        MOVL    PAKET_RAM+BOOT_DSC+2,R0 ; Make R0 point to 1st boot
                                        ; dev desc.
        TSTL    R3                      ; Is R3 = 0?
        BNEQ    1000$                   ; Br if no.
        CMPB    #^X11,R5                ; This a CRD boot?
        BNEQU   900$                    ; Br if no to use normal default
                                        ; (dev A).
        MOVL    22(R0),R3               ; Put CRD default descriptor
                                        ; in R3 (dev D).
        BNEQU   1000$                   ; Br if non 0-else load with
                                        ; normal default.
900$:   MOVL    -2(R0),R3               ; Load default device designation.
1000$:  EXTZV   #16,#16,R3,R2           ; Put device mnemonic in R2.
        EXTZV   #8,#8,R3,R8             ; Put VAXBI node # in R8 to be
                                        ; converted.
        BSBB    CONVERT                 ; Go convert VAXBI node to binary.
        MOVL    R8,R1                   ; Put it in R1.
        MOVL    R3,R8                   ; Setup to convert unit #.
        BSBB    CONVERT                 ; Go convert unit number.
        MOVZBL  R8,R3                   ; Put binary unit number in R3.
```

```
; Match the 2-character device generic code in R2 with the same field
; of the boot device descriptors. If a match is found, we dispatch to
; the boot routine. Otherwise, print a hexadecimal 46 on the console,
; and halt.

        MOVZBL  #4,R8                   ; Loop count in R8.
500$:   CMPW    R2,(R0)+                ; Device match?
        BEQL    MATCH                   ; Br if yes.
        ADDB2   #6,R0                   ; Update pointer.
        SOBGTR  R8,500$                 ; Br if more locations to check.
        ;
        ; Nothing matches - print error message and halt.
        ;
INVALID_DEV:
        MOVZBL  #13,R0                  ; Load (CR).
        BSB     1000$                   ; Go send it.
        MOVZBL  #10,R0                  ; Load a line feed.
        BSB     1000$                   ; Go send it.
        MOVZBL  #^X3F,R0                ; Load a "?" into R0.
        BSB     1000$                   ; Go send it.
        MOVZBL  #^X34,R0                ; Load a "4" into R0.
        BSB     1000$                   ; Go send it.
        MOVZBL  #^X34,R0                ; Load a "4" into R0.
        BSB     1000$                   ; Go send it.
        HALT    ; Halt.
        ;
        ; Subroutine to print the character in R0 on the console
        ;
1000$:  MTPR    R0,#TXDB                ; Load character into data buffer.
2000$:  MFPR    #TXCS,R0                ; Wait until character prints.
        BBC     #RDY,R0,2000$           ; Ditto.
        RSB                             ; Print complete - return.
        ;
        ; A match was found - dispatch to boot routine.
        ;
MATCH:  JMP     @(R0)                   ; GO to boot routine.

; Subroutine to convert ASCII character in R8 to binary and check if valid
; (0 - 15 decimal). If not, an invalid boot device message is printed on
; the console and a halt is executed. Note that it is the converted value
; that is checked. Therefore, if the console accepts a ":", this will
; convert to an A(hex) which is valid.
;
CONVERT:
        BBS     #^X6,R8,100$            ; Branch if its A thru F.
                                        ; Character is in range 0 thru 9.
        SUBB    #^X30,R8                ; Subtract out ASCII.
        BRB     200$
100$:                                   ; Character is in range A thru F.
        SUBB    #^X37,R8                ;
                                        ;
                                        ; Verify character is in the
                                        ; range 0 to F hex.
200$:
        CMPB    #^X0F,R8                ; Is character valid (0-15)?
        BLSSU   INVALID_DEV             ; Br if no.
        RSB
ENDKA8:                                 ; Delimit the end.
        .END    STARTKA8                ; Remove for file inclusion.
```

J.2 Sample RX50 Bootstrap Code

```
.TITLE   RX50 ROM BOOTSTRAP ROUTINE
.ident   /V1.14/

;++
;
;                          COPYRIGHT (c) 1985 BY
;                  DIGITAL EQUIPMENT CORPORATION, MAYNARD,
;                    MASSACHUSETTS. ALL RIGHTS RESERVED.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION
; OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER COPIES THEREOF
; MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO
; TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE, AND
; SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
;
;
;--


;++
;
; FACILITY:
;
;       VAX#8200 Device-Specific RX50 ROM BOOT
;
; AUTHOR:
;
;
; ABSTRACT:
;
;       This routine reads in the boot block from the RX50 diskette and
;       executes it. This is LBN 0 which is track 1 sector 1.
;       The routine assumes that the standard VAX boot block with the
;       first 3 longwords is set up as follows:
;
;           1st longword: Number of blocks in primary boot
;           2nd longword: LBN of primary boot (swapped)
;           3rd longword: Load add relative to start of good 64k (200 hex)
;
;       The boot block code starts immediately after the third longword.
;
;       Drive and disk selection are mapped to the unit number as follows:
;
;               UNIT #0                 Drive 0 Disk 0
;               UNIT #1                 Drive 0 Disk 1
;               UNIT #2                 Drive 1 Disk 0
;               UNIT #3                 Drive 1 Disk 1
;
;
; REVISION HISTORY:
;
;
;--
```

```
.page
.sbttl  Declarations

        READ    = ^O100                 ; RX50 drive specific definitions
        DONE    = ^O10
        RXCMD   = ^O4
        RXBUF   = ^O20
        RXGO    = ^O24
        RXTRK   = ^O6
        RXSEC   = ^O10
        RXCA    = ^O22

        DZ_DEV_TYPE = 64                ; Device type code
        BOOT_CODE_START = 12-^X200      ; Offset to start of boot
                                        ; code
                ; in boot block
        RCX50_REGS = ^X200B0000         ; RCX50  register space
        SIGN_BIT = ^X7                  ; Sign bit position (bit 7)


.page
.sbttl   Boot the boot block

;******************************************************************************
        .long   ^X602DE4BD              ; Patch chksum
;******************************************************************************

;++
; Functional Description
;
; Brings in Logical Block 0 from the RX50 drive 0, diskette 0
;
; Inputs:
;       R3      - Unit number of boot device
;       R5      - Software boot control flags
;       SP      - Base address + ^X200 of 64kb of good memory
;
; Implicit inputs:
;
;       LBN of the boot block is 0
;       Boot block is to be read into -200(sp) - i.e. start of 64k block
;       Xfer address of boot block code is at BOOT_CODE_START(SP).
;
; Outputs: - (inputs to BOOTBLK and primary boot)
;
;       R0      - Device type code for RX50 console device (64.)
;       R1      - 0 (not UNIBUS or MASSBUS)
;       R2      - 0 (not UNIBUS or MASSBUS)
;       R3      - Unit number of boot device
;       R5      - Software boot control flags
;       R6      - Physical address of RX50 mini driver
;       SP      - Base address + ^X200 of 64kb of good memory - passed
;                 parameter
;--
; Boot file header

STARTCSBOOT:
RX50_BOOT:
        BRB      STARTCS                ; Br to start of code
        .WORD   ENDCSBOOT-STARTCSBOOT   ; Size of file in bytes
        .WORD   114                     ; Version number
        .ASCII  /SC/                    ; Device code reversed

; Start of boot code
```

```
STARTCS:
        ADDL    #^X100,SP               ; Move stack into middle of
                                        ; 2nd page to enable boot
                                        ; into 1st page.
        CLRQ    R1                      ; CLR R1+R2 (not UNIBUS
                                        ; or MASSBUS).
        CLRL    R8                      ; LBN to read is set to 0.
2000$:  PUSHAB  -^X300(SP)              ; Push base add of good 64k on stack
                                        ; (Load address of boot block).
        JSB     B^DRIVER$_RX50          ; Go read in a block.
        BLBS    R0,100$                 ; Br if no error.
        HALT                            ; ERROR - HALT.
100$:   MOVZBL  #DZ_DEV_TYPE,R0         ; Load R0 with device type code.
        MOVAB   B^DRIVER$_RX50,R6       ; Make R6 point to start of driver.
        SUBL    #^X100-4,SP             ; Restore SP to original position.
        JMP     BOOT_CODE_START(SP)     ; Pass control to the boot block.


.page
.sbttl  RX50 MINI DRIVER

;++
; RX50 mini driver. Reads in a logical block.
; It is called as a subroutine. The physical address in which the
; block is to be loaded must be placed on the stack prior
; to calling, which will put it at 4(SP) upon entry.
;
; Inputs:
;       R8      logical block number
;       4(SP)   physical load address
;
; Outputs:
;       R0      simple completion status - low bit set = success
;
;--


DRIVER$_RX50:

        PUSHR   #^M<R2,R3,R5,R6,R7,R8,R9> ; Save volatile registers.

;       ****** (affects offset @ 3000$)******
; Convert unit number in R3 to drive and disk select.
; Convert LBN in R8 to physical device address.
;
;       INPUTS:                         R8   LBN
;       R3      Unit Number
;
;       OUTPUTS:                        R5 drive and disk select data
;       R6      track number
;       R8      sector number
;
; NOTES: SIDE = 0                       BN = LBN MOD 800 (BLK # on a side)
;       TRK = BN DIV 10                 BNA = BN MOD 10 (BLK # on a track)

; Convert unit number in R3 to disk and drive select
        DECL    R3                      ; Dec unit number so unit 1 select
                                        ; drive 0.
        CMPB    R3,#3                   ; Is unit number valid (in range 0-3)?
        BGTRU   ERROR                   ; Br if no.
        ADDB3   R3,R3,R5                ; Shift to left and put in R5.
        CLRL    R9                      ; Clear for EDIV.
```

```
        EDIV    #10.,R8,R6,R7           ; R6(=TRK R7(=BNA
        ADDL2   R6,R8                   ; R8(== BN + TRK
        MULL2   #2,R8                   ; R8(== (BN+TRK)*2
        DIVL2   #5,R7                   ; R7(== BNA/5
        ADDL2   R7,R8                   ; R8(==(BNA/5 + ((BN+TRK)*2))
        EDIV    #10.,R8,R0,R8           ; R8 (==R8 MOD 10
        INCL    R8                      ; Make sectors start at 1.
        INCL    R6                      ; Make tracks start at 1.
        CMPB    #80.,R6                 ; Is this the last track on surface?
        BNEQ    300$                    ; Br if no.
        CLRL    R6                      ; Set track to 0 (last track=0).

; Conversion of LBN to physical address is complete.
; Read in a single block using the physical information.

300$:   MOVL    #RCX50_REGS,R2          ; Load R2 with RX50 register add.
        CLRL    R0                      ; Preset return status to error.
        BISB    #READ,R5                ; Combine read code with disk
                                        ; and drive select.
        MOVB    R5,RXCMD(R2)            ; Set func to read sector.
3000$:  MOVL    4+28(SP),R5             ; Move physical address of buffer to R5.
        MOVB    R6,RXTRK(R2)            ; Select the track.
        MOVB    R8,RXSEC(R2)            ; Select the sector.
        MOVB    #0,RXGO(R2)             ; Start the read sector operation.
500$:   BITB    #DONE,RXCMD(R2)         ; Done yet?
        BEQL    500$                    ; Br back to wait for done.
        BBS     #SIGN_BIT,RXCMD(R2),ERROR ; Br if error.
        TSTB    RXCA(R2)                ; Clear silo address.
        MOVZWL  #512,R9                 ; Load loop count.
700$:   MOVB    RXBUF(R2),(R5)+         ; Load a byte into the buffer.
        SOBGTR  R9,700$                 ; Loop until entire block is loaded.
        INCL    R0                      ; Indicate success.
ERROR:  POPR    #^M<R2,R3,R5,R6,R7,R8,R9> ; Restore volatile registers.
        RSB                             ; Return.

ENDCSBOOT:                              ; Delimit the last location.
        .end    STARTCSBOOT             ; Remove for multifile inclusion.
```

J.3 Sample DU Series Bootstrap Code (MSCP Devices)

```
.TITLE   VAX#8200 BUA MSCP Boot ROM
.IDENT  /V1.01/

;
;                       COPYRIGHT (c) 1985 BY
;                  DIGITAL EQUIPMENT CORPORATION, MAYNARD,
;                    MASSACHUSETTS. ALL RIGHTS RESERVED.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
; ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION
; OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER COPIES THEREOF
; MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON. NO
; TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE, AND
; SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT CORPORATION.
;
; DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
;

;++
;
```

```
; FACILITY:
;
;       VAX#8200 DU Boot Device Type Bootstrap ROM
;
; ABSTRACT:
;
;       This ROM code reads in the boot block from an MSCP speaking device
;       which adheres to the UQSSP port specification. It supports the
;       VAXBI to UNIBUS adapter (BUA), the VAXBI Low End Storage
;       Interconnect Adapter (BLA) and the VAXBI to Disk Adapter (BDA).
;
;       The code reads in the boot block and executes it. It assumes the
;       boot block to be the VAX standard boot block. The default addresses
;       for the SA and IP registers are always used. There is no mechanism
;       for controller selection on a given node.
;
; AUTHOR:
;
;
;
; REVISION HISTORY:
;
;
;
;--
.PAGE
.SBTTL  Memory Map Of Boot Process
;  Base of good
;  memory +       0 +--------------------------+
;                   |                          |
;                   |                          |
;               1FF |                          |
;                   +--------------------------+ <--- SP Passed to ROM
;                   |                          |      code boot block
;                   |                          |
;                   |                          |
;                   |                          |
;                   |                          | <--- Available for file
;                   |                          |      brought in by
;                   |                          |      boot block
;                   |                          |
;                   |                          |
;                   +--------------------------+
;              FA00 |                          | <--- Host
;                   |                          |      communications
;                   |                          |      area
;                   |                          |
;                   |                          |
;                   |                          |
;                   |                          | <--- Available for file
;                   |                          |      brought in by
;                   |                          |      boot block
;              FFFF |                          |
;                   +--------------------------+
;                                              |
;                                              |
;                                              V      Untested memory
.PAGE
.SBTTL  Notes

;       . There is no check that the BUA passed its self-test.
;       . No check that selected node has a BUA.
;       . There is no mechanism for using an address other than the
;       . default address.
```

```
;         The code for the BLA and BUA is identical. The BDA has
;         significant differences as follows:
;            1. The IP address reg is in the VAXBI node space at offset F2.
;            2. All addresses used by the BDA are physical. Addresses used
;               by the BUA and BLA are relative to the start of good 64k.
;               This affects the address of the communications area in the
;               initialization table and the addresses in the buffer
;               descriptors.
;            3. The BDA does not initiate initialization when the IP register
;               is written. It is started by setting the Node Reset (NRST)
;               bit in the VAXBI CSR.
;
;            4. ***RESTRICTIONS***
;               The communications area starts at FA00. If the boot block
;               loads a program into the first 64k of good memory, the
;               communications area will be overwritten if the program is
;               too big. The boot block can however load large programs
;               starting beyond the communications area. This is the case
;               for VDS, which loads into the second good 64k.
;
;               . The boot block and any program that it loads must be position
;               independent if it is to be guaranteed to work properly.
;               Otherwise problems will arise if the start of good memory is
;               not at location 0.
;


.PAGE
.SBTTL Declarations

BUA_BLA_DTYPE   = 17                    ; Boot code device type for UDA BLA
BDA_DTYPE       = 33                    ; Boot code device type for BDA
BOOT_CODE_START = 12 - ^X200

;
; UDA I/O ADDRESSES
;

DU_UDAIP = ^O012150                     ; Address of initialize and polling
                                        ; register
DU_UDASA = DU_UDAIP + 2                 ; Address of status and address
                                        ; register
SA = DU_UDASA - DU_UDAIP                ; Offset between the registers
;
; Bit positions for initialization
;

ERR = 15                                ; Error bit
S4  = 14                                ; Step 4 bit
S3  = 13                                ; Step 3 bit
S2  = 12                                ; Step 2 bit
S1  = 11                                ; Step 1 bit

;
; Bit definitions for initialization
;

GO   = ^O0000001                        ; Go
STEP = ^O0100000                        ; Step indicator

;
; Host communications area definitions
;
```

```
OWN = ^X80000000                        ; Ownership of packet, 0 = host,
                                        ; 1 = UDA

;
; Communications area offsets
;
                                        ; 128 bytes needed for host comm
                                        ; area
RING = 0                                ; Host communications area
RMSG = RING + 12                        ; Buffer for response packet
                                        ; from UDA
RCMD = RMSG + 64                        ; Buffer for command packet to UDA
COMM_END = RCMD +48                     ; Length of comm. area - 4 bytes
;
; Command/response packet definitions
;

OP_ONL = ^011                           ; On-line command op-code
OP_RD  = ^041                           ; Read command op-code

P_UNIT =  4                             ; Unit number offset into
                                        ; packet
P_LBN  = 28                             ; LBN offset into packet
P_BUFF = 16                             ; Buffer descriptor offset into packet
P_SHUN = 32                             ; Shadow unit number
P_OPCD =  8                             ; Op-code offset into packet
P_BCNT = 12                             ; Byte count offset into packet
P_STS  = 10                             ; End packet status offset
P_LEN  = -4                             ; Length of packet offset


; BUA Definitions and offsets

        BIDEVCODE_BUA = ^X102           ; VAXBI device code for BUA
        BIDEVCODE_BLA = ^X103           ; VAXBI device code for BLA
        BIDEVCODE_BDA = ^X10E           ; VAXBI device code for BDA
        BI_ADD  = ^X20000000            ; VAXBI node address space start
        BICSROFF = ^X04                 ; Offset to VAXBI CSR register
        BDAIPADD_OFF = ^X0F2            ; Offset to BDA IP reg
        NEXT_NODE = ^X00002000          ; Offset to next VAXBI node
        ADAPTER_WINDOW = ^X20400000     ; Add of 1st adapter window
        NEXT_WINDOW = ^X40000           ; Offset to end of adapter window
        START_ADD_OFF = ^X20            ; Offset to starting address
        END_ADD_OFF = ^X24              ; Offset to end address
        MAP_REG_OFF = ^X800             ; Offset to BUA map reg
        VALID = (1@31)                  ; Valid bit for map registers
        SST = (1@10)                    ; Node reset bit in BICSR

.PAGE
.SBTTL  BUA Initialization and Set-up


;-----------------------------------------------------------------------
;        SET UP THE NODE
;-----------------------------------------------------------------------
;
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       Loads R10 with the base address of the VAXBI node space, then
;       performs setup based on the node type as follows:
;
```

```
;       BDA node
;       --------
;       Load R11 with device type code for BDA.
;       Load R2 with address of IP reg.
;       Load R0 with R2 + 2 (BDA has separate SA registers for write
;       and read).
;       Make address of communications area in Init table physical (RINGAD).
;       Load ADD_TYPE_FLAG with 1 to indicate BDA/Physical addressing.
;
;       Non BDA node
;       ------------
;       Load R11 with device type code for BUA/BLA.
;       Load R2 with physical address of controllers CSR (IP reg).
;       Load R0 with R2.
;       Sets up the VAXBI node by loading the start and end addresses of
;       the adapter window into the BIIC of the BUA. Sets up the
;       UNIBUS map registers and sets up R2 with CSR (IP) address.
;
; INPUTS
;
;       R1        - VAXBI node number for this BUA
;       R3        - Unit number of boot device in binary
;       R5         Software boot control flags
;       SP        - (base_address + ^X200) of 64kb of good memory
;
;
; OUTPUTS:
;
;       R1        - VAXBI node number for this BUA
;       R2        - Physical address of the boot deviceIs CSR
;       R3        - Unit number of boot device in binary
;       R5        - Software boot control flags
;       R10       - Base address of the VAXBI node
;       SP        - (base_address + ^X200) of 64kb of good memory
;
; Work Registers:
;       R4 and R8
;
;--
;
;*****************************************************************************
        .LONG   ^X7A1E394D              ; CHKSUM - FOR FILE HANDLING
                                        ; PURPOSES ONLY
;*****************************************************************************
;
;  BOOT CODE HEADER
;
STARTDUBOOT:
        BRB     STARTDU                 ; Branch to start of code.
        .WORD   ENDDUBOOT - STARTDUBOOT ; Size of file in bytes
        .WORD   101                     ; Version number
        .ASCII  /UD/                    ; Device designation reversed
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;  Location to change if you have a second controller on the UNIBUS !
;       ^X1468 for 1st controller ^X00DC for second controller
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
UDAIPADD:
        .LONG    ^X1468                 ; UDA IP address

AZTEC_BOOT:
RA60_BOOT:
STARTDU:
        ADDL    #^X100,SP               ; Move stack off of page #0 for now
                                        ; to make room for boot block.
        SUBL3   #^X300,SP,R7            ; Pointer to start of good 64k==)R7
        MOVZBL  #BUA_BLA_DTYPE,R11      ; Set device code for BUA, BLA.
;
```

```
; Setup pointers: R10(=base add of VAXBI node, R8(=add of start add
; field.
;
        MULL3   R1,#NEXT_NODE,R10       ; Calculate VAXBI node address.
        ADDL2   #BI_ADD,R10             ; Get starting add of VAXBI
                                        ; node in R10.
        ADDL3   #START_ADD_OFF,R10,R8   ; Get add of starting add
                                        ; field in R8 .
        CMPW    #BIDEVCODE_BDA,(R10)    ; Is this VAXBI node a BDA?
        BNEQU   100$                    ; Br if no - assume BUA or BLA.
;
; Node is verified to be a BDA
;-------------------------------------------------------------------------
;   Load R11 with device type code for BDA.
;   Load R2 with IP address.
;   Load R0 with R2 + 2.
;   Make add of communications area in the init table physical (RINGAD).
;   Load ADD_TYPE_FLAG with 1 to indicate physical addressing/BDA.
;
        MOVZBL  #BDA_DTYPE,R11          ; Load R11 with device code for BDA.
        MOVAL   BDAIPADD_OFF(R10),R2    ; Load R2 with phys address of
                                        ; IP reg.
        ADDL3   #2,R2,R0                ; Load R0 with IP add+2 (For
                                        ; BDA writes.
        ADDL2   R7,W^RINGAD             ; Make add in init table physical.
        MOVB    #1,W^ADD_TYPE_FLAG      ; Indicate address type is physical.
        BRB     CONT                    ; Branch to continue.
;
; Node is assumed to be BUA or BLA
;-------------------------------------------------------------------------
;   Setup starting and end address of adapter window in BIIC.
;   Load R2 with physical address of the controllers CSR IP add.
;
100$:   MULL3   #NEXT_WINDOW,R1,R2      ; Calculate adapter window.
        ADDL3   #ADAPTER_WINDOW,R2,(R8) ; Load starting add in BIIC.
        ADDL3   #^O760000,(R8),R2       ; R2(=addres of base of UNIBUS
                                        ; I/O page.
        ADDL2   UDAIPADD,R2             ; R2(==Physical add of
                                        ; controllers CSR
        MOVL    R2,R0                   ; R0(==CSR add (for BDA
                                        ; compatiblty)
        ADDL3   #NEXT_WINDOW,(R8)+,(R8) ; Load end add in BIIC.
;
; Setup the UNIBUS map registers - Map registers 0 thru xxx are mapped
; sequentially starting at start of 64K of good memory.
; Mapping is such that address 0 is start of good 64k.
; The VALID bit is set and the non buffered data path selected.
; (At this point - R8 points to the end address field)
;
        ADDL2   #MAP_REG_OFF-END_ADD_OFF,R8 ; Pointer to Map
                                        ; registers==)R8
        MOVL    R7,(R8)                 ; Starting add of good 64k =)
                                        ; 1st map reg
        ASHL    #-9,(R8),(R8)           ; Position physical add 29:9
                                        ; into 20:0.
        BISL2   #VALID,(R8)             ; Set Valid bit in 1st map
                                        ; table entry.
        MOVZWL  #400-1,R4               ; Map table entry count ==) R8.
1000$:  ADDL3   #1,(R8)+,(R8)           ; Load map table entry.
        SOBGTR  R4,1000$                ; Br back if more to load.

.PAGE
.SBTTL  ROM Control Subroutine
```

```
;++
;
; FUNCTIONAL DESCRIPTION:
;
;       The subroutine reads block 0 off the boot device's volume,
;       loads it into the first page of usable memory (the page
;       mapped by UNIBUS map register #0) and transfers control to
;       the fourth longword (byte #12) of this boot block.
;
; INPUTS:
;
;       R0 - Adjusted IP address
;          = R2 for BLA and BUA
;          = R2+2 for BDA (accommodates different add for writes)
;       R1      - VAXBI node number for this BUA
;       R2      - Physical address of the boot devices CSR
;       R3      - Unit number of the boot device
;       R5      - Software boot control flags
;       SP      - (base_address + ^X200) of 64kb of good memory
;
; IMPLICIT INPUTS:
;
;       LBN of boot block is 0.
;       Boot block is to be read into (SP)-^X200 (i.e. relative 0).
;       Transfer address of boot block code is at BOOT_CODE_START(SP).
;
; OUTPUTS:
;
;       R0      - Adjusted IP address
;       = R2 for BLA and BUA
;       = R2 + 2 for BDA (accommodates different add for writes)
;       R1      - VAXBI node number for this BUA
;       R2      - Physical address of the boot device's CSR
;       R3      - Unit number of the boot device
;       R5      - Software boot control flags
;       R6      - Address of the device-specific read block routine
;       SP      - (base_address + ^X200) of 64kb of good memory
;
;       This routine preserves registers R3, R5, R10-R11, AP, and SP.
;
;--

CONT:   PUSHR   #^M<R5>             ; Save register for temp.
;
;-------------------------------------------------------------------
;       INITIALIZE THE CONTROLLER AND TELL IT TO GO
;-------------------------------------------------------------------

        MOVL    #2,R6               ; Number of retries

IERROR: SOBGEQ  R6,RETRY            ;
        HALT                        ; Halt after two tries.

RETRY:  BLBS    ADD_TYPE_FLAG,100$  ; Br if this is a BDA.
        CLRW    (R2)                ; Start init by writing IP reg
                                    ; (BUA/BLA).
        BRB     500$                ; Br to continue.
100$:   BISW    #SST,BICSROFF(R10)  ; Start init by setting SST
                                    ; bit in BICSR.
500$:   MOVL    #S1,R5              ; Step 1 compare data ==) R5.

        MOVAB   B^TABLE1,R8         ; Pointer to init data table==)R8
ILOOP:  MOVW    SA(R2),R9           ; Read status address reg.
        BLSS    IERROR              ; Br if error to retry.
        BBC     R5,R9,ILOOP         ; Wait for proper step bit to set.
        MOVW    (R8)+,SA(R0)        ; Write next command to UDASA.
        AOBLEQ  #S4,R5,ILOOP        ; Repeat until finished.
```

```
;-----------------------------------------------------------------
; Do set-up and call primitive driver to load LBN 0 - pass control to
; boot block
;-----------------------------------------------------------------
;
; Set up output registers and call a device-specific subroutine that
; reads in one block off the boot device. The block is block 0, the
; boot block.
;
; Set up registers as follows:
;
;         R2 - boot device's CSR address
;         R3 - device unit number
;         R5 - Relative address
;         R8 - LBN 0

        CLRL    R5                      ; Put relative load address in R5.
        CLRL    R8                      ; Set LBN to 0 ==) R8.
        PUSHL   R7                      ; Put physical load address on
                                        ; stack.
        BSBB    BOO$UDA50_QIO           ; Call driver to read in LBN 0.
        BLBS    R0,10$                  ; Branch on successful read..
        HALT                            ; Error, halt processor.
;
; Set up the remaining registers needed by VMB and by the boot block.
; Then transfer control to the boot block code.
;

10$:                                    ; Read was successful.
        ADDL2   #4,SP                   ; Remove physical load add from
                                        ; stack.
        MOVAB   B^BOO$UDA50_QIO,R6      ; Store address of driver.
        MOVL    R11,R0                  ; Load device type.
        POPR    #^M<R5>                 ; Restore register.
        SUBL    #^X100,SP               ; Restore original stack position.
        JMP     BOOT_CODE_START(SP)     ; Give control to boot block.
.PAGE
.SBTTL UDA TABLES - tables used for UDA initialization

;-----------------------------------------------------------------
;       TABLES CONTAINING DATA FOR INITIALIZATION SEQUENCE
;-----------------------------------------------------------------
; The address of the ring base as given below is relative. If the
; device does not MAP to the start of good 64k (such as the BDA),
; then this address will be overwritten with the physical address as
; part of the set-up.
;
;
TABLE1: .WORD    STEP                   ; Step 1 - desc lengths = 1
                                        ; (2**0), vector
RINGAD: .WORD   ^XFA04                  ; Step 2 - lo address of ring base
        .WORD   0                       ; Step 3 - high address of
                                        ; ring base
        .WORD   GO                      ; Step 4 - go bit to enable UDA


;-----------------------------------------------------------------
;       Boot code internal storage area
;-----------------------------------------------------------------
;
;       FLAG field indicating type of address and BDA/non BDA

ADD_TYPE_FLAG: .BYTE 0                  ; 1=PHYSICAL-BDA/0=RELATIVE
```

```
.PAGE
.SBTTL   BOO$UDA50_QIO - primitive device-dependent read/write driver


;-----------------------------------------------------------------------
;       UQ PORT PRIMITIVE DEVICE DRIVER
;-----------------------------------------------------------------------
;
;++
;
; FUNCTIONAL DESCRIPTION:
;
; INPUTS:
;
;        R2    - physical address of boot device's CSR (IP register)
;        R3    - unit number of boot device
;        R5    - starting address of transfer (relative to load address)
;        R8    - LBN to transfer from boot device
;        4(SP) - physical address of transfer
;
; IMPLICIT INPUTS:
;
;       RINGAD is loaded with either physical or relative address of
;       ring base ADD_TYPE_FLAG is set up to indicate physical or
;       relative addressing for BUA and BLA. The UNIBUS adapter map
;       registers are already set up. The 64k of good memory are
;       mapped to map registers 0-127. Additional map registers
;       (up to 399) are set up to handle direct booting of the
;       VAX Diagnostic Supervisor.
;
;
; OUTPUT:
;
;        R0    - SS$_NORMAL on successful read
;                low bit clear on error during read
;        R7-R9 - scratch registers
;
;        This routine preserves registers R1-R6, R10-R11, AP, and SP.
;
;--
BOO$UDA50_QIO:
       SUBL3   R5,4(SP),R7             ; Get phys add of communications
                                       ; area.
       ADDL2   RINGAD,R7               ; (Phys add - rel add + ringadd)
       ;
       ; Issue ONLINE command
       ;
100$:  BSBB    SETUP_IO                ; Subroutine sets up parts of
                                       ; packet common to ONLINE
                                       ; and READ commands.
                                       ; NOTE: SETUP_IO returns R0 = 0.
       MOVW    #OP_ONL,B^RCMD+P_OPCD(R7); Set ONLINE opcode into packet.
       TSTW    (R2)                    ; Tickle UDA, cause it to initiate
                                       ; polling.
XDONE0:TSTL    (R7)                    ; Wait until UDA says that we
       BLSS    XDONE0                  ; are finished (ownership bit=:)0).
       TSTB    B^RMSG+P_STS(R7)        ; Check status in response packet.
       BNEQ    QIO_RETURN              ; Br on error to QIO_RETURN.
       ;
       ; Set up the command packet to read a single logical block.
       ;
```

```
        BSBB    SETUP_IO                ; Subroutine sets up parts of
                                        ; packet common to ONLINE
                                        ; and READ commands.
                                        ; NOTE: SETUP_IO returns R0 = 0.
        MOVW    #OP_RD,B^RCMD+P_OPCD(R7); Load "READ" opcode into packet.
        MOVL    R8,B^RCMD+P_LBN(R7)     ; Load LBN in packet.
        MOVL    4(SP),B^RCMD+P_BUFF(R7) ; Load physical address of data
                                        ; buffer.
        BLBS    ADD_TYPE_FLAG,100$      ; Br if physical addressing (BDA).
        MOVL    R5,B^RCMD+P_BUFF(R7)    ; Overlay physical add with
                                        ; relative.
100$:   MOVL    #2,B^RCMD+P_BCNT+1(R7)  ; Byte count is 512 (2*256).
                                        ;
        TSTW    (R2)                    ; Access IP reg to cause
                                        ; controller to initiate polling.
XDONE1:TSTL     (R7)                    ; Wait until UDA says that we
        BLSS    XDONE1                  ; are finished.
        TSTB    B^RMSG+P_STS(R7)        ; Check status.
        BNEQ    QIO_RETURN              ; Return error.

        INCL    R0                      ; Return successful transfer.

QIO_RETURN:                             ; Return to caller.
        RSB                             ; Return.

;-----------------------------------------------------------------------
;        SUBROUTINE TO PERFORM COMMON PACKET SET-UP
;-----------------------------------------------------------------------
;
; Set up part of command packet common to ONLINE and READ commands.
; Fill in ring pointers for command response.
;
; Implicit Inputs:
;       RINGAD must be adjusted with either the physical or relative
;       address of the communications area.
;
SETUP_IO:
        MOVL    R7,R9                   ; Set up to zero communications
                                        ; area.
        MOVL    #COMM_END/8,R0          ; Ditto.

CLOOP:  CLRQ    (R9)+                   ; Clear packet storage area.
        SOBGTR  R0,CLOOP
                                        ; NOTE: this leaves R0 = 0.
        MOVW    #(RCMD-RMSG_,B^RMSG+P_LEN(R7); Load length into response.
        MOVW    #36,B^RCMD+P_LEN(R7)    ; Write length of command.
        MOVW    R3,B^RCMD+P_UNIT(R7)    ; Load unit number.
        ADDL3   #OWN!RMSG,RINGAD,(R7)   ; Load resp descriptor with add of
                                        ; response packet and owned
                                        ; by port.
        ADDL3   #OWN!RCMD,RINGAD,B^4(R7) ; Load command descriptor
                                        ; with add of command packet
                                        ; and owned by port.
        RSB

ENDDUBOOT:                              ; Delimit end of code.
.END    STARTDU                         ; Remove for file inclusion.
```

Appendix K
Unexpected Error Conditions

K.1 ID Parity Error Interrupts Following Retry Timeout

A retry timeout error condition occurs if the KA820 module fails to access a VAXBI node after 4096 retries, as specified by the VAXBI design. For example, if a memory node is locked, and the KA820 module tries to perform an IRCI (interlock read with cache intent) to an address on that node, the KA820 module will retry the transaction up to 4096 times, incrementing the retry timer with each try. If a write unlock transaction does not unlock the memory before the retry timer expires, a timeout will occur. In response to this condition, KA820 microcode initiates a machine check and pushes a status word with the event code 1D (hex) on the stack.

Software can evaluate the condition and take appropriate action.

However, as a side effect of the retry timeout, the IPE (ID Parity Error) bit may be set in the Bus Error Register (BER) of each node on the VAXBI bus. Each node that has the HEIE (Hard Error Interrupt Enable) bit set in the BICSR will then interrupt the processor.

Exception condition handling software should be written with this consideration in mind.

K.2 Clearing the Bus Error Register

Error bits in the Bus Error Register (BER) on any node in a VAX 8200 computer system may be set or cleared following power-up. Software should write 3FFF 0007 (hex) to each BER in the system to clear these bits.

K.3 Interrupts Following Initialization

The BR ⟨7:4⟩ interrupt flags in the port controller may be set or cleared following processor initialization. Software should deal with this situation by setting up the system control block (SCB) before lowering the IPL below 17 (hex).

Glossary

This glossary defines terms used to describe the VAXBI bus, the BIIC, and the KA820 module.

ACK data cycle

A data cycle of a read-type or write-type transaction during which the slave asserts the ACK CNF code to acknowledge that no error has been detected and that the cycle is not to be stalled.

AC LO L

A signal that indicates the condition of ac power to the computer system. When AC LO L is false, dc power will remain steady for at least 4.2 milliseconds.

Adapter

A node that interfaces other buses, communication lines, or peripheral devices to the VAXBI bus.

Arbitration cycle

A cycle during which nodes arbitrate for control of the VAXBI bus.

Assert

To cause a signal to take the "true" or asserted state.

Asserted

To be in the "true" state.

Assertion

The transition of a signal from the "false" state to the "true" state.

Atomic

Pertaining to an indivisible operation.

Attached processor

A second or third processor in a multiprocessor system. Attached processors do not have direct access to the control panel, the console terminal, serial-line units, the RCX50 controller, or the watch chip.

Backup translation buffer (BTB)

A small memory on the KA820 module that stores translations (page table entries) for 512 recently used virtual address pages. The backup translation buffer backs up the mini-translation buffer (MTB).

Bandwidth

The data transfer rate measured in information units transferred per unit of time (for example, megabytes per second). All bandwidth figures quoted in this manual take into account command/address and embedded ARB cycle overhead.

BCI

BI chip interface; a synchronous interface bus that provides for all communication between the BIIC and the user interface.

BIIC

Bus interconnect interface chip; a chip that serves as a general purpose interface to the VAXBI bus.

BIIC CSR space

The first 256 bytes of the 8K-byte nodespace, which is allocated to the BIIC's internal registers. See also Nodespace.

BIIC-generated request

A transaction request generated by the BIIC rather than by the user interface. The BIIC can request only error interrupts.

BIIC-generated transaction

A transaction performed solely by the BIIC with no assistance from the master port interface. Only INTR and IPINTR transactions can be independently generated by the BIIC. The user interface initiates BIIC-generated transactions by using the IPINTR/STOP force bit, the user interface or error interrupt force bits, or by asserting one of the BCI INT⟨7:4⟩ L lines. A BIIC-generated transaction can also result from a BIIC-generated request, which results from a bus error that sets a bit in the Bus Error Register.

Boot RAM

An 8K-byte memory on the KA820 module used as temporary storage for boot macrocode.

BTB

See Backup translation buffer.

Bus access latency

The delay from the time a node desires to perform a transaction on the VAXBI bus until it becomes master.

Bus adapter

A node that interfaces the VAXBI bus to another bus.

Busy extension cycle

A bus cycle during which a VAXBI node, not necessarily the master or the slave of a transaction, asserts the BI BSY L line to delay the start of the next transaction.

Cache

A small, high-speed memory on the KA820 module that contains copies of recently accessed physical memory locations.

Cold start

Boot to bring a fresh copy of the operating system into memory.

Command/Address cycle

The first cycle of a VAXBI transaction. The information transmitted in this cycle is used to determine slave selection. In some cases the data on the BI D⟨31:0⟩ L lines is not an actual address, but it serves the same purpose: to select the desired slave node(s). For example, during an INTR command a destination mask is used.

Command confirmation

The response sent by the slave(s) to the bus master to confirm participation in the transaction.

Command confirmation cycle

The third cycle of a VAXBI transaction during which slave(s) confirm participation in the transaction.

Configuration data

Data loaded into the BIIC on power-up that includes the device type and revision code, the parity mode, and the node ID.

Console

The manual control system integrated into the KA820 module in microcode. It lets you start and stop the processor and run diagnostics using a terminal.

Console mode

A condition of the CPU in which the computer is halted and responds only to machine control commands typed on the terminal connected to serial-line unit 0.

Control store

A set of five ROM/RAM chips containing KA820 microcode.

Corrected read data (CRD)

Data sent on the VAXBI bus in response to a read transaction in which the slave device has detected and corrected a data error. The slave device sends a corrected read data (CRD) code on the VAXBI bus when it sends the data.

CRD

See Corrected read data.

Cycle

The basic bus cycle of 200 nanoseconds (nominal), which is the time it takes to transfer the smallest piece of information on the VAXBI bus. A cycle begins at the leading edge of T0/50 and continues until the leading edge of the next T0/50.

Data cycle

A cycle in which the VAXBI data path is dedicated to transferring data (such as read or write data, as opposed to command/address or arbitration information) between the master and slave(s). During read STALL data cycles, the BI D⟨31:0⟩ L and I⟨3:0⟩ L lines contain undefined data. See also ACK data cycle, Read data cycle, STALL data cycle, Vector data cycle, and Write data cycle.

Data transfer transactions

VAXBI transactions that involve the transfer of data as well as command/address information: read-type, write-type, IDENT, and BDCST transactions.

DC LO L

A signal that indicates the condition of dc power to the VAXBI backplane. When DC LO L is false, power to the backplane is steady.

Deassert

To cause a signal to be in the "false" or deasserted state.

Deasserted

To be in the "false" state.

Deassertion

The transition of a signal from the "true" state to the "false" state.

Decoded ID

The node ID expressed as a single bit in a 16-bit field.

Device type

A 16-bit code that identifies the node type. This code is contained in the BIIC's Device Register.

Direct memory access (DMA) adapter

An adapter that directly performs block transfers of data to and from memory.

EEPROM

A 16K-byte electrically erasable programmable read-only memory used on the KA820 module to store customer choices for KA820 options, VAX bootstrap macrocode, and microcode patches.

Embedded arbitration cycle

An arbitration cycle that occurs (is embedded) in a VAXBI transaction.

Encoded ID

The node ID expressed as a 4-bit binary number. The encoded ID is used for the master's ID transmitted during an embedded ARB cycle.

Even parity

The parity line is asserted if the number of asserted lines in the data field is an odd number.

Event flag

A status posting bit maintained by an operating system or the VAX Diagnostic Supervisor, used to pass signals between the operator and software or between programs.

Expansion module

A VAXBI module that does not attach directly to the VAXBI bus. A VAXBI node that requires more than one module has one module that attaches directly to the VAXBI (that is, contains a BIIC) and one or more expansion modules that communicate with the first module over the user-defined I/O section.

Extension cycle

A bus cycle during which a VAXBI transaction is made longer. See STALL data cycle, Busy extension cycle, and Loopback extension cycle.

F chip

One of three chips in the KA820 processor chip set: floating point accelerator.

H

Designates a high-voltage logic level (that is, the logic level closest to Vcc). Contrast with L.

IDENT arbitration cycle

The fourth cycle of an IDENT transaction during which nodes arbitrate to determine which is to send the vector.

I/E chip

One of three chips in the KA820 processor chip set, implementing the instruction buffer, microsequencer, instruction execution unit, and MTB.

Illegal confirmation code

A CNF code that is not permitted in a particular VAXBI cycle (such as a RETRY command confirmation to a multi-responder command).

Interlock commands

The two commands, IRCI (Interlock Read with Cache Intent) and UWMCI (Unlock Write Mask with Cache Intent), that are used to implement atomic read-modify-write operations.

Internode transfer

A VAXBI transaction in which the master and slave(s) are in different VAXBI nodes. Contrast with Intranode transfer.

Interrupt port

Those BCI signals that are used in generating INTR transactions.

Interrupt port interface

That portion of user logic used to interface to the interrupt port of the BIIC.

Interrupt sublevel priority

Interrupt priority information used during an IDENT transaction to determine which node with a pending interrupt is to provide the vector. The interrupt sublevel priority corresponds to the node ID.

Interrupt vector

In VAX/VMS systems, an unsigned binary number used as an offset into the system control block. The system control block entry pointed to by the VAXBI interrupt vector contains the starting address of an interrupt handling routine. (The system control block is defined in the *VAX Architecture Handbook*.)

Intranode transfer

A transaction in which the master and slave are in the same node. Loopback transactions are intranode transfers. Contrast with Internode transfer.

Invalidate

To mark an entry in the cache, MTB, or BTB to show that it no longer contains current information.

L

Designates a low-voltage logic level (that is, the logic level closest to ground). Contrast with H.

Latency

Read access time; see Bus access latency.

Local memory

VAXBI memory that can be accessed without using VAXBI transactions; for example, VAXBI-accessible memory on a single board computer.

Logical console

A CPU that performs console functions for another CPU.

Loopback extension cycle

A cycle of a loopback transaction during which a node asserts both BI BSY L and BI NO ARB L to delay the start of the next transaction.

Loopback request

A request from the master port interface asserted on the BCI RQ⟨1:0⟩ L lines which permits intranode transfers to be performed without using the VAXBI bus.

Loopback transaction

A transaction in which information is transferred within a given node without use of the VAXBI data path. Contrast with VAXBI transaction.

Macrocode

Instructions written in MACRO-32 or a higher level programming language.

Mapped adapter

A DMA adapter that performs data transfers between a system with a contiguous memory space and VAXBI address space (in which memory need not be contiguous). The mapping is done by using a set of map registers located in the adapter.

Master

The node that gains control of the VAXBI bus and initiates a VAXBI or loopback transaction. See also Pending master.

Master port

Those BCI signals used to generate VAXBI or loopback transactions.

Master port interface

That portion of user logic that interfaces to the master port of the BIIC.

Master port request

A request (either VAXBI or loopback) generated by the master port interface through the use of the BCI RQ⟨1:0⟩ L lines.

Master port transaction

Any transaction initiated as a result of a master port request.

M chip

One of three chips in the KA820 processor chip set, used for BTB and cache tag storage, internal processor registers, interrupt handling, memory management, clock generation, serial-line units, and interface to the port controller.

Microcode

Instructions contained in control store that carry out the functions of the VAX instruction set, console functions, and self-test.

Mini-translation buffer (MTB)

A translation buffer that stores translations (page table entries) for five recently used virtual address pages.

Module

A single VAXBI card that attaches to a single VAXBI connector.

MTB

See Mini-translation buffer.

Multi-responder commands

VAXBI commands that allow for more than one responder. These include the INTR, IPINTR, STOP, INVAL, and BDCST commands.

Node

A VAXBI interface that occupies one of sixteen logical locations on a VAXBI bus. A VAXBI node consists of one or more VAXBI modules.

Node ID

A number that identifies a VAXBI node. The source of the node ID is an ID plug attached to the backplane.

Node reset

A sequence that causes an individual node to be initialized; initiated by the setting of the Node Reset bit in the VAXBI Control and Status Register.

Nodespace

An 8K-byte block of I/O addresses that is allocated to each node. Each node has a unique nodespace based on its node ID.

Null cycle

A cycle in which all VAXBI lines are deasserted (that is, no transaction or arbitration is taking place).

Odd parity

The parity line is asserted if the number of asserted lines in the data field is an even number.

Parity mode

Specifies whether parity is generated by the BIIC or by the user interface.

Pending master

A node that has won an arbitration but which has not yet begun a transaction.

Pending request

A request of any type, whether from the master port or a BIIC-generated request, that has not yet resulted in a transaction.

Physical address

A 30-bit integer identifying a byte location in physical memory or a register in an I/O device.

Physical console

Serial-line unit 0 and the related microcode functions available on the primary processor.

Pipeline request

A request from the master port that is asserted prior to the deassertion of BCI RAK L for the present master port transaction; that is, a new request is posted prior to the completion of the previous transaction.

Port controller

An interface that connects the processor section of the KA820 module with the VAXBI interface and the PCI bus devices (EEPROM, boot RAM, watch chip, and RCX50 controller).

Power-down/Power-up sequence

The sequencing of the BI AC LO L and BI DC LO L lines upon the loss and restoration of power to a VAXBI system. See also System reset.

Primary processor

The processor installed in VAXBI slot K1J1. This processor is connected to the control panel, RCX50 controller, watch chip, serial-line units 0-3.

Private memory

Memory that cannot be accessed from the VAXBI bus.

Program I/O mode

The normal condition of the CPU, in which the computer executes software. Characters typed on the console terminal are passed through to the operating system. The console terminal acts like a standard terminal on one of the other serial-line units.

Programmed I/O (PIO) adapter

An adapter that does not access memory on the VAXBI bus but interacts only with a host processor.

RCLK (receive clock)

The clock phase during which information is received from the VAXBI bus; equivalent to T100/150.

RDS

See Read data substitute.

Read data cycle

A data cycle in which data is transmitted from a slave to a master.

Read data substitute (RDS)

Faulty, uncorrectable data sent on the VAXBI bus in response to a read-type transaction. The slave device sends the read data substitute (RDS) code on the VAXBI bus when it sends the bad data.

Read-type commands

Any of the various VAXBI read commands, including READ, RCI (Read with Cache Intent), and IRCI (Interlock Read with Cache Intent).

RESERVED code

A code reserved for use by DIGITAL.

RESERVED field

A field reserved for use by DIGITAL. The node driving the bus must ensure that all VAXBI lines in the RESERVED field are deasserted. Nodes receiving VAXBI data must ignore RESERVED field information. This requirement provides for adding functions to future VAXBI node designs without affecting compatibility with present designs. Example: The BI D⟨31:0⟩ L and BI I⟨3:0⟩ L lines during the third cycle of an INTR transaction are RESERVED fields.

Reset module

In a VAXBI system, the logic that monitors the BI RESET L line and any battery backup voltages and that initiates the system reset sequence.

Resetting node

The node that asserts the BI RESET L line.

Retry state

A state that the BIIC enters upon receipt of a RETRY confirmation code from a slave. If the master reasserts the transaction request, the BIIC resends the transaction without having the user interface provide the transaction information again. The command/address information and the first data longword, if a write transaction, are stored in buffers in the BIIC.

Self-test

A microcoded test that identifies hardware faults on a VAXBI module.

Single-responder commands

VAXBI commands that allow for only one responder. These include read-type and write-type commands and the IDENT command. Although multiple nodes can be selected by an IDENT, only one returns a vector.

Slave

A node that responds to a transaction initiated by a node that has gained control of the VAXBI bus (the master).

Slave port

Those BCI signals used to respond to VAXBI and loopback transactions.

Slave port interface

That portion of user logic that interfaces to the slave port of the BIIC.

STALL data cycle

A data cycle of a read-type or write-type transaction during which the slave asserts the STALL CNF code to delay the transmission of the next data word.

System reset

An emulation of the power-down/power-up sequence that causes all nodes to initialize themselves; initiated by the assertion of the BI RESET L line.

Target bus

The bus that a VAXBI node interfaces to the VAXBI bus.

TCLK (transmit clock)

The clock phase during which information is transmitted on the VAXBI bus; equivalent to T0/50.

Transaction

The execution of a VAXBI command. The term "transaction" includes both VAXBI and loopback transactions.

UNDEFINED field

A field that must be ignored by the receiving node(s). There are no restrictions on the data pattern for the node driving the VAXBI bus. Example: The BI D⟨31:0⟩ L and BI I⟨3:0⟩ L lines during read STALL data cycles and vector STALL data cycles are UNDEFINED fields.

User interface

All node logic exclusive of the BIIC.

User interface CSR space

That portion of each nodespace allocated for user interface registers. The user interface CSR space is the 8K-byte nodespace minus the lowest 256 bytes, which comprise the BIIC CSR space.

User interface request

A transaction request from the user interface, which can take the form of a master port request, an assertion of a BCI INT⟨7:4⟩ L line, or the setting of a force bit.

VAXBI primary interface

The portion of a node that provides the electrical connection to the VAXBI signal lines and implements the VAXBI protocol; for example, the BIIC.

VAXBI request

A request for a VAXBI transaction from the master port interface that is asserted on the BCI RQ⟨1:0⟩ L lines.

VAXBI system

All VAXBI cages, VAXBI modules, reset modules, and power supplies that are required to operate a VAXBI bus. A VAXBI system can be a subsystem of a larger computer system.

VAXBI transaction

A transaction in which information is transmitted on the VAXBI signal lines. Contrast with Loopback transaction.

VAX interrupt priority level (IPL)

In VAX/VMS systems, a number between 0 and 31 that indicates the priority level of an interrupt with 31 being the highest priority. When a processor is executing at a particular level, it accepts only interrupts at a higher level, and on acceptance starts executing at that higher level.

VAX port adapter

In a VAXBI system, an adapter that conforms to the VAX port architecture, uses interlock transactions to access command and response queues in VAXBI memory, and performs virtual-to-physical memory translation by using page tables located in memory on the VAXBI bus.

Vector

An address pointing to a routine that services interrupts or exceptions. The system control block is a table of vectors.

Vector data cycle

A data cycle in which an interrupt vector is transmitted from a slave to a master.

Virtual address

A 32-bit integer identifying a byte location mapped by memory management.

Warm start

Restarting the software at the point where processing stopped in a power failure.

Window adapter

A bus adapter that maps addresses that fall within one contiguous region (a "window") of a bus's address space into addresses in a window (possibly in a different region) in another bus's address space.

Window space

A 256-Kbyte block of I/O addresses allocated to each node based on node ID and used by bus adapters to map VAXBI transactions to other buses.

Write data cycle

A data cycle in which data is transmitted from a master to a slave.

Write-type command

Any of the various VAXBI write commands, including WRITE, WCI (Write with Cache Intent), WMCI (Write Mask with Cache Intent), and UWMCI (Unlock Write Mask with Cache Intent).

Index

Mnemonics and Abbreviations Associated with the KA820 Processor Module EK–KA820–TM

ACCS	Accelerator Control and Status Register
AC LO L	ac power is unsteady when this signal is true
ACK	acknowledge transaction confirmation on the VAXBI bus
AP	Argument Pointer Register; also attached processor
ASTLVL	asynchronous system trap level
ARB	arbitration cycle on the VAXBI bus
BAD	VAXBI signal indicating a failing node
BBU	battery-backup unit
BCI	bus connecting the BIIC with the other circuits on a node
BCICSR	BCI Control and Status Register
BDCST	broadcast transaction
BER	Bus Error Register
BIIC	backplane interconnect interface chip
BINID	VAXBI Node ID Register
BROKE	VAXBICSR bit indicating self-test failure
BTB	backup translation buffer
CAL	11-bit address bus for cache and BTB RAMs
CAM	contents-addressable memory (part of control store)
CMISS	cache miss or BTB miss
CNF	VAXBI transaction confirmation code
CPUCLK	clock signal
CPU	central processor unit
CRD	corrected read data
CSR	a control and status register
DAL	32-bit data and address lines bus
DATI	data-in UNIBUS transaction
DATIP	data-in-pause UNIBUS transaction
DATO	data-out UNIBUS transaction
DATOB	data-out-byte UNIBUS transaction
DC LO L	dc power is unsteady when this signal is true
DTYPE	Device Register
DWBUA	VAXBI to UNIBUS adapter
EBDAN	KA820-specific serial-line unit diagnostic program
EBKAX	KA820-specific cluster exerciser diagnostic program
EBSAA	KA820-specific VAX diagnostic supervisor
EEPROM	electrically erasable programmable read-only memory
EIA	specification of voltage levels used by serial line units
EINTRCSR	Error Interrupt Control Register
ENB APT	enable APT signal
ENB PIPE	enable pipeline signal
EVKAA	hardcore instruction test diagnostic program
EVKAB	basic instruction exerciser diagnostic program
EVKAC	floating-point instruction exerciser diagnostic program
EVKAE	privileged architecture exerciser diagnostic program
F chip	floating-point accelerator chip in the CPU
FP	Frame Pointer Register
FPD	First Part Done flag
GPR	a general purpose register (R0–R15)
HEX	hexadecimal (base 16 notation)
HFSB	Hardware Fault State Bit
ICCS	Interval Clock Control Register
ICR	Interval Count Register
IDENT	the VAXBI transaction used to solicit an interrupt vector
I/E chip	instruction and execution chip in the CPU
IPL	interrupt priority level
IPR	an internal processor register
IRCI	interlock read with cache intent VAXBI transaction
INTR	interrupt VAXBI transaction
INVAL	invalidate VAXBI transaction
IPINTR	interprocessor interrupt VAXBI transaction
K1J1	VAXBI slot assigned to the primary processor
KK810	system control panel assembly
LED	light emitting diode
LP	loopback
M chip	memory interface chip in the CPU
MFPR	move from processor register instruction

MIB	40-bit microinstruction bus
MTB	mini-translation buffer
MTPR	move to processor register instruction
NO ACK	no acknowledgment VAXBI transaction confirmation
NODE ID	a number that identifies a VAXBI node
P0BR	P0 Base Register
P1BR	P1 Base Register
P0LR	P0 Length Register
P1LR	P1 Length Register
PAL	7-bit bus used to address BTB and cache tags within the M chip
PC	Program Counter Register
PCBB	Process Control Block Base Register
PCI	16-bit port controller interface bus
PCM	a VAXBI control module used in the KK810 assembly
PCntl	port controller
PFN	page frame number
PME	performance monitor enable signal
PP	primary processor
PRIM	a power control circuit used on the PCM module
PSL	Processor Status Longword
PTE	page table entry
RAM	random access memory
RCI	read with cache intent VAXBI transaction
RCX50	controller for the RX50 diskette drive
RDS	read data substitute (containing an uncorrectable error)
REI	return from exception or interrupt instruction
ROM	read-only memory
RPB	restart parameter block
RX50	diskette drive
RXCD	Receive Console Data Register (for VAXBI internode communication)
RXCS	Serial-Line Unit 0 Receive Control and Status Register
RXDB	Serial-Line Unit 0 Receive Data Buffer Register
SCB	system control block
SCBB	System Control Block Base Register
SID	System Identification Register
SISR	Software Interrupt Summary Register
SLU	serial-line unit
SP	Stack Pointer Register
STOP	stop VAXBI transaction
TODR	Time of Day Register
TXCS	Serial-Line Unit 0 Transmit Control and Status Register
TXDB	Serial-Line Unit 0 Transmit Data Buffer Register
UART	universal asynchronous receiver transmitter used in SLU
UWMCI	unlock write mask with cache intent VAXBI transaction
VAXBI	32-bit synchronous system bus
VAXBICSR	VAXBI Control and Status Register
VCR	VAX Can't Retry flag
VDS	VAX Diagnostic Supervisor
VMB	primary bootstrap program
WCI	write with cache intent VAXBI transaction
WCSA	first control-store read access register
WCSD	second control-store read access register
WCSL	control-store load register
WMCI	write mask with cache intent VAXBI transaction

Digital Equipment Corporation • Maynard, MA 01754