IBM RT PC Virtual Resource Manager Technical Reference Version 2.1

Virtual Resource Manager Programming Reference

Programming Family

Personal Computer Software

SC23-0816-0

IBM RT PC Virtual Resource Manager Technical Reference Version 2.1

Virtual Resource Manager Programming Reference

Programming Family

Personal Computer Software

First Edition (January 1987)

This edition applies to Version 2.1 of the Virtual Resource Manager, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT Personal Computer dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas, 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1985, 1987

Virtual Resource Manager Programming Reference

Programming Family

IBM

Personal
Computer
Software

First Edition (January 1987)

This edition applies to Version 2.1 of the Virtual Resource Manager, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT Personal Computer dealer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas, 78758. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1985, 1987

Some material reprinted courtesy of Motorola, Inc.

©Copyright Motorola, Inc. 1985

IBM TECHNICAL NEWSLETTER

for the

RT Personal Computer Virtual Resource Manager

Virtual Resource Manager Programming Reference

© Copyright International Business Machines Corporation 1985, 1987
© Copyright Motorola, Inc. 1985

—OVER—

Order Numbers:
79X3823
SN20-9858
June 26, 1987
© Copyright IBM Corp. 1987

TB79X3824
Printed in U.S.A.

Summary of Changes

This technical newsletter provides additional command and device support available with the Virtual Resource Manager.

A change to the text is indicated by a vertical bar to the left of the change.

Perform the following:

Remove Pages	**Insert Update Pages**
Title page	Title page
1-11 and 1-12	1-11 and 1-12
2-7 and 2-8	2-7 and 2-8
2-15 and 2-16	2-15 and 2-16
3-7 to 3-10	3-7 to 3-10
4-65 to 4-70	4-65 to 4-70
5-23 and 5-24	5-23 and 5-24
5-33 to 5-36	5-33 to 5-36
5-141 and 5-142	5-141 and 5-142
7-1 to 7-14	7-1 to 7-24
8-7 and 8-8	8-7 and 8-8
D-21 and D-22	D-21 to D-24

Note: Please file this cover letter at the back of the manual to provide a record of changes.

June 26, 1987

About This Book

Audience and Purpose

This book describes the Virtual Resource Manager (VRM), which is a collection of processes, device drivers, and commands that control and extend hardware functions for an operating system. The VRM shields the operating system from hardware changes and allows more than one operating system (and their applications) to run simultaneously.

This book also defines the Virtual Machine Interface (VMI) to the VRM. The VMI controls how an operating system communicates with the VRM. This information is useful to developers who design or modify operating systems components that run in the virtual machine environment.

This book is intended for systems programmers and developers who need to understand the role of the Virtual Resource Manager in the RT Personal Computer[1]. The reader of this book is expected to have an understanding of hardware and operating systems fundamentals.

How to Use This Book

This book describes the programming environment of the Virtual Resource Manager (VRM), which is a software layer between the hardware and the operating system of the RT PC.

Virtual Resource Manager Device Support describes VRM IPL and configuration, as well as how the VRM supports specific devices and device subsystems.

The VRM communicates with operating systems by way of supervisor call instructions and virtual interrupts sent across the Virtual Machine Interface (VMI). The VMI provides a uniform interface to the various configurations of input/output devices operating below the virtual machine level.

---

1 RT Personal Computer, RT PC, and RT are trademarks of International Business Machines Corporation.

This book contains programming information on the VRM and virtual machine characteristics that will enable you to develop and implement processes, access methods, device drivers and other VRM components.

In addition to the calls that cross the VMI, this book also describes the internal VRM runtime routines and the commands used with the VRM debugger.

The book consists of the following sections:

Chapter 1, "Virtual Resource Manager Concepts" on page 1-1 provides a brief introduction to the VRM and its place in the RT PC system. This chapter provides a high-level overview of the characteristics of the VRM and the Virtual Machine Interface.

Chapter 2, "Virtual Machine Interface Characteristics" on page 2-1 goes into more detail about the VMI and discusses virtual machine control registers, interrupt processing, and accessing virtual memory segments.

Chapter 3, "VRM Programming Environment" on page 3-1 provides a closer look at the functional characteristics of the VRM. This chapter describes the I/O subsystem, naming conventions, and the component management tasks performed by the VRM.

Chapter 4, "System Control Instructions" on page 4-1 describes the instructions that can change the state of the virtual machine. These instructions are **load program status** and **supervisor call**. The supervisor call instructions are listed and defined.

Chapter 5, "Virtual Resource Manager Programming Interfaces" on page 5-1 lists and describes the VRM runtime routines that allow you to tailor the open-coded system to your needs and resources. These routines control everything the VRM does, including input/output control, device, queue and memory management, and error logging and trace capabilities.

Chapter 6, "Managing Minidisks" on page 6-1 explains the functions related to the creation and management of minidisks in the system.

Chapter 7, "Floating-Point Services" on page 7-1 describes the floating-point computation support provided by the VRM for the various floating-point hardware options.

Chapter 8, "Virtual Resource Manager Debugger" on page 8-1 discusses how to use the VRM debugger to locate and correct errors in user-installed code.

Appendix A, "Index of Supervisor Call Instructions" on page A-1 lists the supervisor call instructions by number and indexes them to the appropriate page.

Appendix B, "TOC Object Module Information" on page B-1 provides information on the TOC (table of contents) object module format used by the VRM. This section is useful for developers who are adding code to or debugging code in the VRM.

Appendix C, "Key Sequences for System Functions" on page C-1 lists the IBM-defined keystroke combinations and the system function that results when you enter the particular key sequence. These key sequences pertain only to the standard RT PC keyboard.

Appendix D, "C Language VRM Subroutines" on page D-1 provides the C language equivalent to the VRM internal calls described in Chapter 5, "Virtual Resource Manager Programming Interfaces" on page 5-1. In addition, this appendix defines the external variables you may use with the VRM.

A glossary and index follow these chapters and appendices.

A Reader's Comment Form and Book Evaluation Form are provided at the back of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

Please note the following items regarding hexadecimal notation, reserved fields, the bit padding and numbering conventions, and highlighting conventions used in this book:

- A hexadecimal value as expressed in this publication is preceded by a zero and a lowercase ***x***. For example, the hexadecimal value 'F3' is represented as 0xF3.
- The value of reserved fields input to the Virtual Resource Manager must be set equal to zero. The value of reserved fields returned by the VRM is unpredictable.
- The bit-numbering convention used by IBM has the most significant bit on the left and the least significant bit for a given field on the right. For example, in a 32-bit word, the most significant bit (bit 0) is the leftmost bit and the least significant bit (bit 31) is the rightmost bit.
- High-order bits in a given field not used to express a value are padded with zeroes. For example, if an input parameter sent in a 32-bit register is only a 16-bit value, the value occupies bits 16 through 31 of the register and bits 0 through 15 of the register are padded with zeroes.
- Terms highlighted in bold-faced italics (such as ***example***) are specific to the RT PC and are included in the glossary. Terms highlighted in bold type (such as **example**) are system-generated items, such as commands, file names, and so on. Terms or phrases that appear in monospace type (such as `example`) are examples of what you might see on a display screen or what you must enter into the system to perform a given function.

Related Information

The following RT PC publications provide additional information on topics related to the VRM. Depending on the tasks you want to perform and your experience level, you may want to refer to the following publications:

- *IBM RT PC Installing and Customizing the AIX Operating System* provides step-by-step instructions for installing and customizing the Advanced Interactive Executive[2] Operating System, including how to add or delete devices from the system and how to define device characteristics. This book also explains how to create, delete, or change AIX and non-AIX minidisks.
- *IBM RT PC Installing the Virtual Resource Manager* provides step-by-step instructions for installing the Virtual Resource Manager and shows you how to change the IBM-recommended choices to suit your system needs. (Available as a separate volume only when the Virtual Resource Manager is purchased separately from the AIX Operating System.)
- *IBM RT PC Hardware Technical Reference* is a three-volume set. Volume I describes how the system unit operates, including I/O interfaces, serial ports, memory interfaces, and CPU interface instructions. Volumes II and III describe adapter interfaces for optional devices and communications and include information about IBM Personal Computer family options and the adapters supported by 6151 and 6150. (Available optionally)
- *IBM RT PC Assembler Language Reference* describes the IBM RT PC Assembler Language and the 032 Microprocessor and includes descriptions of syntax and semantics, machine instructions, and pseudo-operations. This book also shows how to link and run Assembler Language programs, including linking to programs written in C language. (Available optionally)
- *IBM RT PC AIX Operating System Programming Tools and Interfaces* describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs. In addition, this book describes the operating system services and how to take advantage of them in a program. This book also includes a diskette that includes programming examples, written in C language, to illustrate using system calls and subroutines in short, working programs. (Available optionally)
- *IBM RT PC Messages Reference* lists messages displayed by the IBM RT PC and explains how to respond to the messages.
- *IBM RT PC AIX Operating System Commands Reference* lists and describes the AIX Operating System commands.

---

[2] Advanced Interactive Executive and AIX are trademarks of International Business Machines Corporation.

- *IBM RT PC C Language Guide and Reference* provides guide information for writing, compiling, and running C language programs and includes reference information about C language data structures, operators, expressions, and statements. (Available optionally)
- *IBM RT PC Problem Determination Guide* provides instructions for running diagnostic routines to locate and identify hardware problems. A problem determination guide for software and three high-capacity (1.2MB) diskettes containing the IBM RT PC diagnostic routines are included.
- *IBM RT PC Keyboard Description and Character Reference* describes the national character and keyboard support for the 101-key, 102-key, and 106-key keyboards, including keyboard position codes, keyboard states, control code points, code sequence processing, and nonspacing character sequences.
- *RT PC VRM/Hardware Quick Reference* contains brief descriptions of the hardware and Virtual Resource Manager. This booklet includes information on command parameters and return codes and hardware and memory layout data.
- *IBM RT PC AIX Operating System Technical Reference* describes the system calls and subroutines that a C programmer uses to write programs for the AIX Operating System. This book also includes information about the AIX file system, special files, file formats, GSL subroutines, and writing device drivers. (Available optionally)

Ordering Additional Copies of This Book

To order additional copies of this book (without program diskettes), use either of the following sources:

- To order from your IBM representative, use Order Number SBOF-0136.
- To order from your IBM dealer, use Part Number 79X3822.

Two binders and the *Virtual Resource Manager Technical Reference* are included with the order. For information on ordering a binder, books, or the *RT PC VRM/Hardware Quick Reference* separately, contact your IBM representative or your IBM dealer.

Contents

Chapter 1. Virtual Resource Manager Concepts 1-1
About This Chapter 1-3
Understanding the VRM 1-4
Understanding the VMI 1-5
Processor and Virtual Machine States 1-9
Virtual Memory Management 1-11
Input/Output Subsystem 1-14
Virtual Machine Communications 1-17

Chapter 2. Virtual Machine Interface Characteristics 2-1
About This Chapter 2-3
VMI Components and Characteristics 2-4
Virtual Machine Interrupts 2-14
Access to Segments 2-26

Chapter 3. VRM Programming Environment 3-1
About This Chapter 3-3
VRM Internal Characteristics 3-4
Device Management 3-8
Process Management 3-13
Queue Management 3-18
Memory Management 3-20
Semaphore Management 3-28
Timer Management 3-29
Program Management 3-30
Minidisk Management 3-31

Chapter 4. System Control Instructions 4-1
About This Chapter 4-5
Load Program Status Instruction 4-6
Supervisor Call Instructions 4-7
Execution Control SVCs 4-9
Memory Management SVCs 4-23
Input/Output SVCs 4-46
Virtual Machine Communications SVCs 4-72
Machine Control SVCs 4-76
NVRAM Control SVCs 4-88

Chapter 5. Virtual Resource Manager Programming Interfaces 5-1
About This Chapter 5-5
Process Management 5-6
Queue Management 5-16
Memory Management 5-42
Semaphore Management 5-62
Timer Management 5-67
Program Management 5-74
Virtual Machine Control Procedures 5-82
Input/Output Procedures 5-86
Minidisk Management 5-102
Device Management 5-106
VRM Trace and Error Process Interfaces 5-119
Event Monitoring 5-128

Chapter 6. Managing Minidisks 6-1
About This Chapter 6-3
Minidisk Manager Operations 6-4
Data Access Operations 6-25
Query Operations 6-27

Chapter 7. Floating-Point Services 7-1
About This Chapter 7-3
VRM Floating-Point Support 7-4

Chapter 8. Virtual Resource Manager Debugger 8-1
About this Chapter 8-5
Debugging Code in the VRM 8-6
Debugger Programming Interfaces 8-6
Rules for Entering Commands 8-9
Debugger Commands 8-13

Appendix A. Index of Supervisor Call Instructions A-1

Appendix B. TOC Object Module Information B-1
TOC Object Module Format B-1
TOC Object Module Definition B-2
Subroutine Linkage Conventions B-8

Appendix C. Key Sequences for System Functions C-1

Appendix D. C Language VRM Subroutines D-1
External Variables D-21

Glossary **X-1**

Index **X-13**

Figures

1-1. RT PC System Structure 1-4
1-2. RT PC Architecture Model 1-7
1-3. The VMI and Multiple Virtual Machines 1-8
1-4. Processor and Virtual Machine States 1-11
1-5. Virtual Memory Units 1-12
1-6. Virtual Memory Addressing 1-14
2-1. Memory Locations Used as Virtual Registers 2-5
2-2. Virtual Timer Status Register 2-12
2-3. Virtual Machine Program Status Block Locations 2-16
2-4. Virtual Machine Program Status Block 2-17
2-5. Status Flags for Execution-Level Interrupts 2-19
3-1. Virtual Resource Manager Structure 3-4
3-2. VRM I/O Subsystem 3-5
3-3. Virtual Resource Manager Elements 3-6
3-4. RT PC Hardware Structure with Standard Processor Card 3-8
3-5. RT PC Hardware Structure with Advanced Processor Card 3-9
3-6. Exception Control Register Format 3-11
3-7. VRM Process States 3-14
3-8. Rules for Process State Change 3-15
3-9. VRM Real Memory Mapping 3-21
3-10. Segment Register Bit Map 3-22
3-11. VRM Virtual Memory Map/Segment Register Conventions 3-23
3-12. Segment Register 15 Mapping 3-24
3-13. Minidisk Map 3-31
4-1. SVC Types 4-8
4-2. Define Device Structure — Header 4-52
4-3. Define Device Structure — Hardware Characteristics 4-54
4-4. Internal Device Type 4-55
4-5. DMA Type 4-55
4-6. Interrupt Type 4-56
4-7. Operation Completion Information 4-59
4-8. Query Device Structure (IODN = 0) 4-60
4-9. Send Command Program Status Block 4-66
4-10. Command Control Block (CCB) 4-68
4-11. Start I/O Program Status Block 4-71
4-12. IPL Record Format 4-80
4-13. Virtual Machine Program Status Block from IPL 4-81
4-14. Structure Returned from "Query VM SVC" 4-84
4-15. Layout of NVRAM 4-88
5-1. Entry Point Array Structure for _change 5-9

5-2. Signal Mask 5-14
5-3. Acknowledge Parameters 5-18
5-4. Start I/O Acknowledgement Queue Element 5-26
5-5. Send Command Acknowledgement Queue Element 5-26
5-6. General Purpose Acknowledgement Queue Element 5-27
5-7. Send Command Queue Element Format 5-33
5-8. Start I/O Queue Element Format 5-33
5-9. General Purpose Queue Element Format 5-33
5-10. Query Path Structure 5-38
5-11. Symbols List for _bind 5-76
5-12. Structure Returned from _queryv 5-83
5-13. VRM Ring Queue 5-92
5-14. DMA Type Field 5-98
5-15. Query Device Structure 5-116
5-16. Send Command Queue Element for errrecvr 5-120
5-17. Acknowledgement Queue Element for errrecvr 5-121
5-18. Error Entry Acknowledgement Queue Element for errrecvr 5-122
5-19. Send Command Queue Element for trace 5-123
5-20. Acknowledgement Queue Element for trace 5-124
5-21. Send Command Queue Element for trace 5-125
5-22. Acknowledgement Queue Element for trace 5-126
5-23. Acknowledgement Queue Element for trace 5-127
5-24. Hardware Error Entry Structure 5-130
5-25. Non-Hardware Error Entry Structure 5-131
5-26. Generic Trace Entry Structure 5-133
5-27. Trace Entry Structure 5-135
5-28. Trace Entry Format 5-138
5-29. Tables Used to Identify Data Structures to be Dumped 5-140
6-1. 'Query Minidisk' Structure 6-20
6-2. 'Query All Minidisks' Structure 6-23
6-3. Minidisk Command Header Format 6-25
7-1. PSB Data Word 1 for Floating-Point Exceptions 7-11
8-1. Screen produced by the Help command 8-14
8-2. Breakpoint Display 8-19
8-3. CTldsp Command Selection Menu 8-22
8-4. Semaphore Control Block Display 8-23
8-5. Timer Request Block Display 8-24
8-6. Process Control Block Display 8-26
8-7. SLIH Control Block Display 8-28
8-8. SLIH Chain Display 8-29
8-9. Module Table Entry Display 8-30
8-10. Device Table Entry 8-31
8-11. Generic Control Block Display 8-32
8-12. Prompt for Queue Element Selection 8-33
8-13. Response Queue Element 8-34
8-14. Send Command Queue Element 8-35

8-15. Start I/O Queue Element 8-36
8-16. General Purpose Queue Element 8-37
8-17. Detach Queue Element 8-37
8-18. Utilization Information Display 8-38
8-19. Queue Control Block Display 8-39
8-20. Path Information Display 8-41
8-21. Device Extension for Path Information Display 8-42
8-22. Timer Information Display 8-43
8-23. Virtual Machine Display 8-44
8-24. Segment Information Display 8-45
8-25. Virtual Page Information Display 8-46
8-26. Find Display (Argument Found) 8-52
8-27. Display from the Ior Command 8-54
8-28. Example of a System Module Map 8-57
8-29. Format of a Screen Display 8-65
8-30. SRegs Display 8-69
8-31. Translation Lookaside Buffer Display 8-80
8-32. Trace Entry Display Format 8-82
8-33. User-defined Trace Entry Display 8-83
8-34. Vars Display 8-84
8-35. Xlate Display 8-85
B-1. TOC Object Module Structure B-2
B-2. TOC Object Module Header B-3
B-3. Format of an ESD Entry B-6
B-4. Format of an RLD Entry B-7
B-5. Contents of a Stack Frame B-9

Chapter 1. Virtual Resource Manager Concepts

CONTENTS

About This Chapter 1-3
Understanding the VRM 1-4
Understanding the VMI 1-5
  Virtual Machine Architecture 1-6
  Real and Virtual Machine Concepts 1-8
  System Integrity and the Virtual Machine 1-9
Processor and Virtual Machine States 1-9
Virtual Memory Management 1-11
Input/Output Subsystem 1-14
  Code Definition 1-15
  Device Definition 1-15
  Device Attach/Detach 1-16
  Input/Output Procedures 1-16
  Device Query 1-17
Virtual Machine Communications 1-17

About This Chapter

This chapter provides a general description of the Virtual Resource Manager (VRM), its primary components and characteristics, and its relationship to other parts of the system. This chapter also introduces the Virtual Machine Interface, which is the VRM's well-defined, uniform interface to operating systems.

Understanding the VRM

The **Virtual Resource Manager** (VRM) is a collection of processes, device drivers and runtime routines that extend and control hardware functions for an operating system. The VRM shields the operating system from hardware changes and allows more than one operating system (and their applications) to execute. To an operating system, the VRM is perceived as hardware.

The following figure shows where the VRM fits in the RT PC system.

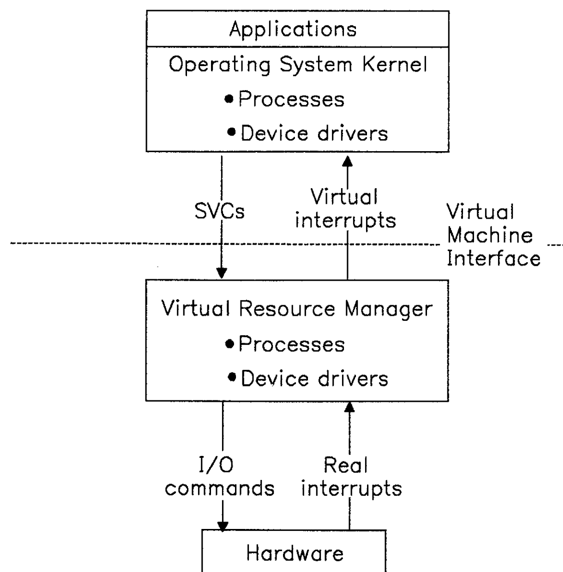


Figure 1-1. RT PC System Structure

The components of the VRM are shown in more detail in Figure 3-1 on page 3-4. The layout of the hardware components is shown in more detail in Figure 3-4 on page 3-8.

Ordinarily, an operating system device driver sends commands directly to a device, and the device returns real interrupts to the operating system in response to these commands. In the RT PC system, however, the VRM receives the operating system commands, called supervisor calls (SVC), and routes them to the appropriate device. On return, the VRM takes the real hardware interrupts and issues virtual interrupts (if necessary) to the operating system. The relationship of real-to-virtual interrupts is seldom one-for-one. However, the command and interrupt structures of the VRM simulate the command and interrupt structures used by an operating system.

The primary components of the VRM are processes and device drivers. The function of VRM processes and device drivers is analogous to the function of operating system processes and device drivers.

Processes receive a portion of the processor's time for program execution and are typically used to control virtual resources. Process types include device manager and protocol processes. ***Device managers*** control virtualized devices, such as virtual terminals. Protocol processes (also known as protocol procedures) handle specific types of functions for virtual devices. All virtual machines are represented as processes in the VRM. Processes allow asynchronous execution of multiple operations. A dispatcher prioritizes and manages the execution of processes.

Device drivers are a collection of subroutines that control the interface between the I/O device adapters and the processor. The most significant difference between VRM and operating system components is in the level of device drivers. The device driver implementation used by the AIX Operating System places the burden of function on the VRM device management facilities (drivers, managers, and protocols). AIX Operating System device drivers are high-level routers of commands that provide limited flexibility. The AIX Operating System has one device driver for the printer, one for the fixed disk, one for diskette, one for a display, one for streaming tape, and so on.

The VRM provides a lower level layer of hardware management. This layer includes the device drivers, device managers (if necessary) and protocol processes that allow for increased flexibility.

It is VRM hardware management, for example, that provides ***virtual devices***. A virtual device is a functionally complete simulation of a physical device. For example, the VRM supports up to 32 virtual terminals that work off a single instance of terminal hardware. Only one virtual terminal is active at a time, however.

While the VRM interface to hardware is as dynamic as the hardware that can be configured on the RT PC, the VRM interface to operating systems (also called ***virtual machines***) is a well-defined, uniform interface. This ***Virtual Machine Interface*** (VMI) not only shields the operating system from hardware changes, but it allows concurrent virtual machines to run on the system.

Understanding the VMI

The VMI is a software interface between operating systems and the VRM. The VMI presents a fixed interface to operating systems. This interface consists of supervisor call instructions (see "Supervisor Call Instructions" on page 4-7 for definitions of all the SVCs) and virtual interrupts. Changes in hardware configurations or VRM programs, therefore, do not necessarily require an operating system change. The VMI also supports the existence of several virtual machines from a single real machine.

Virtual Machine Architecture

A virtual machine is, by definition, a simulation of a physical machine and its related devices. Therefore, the operations and components of a virtual machine closely parallel the operations and components of a physical machine.

A virtual machine typically runs on the normal execution level (level 7) until it completes its work, is interrupted, or is preempted.

Interrupts are grouped and prioritized by levels determined by the source or cause of the interrupt. An interrupt handler can be assigned to each level or sublevel and executes when an interrupt on the corresponding level or sublevel is received.

Now that a processing model is defined, the similarities and differences between physical and virtual machine components can be described. Figure 1-2 on page 1-7 shows some of the similarities and differences between virtual and physical machines. When a process executes, data manipulation and computation are done in the general purpose registers of the processor. System control registers (SCRs) keep track of such facilities as the processor, timer, and interrupts.

For example, the instruction address register (IAR) contains the address of the next instruction to be executed. The interrupt control status register (ICS) contains data that indicates pending and masked interrupts and machine state. The condition status register (CS) contains information about the results of certain executed instructions.

The virtual machine counterparts to the system control registers are the virtual machine control registers (VMCR). Each virtual machine maintains a set of VMCRs to save and restore its state and pertinent data as it and other virtual machines are dispatched.

For more details on SCRs, see *IBM RT PC Hardware Technical Reference.* VMCRs are discussed in "Virtual Machine Control Registers" on page 2-4 of this book.

When an interrupt occurs, a set of interrupt handlers determine the cause of the interrupt and perform interrupt-specific processing. Interrupt handling involves saving the status of the machine at the time of the interrupt and determining the address of that interrupt level's interrupt handling routine. This is accomplished with the use of program status words in the physical machine and program status blocks in the virtual machine. For both physical and virtual machines, then, the interrupt level points to a program status word or block, which in turn directs the machine to the address of the appropriate interrupt handling routine. The sequence of events is much the same for physical and virtual machines.

For more information on program status words, see *IBM RT PC Hardware Technical Reference.* Program status blocks are defined in "Virtual Machine Program Status Blocks" on page 2-15 of this book.

As shown in Figure 1-2 on page 1-7, most virtual machines include an operating system and one or more applications. Multiple virtual machines may have different operating systems or multiple copies of one operating system.

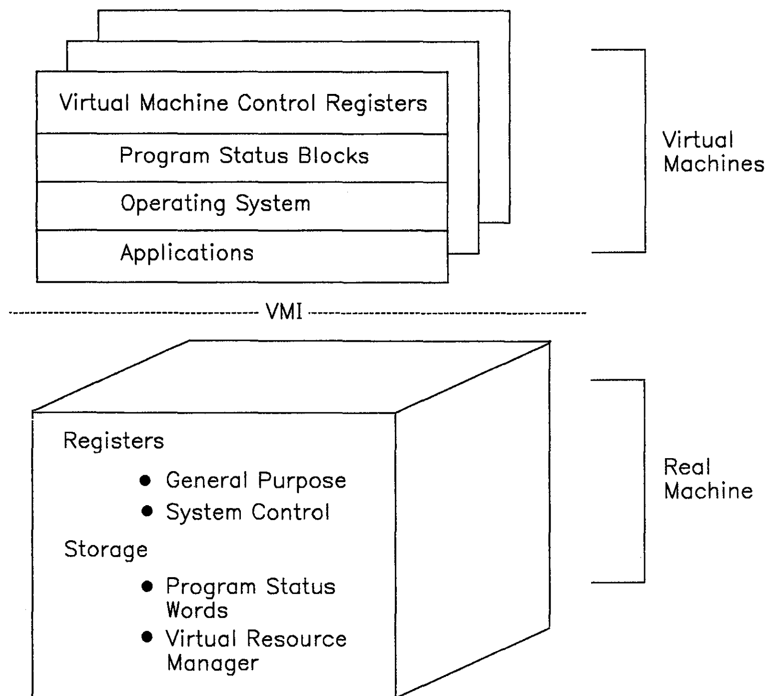


Figure 1-2. RT PC Architecture Model

When a virtual machine is successfully IPLed, the initial machine state is defined as follows:

- The virtual machine is established as VRM process and is assigned a virtual machine identifier (VMID).
- The virtual machine is initialized in operating system state on the base execution level (level 7).
- Page 0 of the IPL segment (segment 0) is pinned.
- The virtual machine's segment 0 is created and loaded so no problem state tasks have access to it.
- The time of IPL is set and the timer is initially disabled.
- All level 0-6 and machine communications interrupts are masked (VICS bit 31 = 1).

For more information on the virtual machine state immediately after IPL, see "IPL Virtual Machine SVC" on page 4-78.

Real and Virtual Machine Concepts

The real machine is the actual hardware components that are connected to make up a system. A typical system usually includes a processor, keyboard, display device, diskette drive, and fixed disk. Other hardware components, such as a plotter, printer, tape drive, or communications device may be added to extend system function. All these devices make up the real machine.

A virtual machine is a functional simulation of a computer and its related devices. Because this functionally equivalent machine is simulated to you, and does not really exist, it is called a virtual machine.

Although virtual machines are logically independent of the physical machine and (in the case of multiple virtual machines) of each other, physical resources must be shared by the users. Physical resources which may be designated as shared among virtual machines include segments of storage, input/output devices such as disks, diskettes, displays, and so on.

Figure 1-3 shows how the VMI presents multiple virtual machines to the VRM and system hardware.

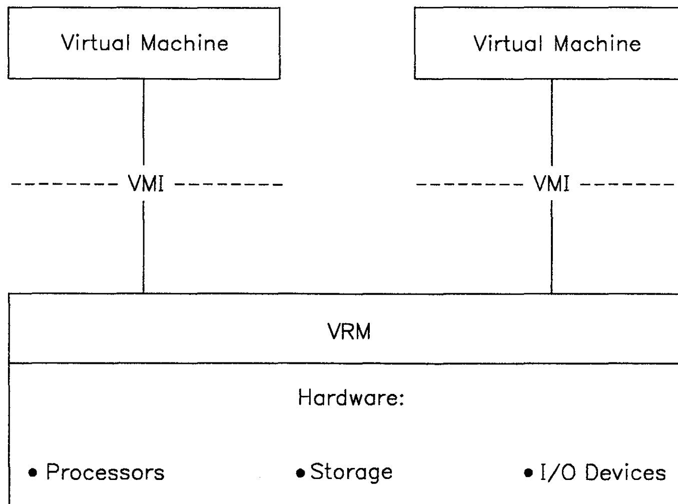


Figure 1-3. The VMI and Multiple Virtual Machines

The principal advantages of the VMI are:

- The VMI allows concurrent execution of multiple operating systems.
- A virtual machine as defined by the VMI has a high-level but physical, machine-like interface. For this reason, the VMI provides more function than the individual hardware components.
- The VMI insulates operating systems from changes in the hardware configuration. To an operating system, the VMI is perceived as hardware. Changes below the VMI level do not necessarily require any change in the operating system.
- The removal of physical constraints gives you the impression of multiple (in the case of virtual terminals) or expanded (in the case of virtual memory) resources.

System Integrity and the Virtual Machine

The VMI isolates virtual machines from each other and from the VRM. However, you can install code into the VRM. If the code is added incorrectly, you can compromise system integrity. Certain limitations are imposed on virtual machine functions. Those limitations include:

- All virtual machines execute in the processor's unprivileged state. Virtual machines have no access to privileged instructions.
- Hardware devices (with the exception of floating-point hardware) are typically accessible only in privileged state. Virtual machines can access bus I/O devices only by setting a bit in a virtual register (see "Virtual Interrupt Control Status (VICS) Register" on page 2-8).

The following section provides more details on privileged and unprivileged instructions, as well as hardware and virtual machine states.

Processor and Virtual Machine States

At the hardware level, the processor can be in one of two execution states, privileged or unprivileged. The execution state determines which instructions the processor allows. In the unprivileged state, the processor executes only unprivileged instructions. Application programs are typically made up of these unprivileged instructions. Privileged instructions execute only when the processor is in the privileged state. Privileged instructions are restricted-use instructions which help control the operating environment of the system. *IBM RT PC Hardware Technical Reference* lists and describes the entire RT PC instruction set.

As stated, the processor operates on two execution levels, privileged and unprivileged. The virtual machine also executes on two levels. The execution states of the virtual machine are:

- Problem state — The state that most applications run in.
- Operating system (OS) state — The state that operating systems run in.

A bit mask in a "virtual register" keeps track of the virtual machine state. A virtual register is a dedicated word of virtual memory that describes the characteristics of the virtual machine. The virtual interrupt control status register (VICS) describes (among other things) the state of the virtual machine. Bit 29 of the VICS determines the virtual machine state. If bit 29 is 1, the virtual machine is in problem state. If bit 29 is 0, the virtual machine is in operating system state. For more information on the VICS, refer to "Virtual Interrupt Control Status (VICS) Register" on page 2-8.

Both problem and operating system state of the virtual machine run in the processor's unprivileged state. When the VRM is running, the processor is in privileged state.

Two instructions, the **supervisor call instruction** (SVC) and the **load program status instruction** (lps) can change the virtual machine's execution state. The SVC instruction can change the execution state from problem to operating system. Each SVC has a function code associated with it. The VRM recognizes the SVC and determines from the function code what actions to take.

SVCs with function code <32,768 (0x7FFF and below) are operating system (OS) SVCs. The VRM sends OS SVCs to the operating system for execution of the specified function. SVCs with function code >32,767 (0x8000 and above) are VRM SVCs directed to the VRM. Not all 32,768 function codes possible for execution by the VRM are supported. Only a few dozen codes represent defined SVC functions. However, all 32K of function codes are reserved for VRM SVC use. Figure 1-4 summarizes processor and virtual machine states and describes VRM and OS handling of SVCs.

Virtual Machine States

	Problem	OS	VRM
VRM SVCs (Code >32,767)	VRM sends an exception to the operating system *	VRM executes	Error in VRM
OS SVCs (Code <32,768)	VRM relays to operating system for execution	VRM relays to operating system for execution	Error in VRM
Hardware State	Unprivileged		Privileged

* Except for Return SVC and Machine Identification SVC

Figure 1-4. Processor and Virtual Machine States

The other instruction that can change the execution state of the virtual machine is the **lps** instruction.

The **lps** instruction is a privileged hardware instruction. Ordinarily, a program check (privileged instruction exception) is generated when an **lps** is issued from problem state. However, when a virtual machine is in operating system state and an **lps** instruction is detected, the VRM emulates the **lps** for the virtual machine. Therefore, the virtual machine must be in operating system state when you issue an **lps** instruction.

This "virtual" **lps** instruction can change the level of interrupt processing, as well as the execution state of the virtual machine.

"Supervisor Call Instructions" on page 4-7 describes the format and variations of the SVCs handled by the VRM. See "Load Program Status Instruction" on page 4-6 for more information, including the format, of the **lps** instruction.

Virtual Memory Management

This section describes the mechanisms and strategy used by the system to provide and direct memory access in the virtual machine environment. The addressing mechanism in the VRM has the following goals:

- To provide virtual memory to the virtual machines, relieving the operating systems in those machines of handling page faults and related mechanics. (If the virtual machine so requests, it can be notified by virtual interrupts when a page fault requiring I/O occurs and when the page fault is cleared. See "Set Interrupt SVC" on page 4-18.)

- To shield the virtual machine operating system from changes in the hardware supporting paging.
- To efficiently use the addressing hardware.

The RT PC system supports an extremely large virtual memory. Total virtual memory addressability is 2^{35} bytes. Each virtual machine can address up to 2^{32} bytes (16 segments times 2^{28} bytes per segment) of virtual memory at a time.

The VRM supports the creation of up to 1024 segments. These resources are shared by the VRM and all virtual machines.

In addition, the VRM reserves two segment registers (14 and 15) for its own use.

This extensive virtual memory must be divided into smaller units for manageability. The three memory units (from largest to smallest) are:

1. Segments
2. Pages
3. Bytes.

The relationship between the memory units is shown in Figure 1-5.

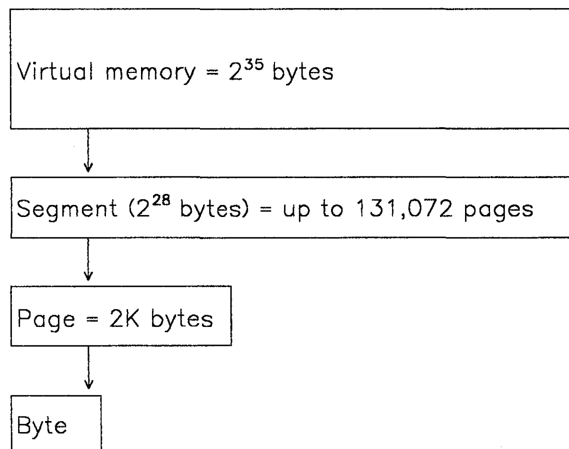


Figure 1-5. Virtual Memory Units

A segment is designed to contain up to 131,072 pages, and a page consists of 2K bytes. Sixteen segment registers allow you to access the various segments. Virtual machines may have many segments, but can simultaneously access only 16 segments at a time. Therefore, 2^{32} is the maximum number of bytes accessible with any given segment register configuration. By changing the contents of the segment registers, all 2^{35} bytes can be accessed. The VRM further restricts this access, as discussed in "Access to Segments" on page 2-26.

A segment is created by the "Create Segment SVC" on page 4-27. The size and protection characteristics are specified by the caller, and a 12-bit segment identifier is returned.

Protection can be specified on the page level. The segment exists until:

- An SVC instruction explicitly destroys the segment.
- The work station is powered off.
- The virtual machine that created the segment is terminated.

  For segments shared by virtual machines, the segment exists until the virtual machine that created the segment with **Create Segment SVC** is terminated.

The ID returned when a segment is created not only uniquely identifies the segment, but also helps determine the segment's virtual address.

Machine instructions generate 32-bit effective addresses, with the four high-order bits specifying a segment register and the 28 low-order bits providing a displacement within the segment. The system provides 16 segment registers, but two of the 16 are always dedicated to input/output operations and cannot be explicitly changed. The VRM can also access registers 0-13, but saves and restores the register contents after use.

The four high-order bits of the effective address specify the segment register, and the segment register contains the 12-bit segment ID. The 12-bit segment ID plus the 28 low-order bits of the effective address yield the segment's 40-bit virtual address. Figure 1-6 on page 1-14 shows how the effective address and segment registers provide the virtual address.

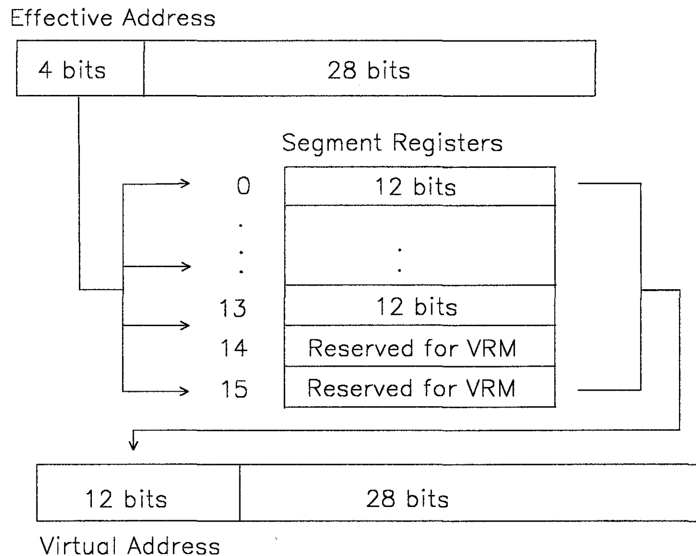


Figure 1-6. Virtual Memory Addressing. The four high-order bits of the effective address specify one of 16 (0-15) segment registers. The segment ID plus the 28-bit displacement of the effective address yield the segment's 40-bit virtual address.

All VMI references to the segment must use the segment ID, either directly or by specifying the segment register containing the segment ID.

Input/Output Subsystem

The input/output subsystem (IOS) supports a high-level functional interface for input/output operations that is different from the processor's I/O read and write instructions. The subsystem provides the mechanisms for I/O device management, manipulation, and data transfer. Each I/O operation is initiated from the operating system state and consists of queueing a work request (a command or a set of commands) to an IOS component.

The IOS is intentionally defined as a queue-driven subsystem that allows work requests to be enqueued to IOS components. The IOS definition makes no statement as to where the actual queue manipulation takes place. In some implementations of the VRM, the manipulation may occur under program control. On other VRM implementations, the same function may occur in an advanced I/O channel. The actual mechanism used by the I/O subsystem is transparent to the subsystem user with the exception of performance characteristics.

The IOS is defined so that most device types may be installed into the system and a set of IOS components not previously part of the VRM may be installed into the VRM to communicate with the device. These IOS components become an extension of the VRM, may run in privileged mode, and may have full access to the VRM facilities. The VRM is dependent on the virtual machine for controlling installation of these IOS components since the potential for compromising the integrity of the system exists.

The interface to the IOS is through a fixed set of IOS SVC handlers that manage devices and control interaction with devices. The virtual machine specifies an ***input/output device number*** (IODN) with each SVC. The IODN uniquely identifies the IOS component that is to service the request. The IOS services execute synchronously to the user's virtual machine, although they may enqueue a command for asynchronous processing. For asynchronous requests, the IOS component signals completion of the request by issuing a virtual interrupt.

Code Definition

The "Define Code SVC" on page 4-49 adds IOS components to the VRM. The **Define Code SVC** allows you to add VRM code without having to rebuild the entire VRM. Each component is given a 16-bit ***input/output code number*** (IOCN) when it is defined.

Device Definition

The virtual machine must use the **Define Code SVC** to install the device code and then use the **Define Device SVC** to start each device that is not part of the standard VRM program product. This process must be repeated at least at every IPL and may occur as often as required by the virtual machine. Devices are identified by IODNs specified when the devices are initially defined. VMI references are always based on this 16-bit IODN. Once a device is defined to the VRM by the **Define Device SVC**, the IODN and its associated IOCN become an extension of the VRM. From the perspective of the VMI, there is no difference between a nucleus device and a third-party installed device.

Certain devices can be addressed directly by issuing problem state instructions and bypassing the IOS. IBM warns you that this technique should be used only with great caution. If the device that is being addressed is capable of generating processor interrupts, then these devices must be made known to the IOS. Processor interrupts are generated in privileged mode and only the VRM can field and directly handle hardware interrupts.

Warning: Use extreme care when dealing with VRM-generated interrupts. If the VRM receives hardware interrupts that it cannot reset, you may jeopardize system integrity and generate an error fatal to the entire work station.

Device Attach/Detach

A virtual machine uses the **Attach Device SVC** to gain access to a device and establish protocols with the VRM with respect to that device. In particular, the virtual machine indicates the interrupt level and sublevel to be used by the VRM when it needs to interrupt the virtual machine. The **Detach Device SVC** is used to terminate the linkage between the virtual machine and the device. Unless a virtual machine is attached to a device, the virtual machine can neither initiate I/O operations nor receive interrupts from that device.

Under certain circumstances, a virtual machine may issue instructions directly to a hardware device; however, that virtual machine never receives any interrupts back from the device unless the virtual machine has properly attached the device.

Input/Output Procedures

After a virtual machine attaches to a device, true I/O commands (for positioning, device control or transfer of data) are allowed. An I/O command consists of initiation, execution, and notification phases. A virtual machine initiates an I/O command with the **Start I/O SVC** or **Send Command SVC** The VRM executes the I/O command when necessary. Optionally, the virtual machine is notified upon the completion of the I/O command by a virtual interrupt from the VRM.

The following I/O procedures are supported:

- I/O Initiation

  The virtual machine specifies the device and controls the execution and flow of the I/O operation in two ways, depending on the supervisor call. For the **Start I/O SVC**, the virtual machine provides the pertinent information in a command control block (CCB); for the **Send Command SVC**, parameters are passed in general purpose registers.

- I/O Execution

  When an I/O command becomes the next command for the device, the VRM begins the execution phase. During this phase, the VRM performs the function requested by the command or issues the appropriate device-level commands to cause control or transfer. Execution of a command element cannot be cancelled once it has begun.

- I/O Notification

  When all command elements for an I/O command have been executed, the VRM then interrogates the interrupt options associated with that I/O command. If an interrupt was requested, then an interrupt is queued for presentation to the virtual machine.

- I/O Cancellation

  The **Cancel I/O SVC** purges queued I/O commands and command elements. Facilities include cancelling commands queued by the virtual machine to a device and cancelling

interrupts queued to the virtual machine from a device. Commands in progress are not cancelled.

Device Query

Information about a device or I/O commands associated with a device may be obtained from the **Query Device SVC**. If the **Query Device SVC** specifies a device with an IODN of 0, the query is interpreted as asking for all defined IODNs.

Virtual Machine Communications

Virtual machines can communicate with messages. Messages are not explicitly structured, and are divided into two types. Immediate messages are short enough (at most 96 bits) to be transmitted in the general purpose registers reserved for SVC parameters. Address messages consist of 40-bit virtual addresses, a length attribute and a type attribute. An address message typically points to part of a segment, but may effectively point to an entire segment. In either case, the recipient understands the message format.

Message sending is asynchronous in that the machine issuing a send command does not wait for the receiver to receive a message. Two priorities of messages are provided, normal and emergency.

A message sent to a virtual machine causes an interrupt to be presented to that machine (when the corresponding level is not masked). The virtual machine specifies the levels and sublevels to be used for normal and emergency messages.

Chapter 2. Virtual Machine Interface Characteristics

CONTENTS

About This Chapter 2-3
VMI Components and Characteristics 2-4
  Virtual Machine Control Registers 2-4
Virtual Machine Interrupts 2-14
  Interrupt and Execution Priority Levels 2-14
  Virtual Machine Program Status Blocks 2-15
  PSB Status Word 2-19
  The Interrupt Processing Cycle 2-24
Access to Segments 2-26
  Segment Register 14 Usage 2-26
  Segment Register 15 Usage 2-26
  Protection of Segments 2-27
  Mapped Page Ranges 2-28
  Segment Sharing Between Virtual Machines 2-29

About This Chapter

This chapter provides a general description of the Virtual Machine Interface (VMI), its primary components and characteristics, and its relationship to other parts of the system.

VMI Components and Characteristics

The VMI consists of:

- The problem state instruction set. Hardware executes these instructions directly. For a detailed discussion of this instruction set, refer to *IBM RT PC Hardware Technical Reference.*
- A simulated privileged machine structure and a set of privileged machine functions. Operating systems running in a virtual machine use these functions to control the state of the virtual machine. The privileged machine structure includes status registers, timers, interrupt handlers, input/output functions, and simulated privileged operations such as the **load program status** instruction.
- A paged virtual memory system. Paging involves copying virtual memory into or out of real memory. A page consists of 2,048 bytes of memory.
- Device-independent interfaces to the display and other input devices.
- Functions for multiple virtual machine management. These functions, which provide for fully independent virtual machines, include initiation, termination, communication between machines, and resource sharing.

Virtual Machine Control Registers

The processor can access a set of 32-bit hardware registers for general purpose operations and system control registers for controlling the processor state. A similar set of "registers" is provided to each virtual machine. These "virtual" registers are actually reserved virtual memory locations. Because each virtual machine has a distinct virtual memory space starting at location 0, these locations are valid for all virtual machines on the system.

Reserved words should not be used by virtual machines. These words will support added functions.

Refer to Figure 2-1 on page 2-5 for a summary of dedicated virtual memory locations and the values associated with each location.

Memory location		Bytes	Value
Decimal	Hex	Hex	
0	0	B8	Reserved
184	B8	4	Number of free paging disk slots
188	BC	4	Number of page replacement cycles
192	C0	4	Number of page faults with preemption
196	4	4	Number of non-paging disk I/Os
200	C8	2	Reserved
202	CA	2	PCB segment ID
204	CC	1	Floating-point hardware (if any)
205	CD	1	Processor type
206	CE	2	Trace buffer synch flag
208	D0	2	VRM sequence number
210	D2	2	Virtual machine sequence number
212	D4	2	Interrupt request buffer
214	D6	1	Process priority
215	D7	1	Floating-point register set
216	D8	4	Bus I/O base address
220	DC	4	Bus memory base address
224	E0	4	Virtual interrupt control status
228	E4	2	Execution level
230	E6	2	Virtual machine ID (right justified)
232	E8	4	Time of day, extended
236	EC	4	Virtual timer status
240	F0	4	Real time of day
244	F4	4	Real time of IPL
248	F8	4	Virtual timer source

Figure 2-1 (Part 1 of 2). Memory Locations Used as Virtual Registers

Memory location		Bytes	Value
252	FC	4	Virtual time since IPL
256	100	18C	PSBs (see Figure 2-3 on page 2-16)
652	28C	24	Task exception restart data (machine communications PSB extension)
688	2B0	24	Task exception restart data (program check PSB extension)
724	2D4	24	Task exception restart data (level 7 PSB extension)

Figure 2-1 (Part 2 of 2). Memory Locations Used as Virtual Registers

The VRM reserves the first 184 (0xB8) bytes of virtual memory, as well as two bytes at location 200 (0xC8). The rest of the memory locations are defined in the following section.

Paging Data

Locations 0xB8 through 0xC8 provide data on paging. The number of free paging disk slots should be used by a virtual machine to control its creation of segments. The virtual machine should not create new segments when this number is small. Using all of the paging disk slots will abend the system. The number of page replacement cycles, page faults with preemption, and non-paging disk I/Os should be used by a scheduler in a virtual machine to prevent ***thrashing***. Thrashing is a condition in which the system is doing so much paging that little useful work can be done.

POST Control Block (PCB) Segment Identifier

The POST (power-on self test) control block is a one-page segment that keeps track of the devices and memory of the real machine. The segment ID is available to the virtual machine at location 0xCA. POST control block addresses start at 0x000 and end at 0x7FF.

Note that the PCB cannot be altered in any way. Therefore, the only virtual memory SVCs that can be performed on this segment are **Load Segment Register SVC** and **Query Page Protect SVC**.

For more information on the format of the POST control block and the fields in a control block entry, see *VRM Device Support*.

Floating-Point Hardware

This byte indicates if any floating-point hardware is configured on the machine, and, if so, the configuration type. Possible values for this field include:

0 = no floating-point hardware configured
1 = Floating-Point Accelerator (FPA1) present
2 = APC floating-point coprocessor is present
4 = Advanced Floating-Point Accelerator (FPA2) present without APC
C = FPA2 present with APC (DMA operations supported).

Processor Type and Mode

Bit 0 of this byte indicates the type of processor in use on the machine. When bit 0 = 0, the Processor and Memory Management Card is configured. When bit 0 = 1, the Advanced Processor Card is configured. Bit 1 indicates if the processor supports loop mode. The virtual machine may optimize loops to run efficiently when in loop mode. When bit 1 = 0, the processor does not support loop mode; when bit 1 = 1, the processor may optimize loops. See *IBM RT PC Hardware Technical Reference* for details on loop mode.

Trace Buffer Synch Flag

This halfword is used as a pacing mechanism because the VRM can send only two buffers of trace data to the operating system at a time. The first byte contains a value for the number of buffers the VRM has sent to the operating system. The second byte contains a value for the number of buffers processed by the operating system. If the operating system count is two more than the VRM count, the VRM cannot send another buffer until the operating system processes one.

VRM Sequence Number

The basic unit of timer granularity is approximately 16.6 milliseconds (ms). Because the VRM can generate many trace entries in this period of time, the sequence number provides a greater granularity so that trace entries can be logged more precisely.

Virtual Machine Sequence Number

The basic unit of timer granularity is approximately 16.6 ms. Because the VRM can generate many trace entries in this period of time, the sequence number provides a greater granularity so that trace entries can be logged more precisely.

Interrupt Request Buffer

The halfword at location 0xD4 contains the interrupt levels queued to the virtual machine. Bits 0-9 map one to one with the corresponding bits of the virtual ICS (see "Virtual Interrupt Control Status (VICS) Register"). Bits 10-15 are zero. The interrupt request buffer keeps track of pending interrupts for virtual machines.

Process Priority

The value found at location 0xD6 reflects the priority of the process executing in the virtual machine. The VRM uses this value to prioritize input/output and paging activities. The process priority can be used by the virtual machine's operating system to influence VRM scheduling algorithms.

The range of values valid for process priority is 0-15. The highest priority is 0. The default value, 15, is the lowest priority.

Floating-Point Register Set

This byte has meaning only when the floating-point accelerator hardware is part of the system. The value at 0xD7 indicates which floating-point register set the virtual machine is using.

Bus I/O Base Address

The bus I/O address is a fullword of storage that contains the address of the beginning of bus I/O space. Virtual machines can perform I/O operations directly to bus-attached I/O devices when using bus I/O space.

Bus Memory Base Address

The bus memory base address is a fullword of storage that contains the address of the beginning of bus-attached memory. Virtual machines can load and store directly to bus-attached memory.

Virtual Interrupt Control Status (VICS) Register

The virtual interrupt control status register (VICS) describes the state of the virtual machine. The VICS is modified by SVC instructions, by virtual machine interrupts, by the **load program status** (lps) instruction, and by storing directly into the VICS register memory location.

Whenever you change any of the values in the VICS (with the exception of disabling certain interrupts), you must issue one of the execution control SVCs in order for the VRM to recognize the change. "Execution Control SVCs" on page 4-9 lists and defines these SVCs.

Disabling interrupts can be done with simple **load** and **store** instructions directly to the VICS and does not require issuance of an execution control SVC for the VRM to recognize

the change. To re-enable interrupts, however, an execution control SVC (such as the **No Operation SVC**) is required to effect the change.

The VICS is contained in location 0xE0.

Bits 7, 10, 13-15, and 24-26 of the VICS are reserved.

Bits 11 and 12 should not be changed by **load** and **store** instructions because the VRM will ignore these changes.

The rest of this virtual register is defined as follows:

Bits 0-6: Level in progress. A value of one in any of these bits indicates the preempted execution levels.

Bit 8: Program check in progress.

Bit 9: Machine communications in progress. Bits 0-9 indicate the execution levels that are in progress by setting the bit corresponding to the preempted levels to one. The bit for the current level is set to zero.

Bit 11: Termination in progress.

Bit 12: Virtual machine in wait state (**Virtual Machine Wait SVC** issued).

When any of bits 16 through 23 equal one, interrupts are inhibited on the indicated level. If interrupts occur for an inhibited level, they are queued.

Bit 16: Level 0

Bit 17: Level 1

Bit 18: Level 2

Bit 19: Level 3

Bit 20: Level 4

Bit 21: Level 5

Bit 22: Level 6

Bit 23: Machine communications level

Bit 27: Bus I/O access control. A value of one in this bit allows the virtual machine to access the bus-attached I/O devices. A value of zero prohibits access. Any attempt to access bus-attached I/O devices when this bit is zero gives the virtual machine a program check interrupt with data address exception on.

Bit 28: Bus memory access control. A value of one in this bit allows the virtual machine to access bus-attached memory. A value of zero prohibits access. Any attempt to access bus-attached memory when this bit is zero gives the virtual machine a program check interrupt with data address exception on.

Bit 29: Problem state. A value of one in this bit puts the virtual machine in problem state; a value of zero puts it in operating state. The VRM uses this bit to determine the legality of SVC calls.

Bit 30: Inhibit paging interrupts. A value of one in this bit puts the virtual machine in wait state until a page is moved to real memory. A value of zero causes a machine communications interrupt with the page fault indicator on when a page fault occurs.

A value of one in this bit causes page faults to be processed synchronously with no 'page fault occurred' interrupt. The setting of this bit does not disable 'page fault cleared' interrupts. A 'page fault cleared' interrupt is generated when the I/O completes for a page fault that resulted in a 'page fault occurred' interrupt. 'Page fault cleared' interrupts can be disabled by setting bits 23 or 31 of the VICS.

This bit is effectively on whenever the virtual machine executes on the program check or machine communications level. These levels cannot be interrupted by a machine communications interrupt.

Bit 31: Inhibit all levels (0-6). A value of zero in this bit causes VRM to use bits 16-23 of the VICS as the interrupt mask. A value of one inhibits all interrupts on levels 0-6 and the machine check level. If interrupts occur on any of these levels, they are queued. A one in this bit supersedes the settings of the interrupt mask in bits 16-23. SVC instruction interrupts and program checks cannot be explicitly masked.

Execution Level

The current execution level for the virtual machine is contained in the halfword at location 0xE4. The execution level value is updated whenever the execution level is changed. A virtual machine can use the processor's **store half** instruction to explicitly change the execution level. Along with the VICS and program status blocks, the execution level defines the current state of the virtual machine.

Virtual Machine Identification

Virtual machine memory location 0xE6 contains the 15-bit virtual machine ID when the virtual machine first receives control. The value at location 0xE6 reflects the ID of the virtual machine dispatched at IPL and is not updated.

Real Time of Day, Extended

The real time of day can be found at location 0xF0 (see "Real Time of Day"). The extended time of day value found at 0xE8 divides the current second into approximately 60 Hz when the counter is enabled (bit 2 of the Virtual Timer Status register is on). Thus, 16.6 ms is the finest granularity of real-time measurement available.

Virtual Timer Status Register

The word of memory at location 0xEC contains information concerning the current interrupt levels and whether the timer is enabled.

This virtual register is changed only as the result of executing the **Set Timer SVC** instruction (see "Set Timer SVC" on page 4-19). The virtual timer status register is defined in Figure 2-2 on page 2-12.

For more information on interrupts, see "Virtual Machine Interrupts" on page 2-14.

Real Time of Day

Location 0xF0 contains the time in seconds from midnight, Jan. 1, 1970. The current second is further divided into 1/60th of a second increments for more precise measurement. This extended value is found at location 0xE8.

Real Time of IPL

The word of storage at location 0xF4 contains a timestamp taken when the virtual machine was loaded.

Virtual Timer Source

In addition to time of day, the VRM supports a virtual machine timer (VMT). The VMT is a counter that allows the operating system in the virtual machine to perform time-slicing, accounting, and performance measurement functions. The VRM decreases the value of the VMT every 16.6 ms. The timer value is contained in storage location 0xF8.

The virtual timer source register can be loaded (by way of the **Set Timer SVC**) to cause an interrupt at fixed intervals. For example, if the timer is loaded with 3, an interrupt occurs every 3 times 16.6 ms. Each virtual machine may have a different setting for the virtual timer source. Therefore, the setting for a particular virtual machine is valid only when that virtual machine is currently running.

An interrupt is set to the virtual machine when all the following conditions are met:

- The counter decreases from 1 to 0.
- The timer is enabled.
- The interrupt level/sublevel has been specified through the "Set Timer SVC" on page 4-19.

Virtual Time since IPL

For each elapsed second, the VRM increases the time from IPL for that virtual machine. This value is reflected to the virtual machine in location 0xFC. The fullword at this location is defined as follows:

Bits	Meaning
0	Enable interrupts. When zero, no interrupts are created. This does not start or stop the virtual machine timer, but enables or disables the setting of virtual machine interrupt request register bits. When the virtual machine is initialized, this bit is zero.
1	Interrupt valid. When zero, no interrupts are created. This bit is set to one when a valid interrupt level is specified. When the virtual machine is initialized, this bit is zero.
2	Enable 60 Hz counter. When this bit equals one, the current second is divided into approximately 60 Hz. (Actual counter value is 16.6 ms, while one hertz equals 16.6666 ms.) Note that when the status of this counter is changed, the change may not take effect for 16.6 ms (if disabling the counter) or one second (if enabling the counter). This extended time-of-day value is then stored at location 0xE8.
3-20	Reserved
21-23	Timer interrupt priority level. A virtual machine timer alarm causes an interrupt on this level if the timer is enabled. The level must be in the range 0-6.
24-31	Timer interrupt sublevel. This is the sublevel for the priority interrupt level. Sublevels must be between 0-255 and must not exceed the number of sublevels defined for that interrupt level.

Figure 2-2. Virtual Timer Status Register

Program Status Blocks

Program status blocks are defined in "Virtual Machine Program Status Blocks" on page 2-15.

Task Exception Restart Data

When the Advanced Processor Card is configured in the system, execution of **load** and **store** instructions may overlap. That is, execution of one instruction may be in progress before the results of the previous instruction are known. The VRM exception handler may now receive two data exception program checks at the same time, and the source of the exception must be known in order to clear it. To handle this situation, the APC maintains an exception stack and a system control register (known as the exception control register) to address this stack.

Data exception program checks result in either a machine communications or program check virtual interrupt. If the virtual machine requires notification of the program check, the data required to restart the task is stored in the virtual machine's page 0 at 0x28C or 0x2B0.

Not all machine communications and program check virtual interrupts require a restart. The following virtual interrupts require a restart:

- Floating-point exceptions
- Segment length exceptions
- Protection exceptions
- Page faults
- Generic data exceptions.

To determine if an interrupt requires a restart, the virtual machine checks the value of the byte at locations 0x28C and 0x2B0. If the value of the byte at 0x28C is not zero, a program check has occurred that requires a restart. If the value of the byte at 0x2B0 is not zero, a machine communications interrupt has occurred that requires a restart. If the bytes at these locations are zero, the virtual machine does not return restart data to the VRM.

In order to restart the process that caused the exception, the virtual machine passes the restart data to the VRM in either of the following ways:

- Issue a **Return from Interrupt SVC** with the restart data at either 0x28C (machine communications interrupt) or 0x2B0 (program check interrupt) and the execution level set as appropriate.
- Copy the restart data into the level 7 PSB extension (0x2D4) and issue a **Dispatch SVC** or **Return from Interrupt SVC** with the execution level set at 7.

The VRM, upon receipt of the restart data, copies the data into the VRM process block for the virtual machine and then issues a **load program status** instruction to restart the process.

If the bytes at 0x28C and 0x2B0 are zero after an exception, no restart is required.

See "Dispatch SVC" on page 4-11 and "Return From Interrupt SVC" on page 4-17 for more information on exception processing.

Virtual Machine Interrupts

Virtual interrupts change the state of the virtual machine. A virtual interrupt is caused by a message or request from another system component or by conditions established by the operating system.

Virtual machines may receive the following types of interrupts:

- SVC instruction. Execution of SVC instructions with a function code less than 32,768 (operating system SVCs) changes the virtual machine state. SVC interrupts are taken immediately by the virtual machine and cannot be masked.
- Program check interrupt. Program check interrupts typically occur due to programming errors, such as an address out of range. Program check errors cannot be masked, and if the virtual machine receives a program check while on the program check level, that virtual machine is terminated.
- Machine communications interrupt. Machine communications interrupts typically occur because of an exception condition or a condition that the operating system can best handle. For example, if a process has generated a page fault, the operating system may want to switch to another process. Machine communications interrupts may be masked by setting bits in the VICS register.
- Execution level interrupts. Instructions execute on different levels according to a priority mechanism. Certain instructions, such as input/output SVCs, can be programmed to create an interrupt upon completion. If the interrupt priority exceeds the current execution level priority, and that interrupt level has not been inhibited, the interrupt is taken. If the interrupt priority does not exceed the execution level priority, the interrupt remains pending (except for program check interrupts). Execution level interrupts may be masked by setting certain bits of the VICS register.

Interrupt and Execution Priority Levels

Execution levels and interrupts (except for SVC instructions) each have a priority mechanism. The lower the numeric value of the execution or interrupt level, the higher the priority. The priorities for both the execution and interrupt levels, with the numeric value of the levels in parentheses, are as follows:

1. Program check (-2)
2. Machine communications (-1)
3. Level 0 (0)
4. Level 1 (1)
5. Level 2 (2)
6. Level 3 (3)
7. Level 4 (4)
8. Level 5 (5)

9. Level 6 (6)
10. Level 7 (7) (execution level only — not an interrupt level).

Interrupts occur only if the interrupt priority is higher than the priority of the current execution level. Interrupts that are not taken remain pending, except for program checks. For example, if the virtual machine execution level is 7, and the interrupt level is 2, the interrupt is taken. If the execution level is 4, and the interrupt level is 6, the interrupt remains pending.

Whenever the execution level changes, VRM stores the new level into the virtual register at location 0xE4 of the virtual machine. This halfword contains:

-2 (0xFFFE)	For the program check level
-1 (0xFFFF)	For the machine communications level
n (0x000**n**)	**n** represents the current execution level (0-7).

Whenever the execution level changes, VRM stores the updated VICS into the word at location 0xE0 of the virtual machine.

Interrupts are also prioritized by order of presentation. If more than one interrupt on different levels is pending and can be taken, the highest priority interrupt is taken. When two or more interrupts have the same priority (same level), the interrupt order is unpredictable.

In addition to the priority mechanism, level interrupts are also controlled by VICS masks. Interrupts are taken only when the VICS masks allow them, and masked interrupts remain pending.

SVC instruction interrupts, which occur on execution level 7 (normal instruction execution level), are taken at any time.

Virtual Machine Program Status Blocks

When an interrupt or SVC instruction occurs, the current status of the virtual machine must be saved so that it can be restored when the virtual machine finishes interrupt processing.

The status of the virtual machine, known as the program status, is derived from certain bits in selected registers. Program status information is stored in dedicated virtual memory locations called ***program status blocks*** (PSB).

Each interrupt level, as well as the SVC instruction, has a dedicated storage location for its PSB. In addition, the machine communications PSB, program check PSB, and level 7 PSB have noncontiguous extension areas to contain restart data. If a restart is required as a result of a machine communications or program check interrupt, the virtual machine returns the data for the appropriate interrupt type to the VRM. The VRM then restarts the interrupted process with the returned data.

When an interrupt occurs, the PSB is located by the type or level of the interrupt. The dedicated PSB locations are shown in Figure 2-3.

Address	PSB
0x100	Level 0 PSB
0x124	Level 1 PSB
0x148	Level 2 PSB
0x16C	Level 3 PSB
0x190	Level 4 PSB
0x1B4	Level 5 PSB
0x1D8	Level 6 PSB
0x1FC	Level 7 PSB
0x220	Machine Communications PSB
0x244	Program Check PSB
0x268	SVC PSB
0x28C	Machine Communications PSB Extension for Restart Data
0x2B0	Program Check PSB Extension for Restart Data
0x2D4	Level 7 PSB Extension for Restart Data

Figure 2-3. Virtual Machine Program Status Block Locations

PSBs consist of 36 bytes of information for each interrupt level or type.

Hardware and virtual machine registers that contribute information to PSBs are:

- Hardware instruction address register. Contains the address of the next instruction to be executed.

- Hardware condition status register. Contains information about the results of the last instruction executed and provides a mechanism for decision making.
- Virtual interrupt control status register. Contains information about the execution level and the state of the virtual machine.

For more information about hardware (as opposed to virtual) registers, see *IBM RT PC Hardware Technical Reference.*

A typical program status block is shown in Figure 2-4.

Offset				
0	Old IAR			
4	Old ICS			
8	Reserved	Old C S	Reserved	Number of Sublevels
12	New IAR / Origin of sublevel vector			
16	Reserved	New ICS	Status Flags	Overrun Count
20	Status			
24	Data			
28	Data			
32	Data			
36				

Figure 2-4. Virtual Machine Program Status Block

The fields that make up a PSB are defined as follows:

- Old IAR (IAR immediately prior to the interrupt)

  The IAR contains the address of the next instruction to be executed. The operating system typically returns to this address when the interrupt is cleared and resumes processing.

- Old ICS

  Bits 0-9 of this word are set as a result of an interrupt and indicate the virtual machine's preempted execution levels. When the processor is executing instructions and an interrupt other than an SVC interrupt occurs, the bit corresponding to the interrupted level is set to one. SVC instruction interrupts do not change the execution level.

  Bits 10-31 of this word contain information regarding interrupt masks, accessible devices and memory, and the virtual machine state at the time of the interrupt. This information is saved so the virtual machine can resume processing in the same state as before the interrupt.

- Old CS (Prior to the interrupt)

  The eight defined bits of the CS contain information at the time of the interrupt. When the operating system resumes with the next instruction (from the IAR), this information is automatically restored.

- Number of sublevels

  Execution level interrupts can come from a variety of I/O devices or the virtual timer facility. Sublevels associated with each main execution level specify the source of the interrupt.

- New IAR (Following the interrupt)

  The new IAR is the address of the interrupt-handling routine associated with an interrupt level and sublevel. In the case of an interrupt with no corresponding sublevels (program check, machine check, and SVC), this field points directly to the address of the interrupt-handling routine. For an interrupt with multiple sublevels, the VRM determines the value of the sublevel. The address of the offset into a table containing interrupt-handling routines is then determined from the following formula:

  New IAR = origin of sublevel vector + (4 times sublevel value)

  For example, assume the interrupt-handling routine for a level 3 interrupt is contained at 0x800. The VRM determines the sublevel to be 2, multiplies the sublevel by 4 (8), and adds this value to the address of the level 3 IAR (0x800 + 8 = 0x808). The address of the interrupt-handling routine for a level 3, sublevel 2 interrupt, therefore, is 0x808.

- New ICS (As desired following the interrupt)

  This field, which represents bits 24-31 of the ICS, dictates access to devices and memory, virtual machine state, and interrupt masks for interrupt processing. For example, a common way to process interrupts is to allow access to bus memory and input/output devices (bits 27 and 28 set equal to zero), place the virtual machine in OS state (bit 29=0) to enable VRM SVCs, and inhibit all other interrupts until the current interrupt is cleared (bit 31=1).

  This byte can be changed by loading and storing to the virtual storage address of the appropriate PSB.

- Status flags (For 0-6 level interrupts)

  PSBs for 0-6 level interrupts contain a field called status flags to indicate the cause of the interrupt. The status flags are defined in Figure 2-5 on page 2-19.

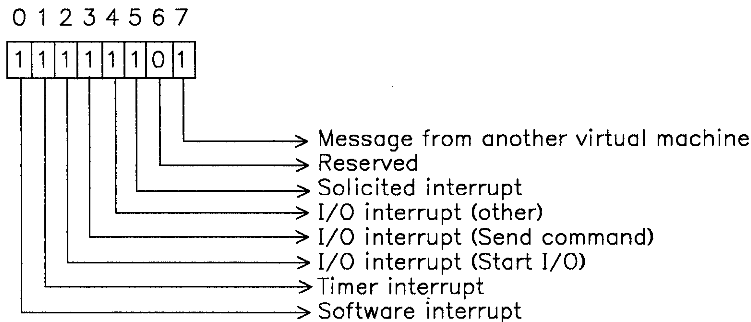


Figure 2-5. Status Flags for Execution-Level Interrupts

- Overrun count

  This field contains a value for the number of interrupts from a particular source lost because the operating system could not handle the interrupt volume. The timer is an example of a device that can cause overrun.

  Programming Note:

  When using timer interrupts to measure elapsed time (ET), use the following formula:

  ET = (# ints. + # overruns per int.) times (timer source times 16.6 ms.)

- Status and data associated with the interrupt.

  The status and data fields provide supplementary information for the current type of interrupt. For example, machine communications and program check interrupt PSBs have an associated status word that helps indicate the cause of these interrupts.

PSB Status Word

As mentioned previously, one word of each program status block is reserved for status information. The status word field is used by the program check and machine communications interrupts, as well as the supervisor call instruction.

Program Check Status Word

Program check interrupts are caused by a variety of conditions and cannot be masked. The status word of the program check PSB contains information indicating the cause of a program check interrupt. Conditions that cause program checks include:

- Execution of an unassigned or unimplemented operation code
- Execution of a privileged instruction with the problem state bit on
- Attempted virtual machine execution of a privileged VRM SVC

- An improper data condition that is detected by the execution of a trap instruction
- I/O access to a nonexistent or inoperative device
- Access to an invalid storage location
- Execution of an SVC with an SVC code >0x8000
- Execution of any VRM SVC with invalid data.

Program checks that may have restart data associated with them are floating-point exceptions (PCS bit 22 set to 1) and data exceptions (PCS bit 30 set to 1). The byte at location 0x2B0 indicates the presence and amount of restart data for a task exception. This byte is defined as follows:

- 0 = no restart data
- 1 = 20 bytes of restart data at 0x2B0
- 2 = 36 bytes of restart data at 0x2B0.

The virtual machine must save the task exception restart data with each task's machine state and be prepared to present the data back to the VRM in order to restart the task when the exception is resolved. The virtual machine can use either the **Return from Interrupt SVC** or the **Dispatch SVC** to present the restart data to the VRM.

If **Return from Interrupt SVC** is used, the virtual machine places the data at location 0x2B0. The VRM restarts exceptions when this byte is not zero.

If **Dispatch SVC** is used, the virtual machine copies the restart data into location 0x2D4 prior to issuing the SVC.

Because of the capability to overlap execution of **load** and **store** instructions on the APC, the origin of data exceptions will not usually be exact (bit 24 = 0). Instruction exceptions, however, will be exact.

The PCS format and content are as follows:

Bits 0-21 - Reserved

Bit 22 - Floating-point exception. Reserved for machines equipped with optional floating-point hardware. See "Floating-Point Exceptions" on page 7-10 for more detailed information.

Bit 23 - Segment length exception. This bit is set to one when you try to access an address outside the range of a valid segment. The first data word in the program check PSB (at 0x25C) contains the effective address that caused the exception.

Bit 24 - Program check with known origin. Bit 24 is set to one when a program check occurs and the address of the instruction causing the program check is the value of the old IAR in the program check PSB.

Bit 25 - Protection Exception. A load or store instruction has violated the protection characteristics of a segment. The first data word in the program check PSB (at 0x25C) contains the effective address that caused the exception.

Bit 26 - Program Trap. Bit 26 is set to one when a trap exception condition is generated by a trap instruction.

Bit 27 - Privileged Instruction Exception. Bit 27 is set to one when the processor attempts to execute an instruction privileged to the operating system when the problem state bit in the virtual machine's VICS is one.

Bit 28 - Illegal Operation Code. Bit 28 is set to one when the attempted execution of an unimplemented operation code is detected or if the processor attempts to execute an instruction privileged to the VRM.

Bit 29 - Instruction Address Exception. Bit 29 is set to one when instruction fetch is attempted from an invalid or protected address.

Bit 30 - Data Address Exception. Bit 30 is set to one:

- When no device on the I/O channel responds to a processor data request
- When a device on the I/O channel responds with an error indication
- When access to a privileged I/O device is attempted by the VM
- When a load or store instruction attempts to address an undefined segment.

Bit 31 - Reserved.

Programming Notes

- If bit 30 is set, bit 24 may or may not be set.
- The detection of a segment length exception sets bits 23, 24, and 30 to one. The old IAR in the program status contains the address of the instruction that caused the illegal reference. If the instruction is the subject of a branch with execute, the old IAR in the program status block contains the address of the branch with execute instruction.
- The detection of a protection exception sets bits 24, 25, and 30 to one. The old IAR in the program status contains the address of the instruction that caused the illegal reference. If the instruction is the subject of a branch with execute, the old IAR in the program status block contains the address of the branch with execute instruction.
- The detection of a program trap condition sets both bits 24 and 26 to ones. The old IAR in the program status block contains the address of the trap instruction.
- The detection of a privileged instruction exception sets bits 24 and 27 to ones. The old IAR in the program status block contains the address of the privileged instruction. If the privileged instruction is the subject of a branch with execute, the old IAR in the

program status block contains the address of the branch with execute instruction. Execution of a privileged VRM SVC in problem state generates this program check.

- The detection of an illegal operation code sets both bits 24 and 28 to ones. The old IAR in the program status block contains the address of the illegal operation. If the illegal operation is the subject of a branch with execute, the old IAR in the program status block contains the address of the branch with execute instruction. Execution of an undefined VRM SVC generates this program check.
- If an instruction address exception occurs, bits 24 and 29 of the PCS are set to ones. The old IAR in the program status block contains the address at which the processor attempted to execute an instruction. If a branch with execute has a subject instruction at an invalid address, the old IAR in the program status block contains the address of the branch with execute instruction.
- If a data address exception occurs, bit 30 of the PCS is set to one, and bit 24 is set to one to indicate the meaning of the old IAR in the program status block. If bit 24 is set to one, the old IAR in the program status block contains the address of the instruction that attempted to access the invalid storage location. If the subject instruction of a branch with execute attempts to access an invalid storage location, the old IAR in the program status block contains the address of the branch with execute instruction.
- If a data address exception occurs, the old IAR in the program status block points to an instruction that can be restarted when the exception conditions are resolved. In most cases, attempted execution of the instruction causing the data address exception has no effect on the values contained in the general purpose registers (GPRs) or data in storage. However, in the case of the load multiple or store multiple instructions, several of the load or store operations can occur before the exception is detected. The processor does not restore all GPRs or storage to the state that existed prior to attempted execution of the instruction causing the data address exception. However, if a load multiple causes an exception, the load multiple base address register (RC) is restored to its original value so that the load multiple instruction can be restarted.
- If a program check interrupt occurs while the virtual machine is on the program check level, the VRM terminates the virtual machine.

Machine Communications Status Word

Machine communications interrupts have several sources and can be masked by setting bit 23 or bit 31 of the VICS to one.

Conditions that cause machine communications interrupts include:

- Floating-point exception
- Page fault
- Imminent shutdown of the virtual machine.

Machine communications interrupts that may have restart data associated with them are floating-point exceptions (MCS bit 22 set to 1) and page faults (MCS bit 29 set to 1). The byte at location 0x28C indicates the presence and amount of restart data for a task exception. This byte is defined as follows:

- 0 = no restart data
- 1 = 20 bytes of restart data at 0x28C
- 2 = 36 bytes of restart data at 0x28C.

The virtual machine must save the task exception restart data with each task's machine state and be prepared to present the data back to the VRM in order to restart the task when the exception is resolved. The virtual machine can use either the **Return from Interrupt SVC** or the **Dispatch SVC** to present the restart data to the VRM.

If **Return from Interrupt SVC** is used, the virtual machine places the data at location 0x28C. The VRM restarts exceptions when this byte is not zero.

If **Dispatch SVC** is used, the virtual machine copies the restart data into location 0x2D4 prior to issuing the SVC.

The format and content of the MCS is:

Bits 0-18 - Reserved.

Bit 19 - Asynchronous purge page range complete. When this condition occurs, the first data halfword of the machine communications PSB (at 0x238) contains the unique identifier returned by the **Purge Page Range SVC**. All input/output associated with the command is now complete.

Bits 20-21 - Reserved

Bit 22 - Floating-point exception. This is reserved for machines equipped with optional floating-point hardware. See "Floating-Point Exceptions" on page 7-10 for more detailed information.

Bit 23 - Special key sequence detected. When this condition occurs, the first data word of the machine communications program status block (located at 0x238) contains the unique identifier of the key sequence. If this word contains zero, a virtual machine dump key sequence was detected.

Bit 24 - Reserved

Bit 25 - With bit 30, indicates that the page fault processing failed because of a hardware error (disk error).

Bits 26-28 - Reserved

Bit 29 - Page fault occurred. A page fault condition occurs only if VICS bit 30 is zero. This interrupt is followed by another machine communications interrupt with MCS bit 30 set to one. When a page fault occurs, the first two data words of the machine communications program status block (0x238) contain the address of the page causing the page fault.

The second data halfword of the PSB (at 0x23A) contains the segment identifier associated with the address. The second data word of PSB (at 0x23C) contains the effective address with the offset within the page set to zero.

Bit 30 - Page fault has been cleared. When a page fault clears, the first two data words of the machine communications program status block (0x238) contain the address of the page that caused the original page fault. The second data halfword of the machine communications PSB (at 0x23A) contains the segment identifier associated with the address. The second data word of the PSB (at 0x23C) contains the segment offset (the four most significant bits set to zero) with the offset within the page set to zero. This interrupt occurs only after a machine communications interrupt occurs with bit 29 set equal to one.

If multiple page faults for the same page have occurred, only one page fault cleared interrupt is given.

Bit 31 - Virtual machine shutdown is imminent. This bit is set when the VRM determines that the virtual machine must terminate. When the operating system is ready to process the termination, the VRM issues a **Terminate Virtual Machine SVC**.

Normally five seconds elapse between the virtual machine interrupt and VRM-forced termination.

Programming Note

The page fault and page fault cleared interrupts allow a virtual machine to dispatch another process while the VRM is resolving the page fault. If page fault interrupts are disabled, the VRM does not dispatch another virtual machine until the page fault is resolved. This mechanism potentially improves the performance of virtual machines that have more than one dispatchable process.

Supervisor Call Status Word

The PSB status word for a supervisor call contains the function code associated with the SVC.

The Interrupt Processing Cycle

Typically, a process executes on the base level (level 7) and is interrupted only to signal completion of an input/output operation or to dispatch another process. A very common interrupt-processing cycle would involve a process executing on level 7 interrupted for notification of a completed I/O operation. Any kind of interrupt has priority over level 7 execution, but assume the interrupt is associated with a level 5 device. The execution level of the current VICS is updated to 5 and the new IAR of the level 5 PSB points to the

appropriate interrupt-handling routine. In addition, the virtual machine's execution level (found at 0xE4) is updated.

When the interrupt-handling routine completes its work, it issues a **Return from Interrupt SVC**. (See "Return From Interrupt SVC" on page 4-17.) This SVC determines if any VICS bits (0-9) equal one (subsequent interrupt pending). If so, the bit corresponding to the new interrupt level updates the appropriate PSB and virtual machine execution level, and the interrupt-handling procedure begins again. If none of the VICS bits (0-9) are one, the original process resumes execution until it receives another interrupt or completes execution.

Processes typically run in virtual problem state. When a process is dispatched, the virtual machine state may be changed to problem state. The **Dispatch SVC** makes this state change when a new process is first dispatched. (See "Dispatch SVC" on page 4-11.)

Dispatch SVC also sets the execution level to 7 and sets bits 0-9 of the current VICS to zero.

When a dispatched process completes execution, it issues the **Return SVC**. This virtual problem state SVC simply returns machine control to the operating system, and changes the virtual machine state back to OS state. (See "Return SVC" on page 4-16.)

The **Return SVC** also sets the execution level to 7 and sets bits 0-9 of the current VICS to zero. The operating system then typically dispatches the next process for execution.

Sometimes, as in the case of multiple, successive interrupts, the virtual machine is not on level 7 when an interrupt occurs. In this case, the bit corresponding to the execution level is set to a binary 1 in bits 0-9 of the current VICS. The bit corresponding to the interrupting level is set to a binary 0. When the first interrupt is cleared, the **Return from Interrupt SVC** determines the next interrupt level to process from the highest-priority bit equal to one in the current VICS (0-9).

Interrupts and the lps Instruction

Another method of returning from an interrupt involves issuing a **load program status** (lps) instruction. The **lps** instruction performs updates and scans the current VICS field to determine pending interrupts, as does the **Return from Interrupt SVC**. The **lps** instruction, however, allows you to set new values for the ICS.

The **lps** instruction loads a new program status by pointing to a block of predefined storage. This block has a format similar to a PSB. The first word of this selected location replaces the value of the IAR. The second word replaces the value of the ICS. To find the new ICS value, you move four bytes beyond the address pointed to by the **lps** instruction.

By setting the bits of the second word, either at initialization time or with load and store instructions, you can change any of the characteristics associated with the ICS. These characteristics include current execution level, interrupt masks, device and storage access, and virtual machine state.

After the **lps** instruction executes, the interrupted process resumes with the next instruction in the newly specified environment. The **lps** instruction differs from the "Return from Interrupt SVC" in that the **lps** instruction can override the PSB defaults without altering the PSB.

At initialization, the operating system typically sets the level 7 PSB to an address in the dispatcher. The dispatcher then gains control when a program that is dispatched issues a **Return SVC**. The operating system sets the ICS in the program status block to reflect the state in which the dispatcher is to run. Normally, the dispatcher runs in the operating system state with interrupts inhibited.

Access to Segments

A virtual machine gains access to any of the 512 segments by issuing a VMI supervisor call. The supervisor call indicates that a particular segment is to be loaded into a particular segment register for certain kinds of access. See "Load Segment Registers SVC" on page 4-32 for more detail on loading registers.

Supervisor calls related to segments and addressing may be made only from the operating system state of the virtual machine. Thus, an operating system running in a virtual machine can directly control which segments can be accessed by its processes and the type of access allowed.

Since the VRM depends on values in virtual page 0 for communication with the virtual machine, a virtual machine must always have a valid segment ID loaded in segment register 0 and a well-defined page 0 in that segment.

Segment Register 14 Usage

Segment register 14 is reserved for the exclusive use of the VRM.

Segment Register 15 Usage

Addresses in this segment register access memory, memory-mapped input/output devices attached to the I/O bus, and the floating-point hardware.

Ranges of addresses within this area can be specified as privileged or unprivileged. Access to a privileged range within this segment by a virtual machine results in a program check data address exception. Access to an unprivileged range results in the appropriate bus memory access.

Note that if a device is specified in the hardware as unprivileged, and the device generates processor interrupts, the VRM must be able to process the interrupt properly and present an appropriate interrupt to a virtual machine. Before using such a device, the virtual machine must use the **Define Device SVC** and the **Attach Device SVC** (see "Input/Output Subsystem" on page 1-14) to ensure that the interrupts will be properly handled.

Warning: Use extreme care when specifying a segment register 15 address range as unprivileged. System integrity is jeopardized.

Protection of Segments

Protection of segments within a virtual machine is the responsibility of that virtual machine. The primary mechanisms for implementation of protection are:

- Distinguishing between virtual problem state and virtual operating system state
- Selective loading of segment IDs into segment registers
- Setting protection characteristics for pages within segments.

Loading segments into registers can be performed only while the virtual machine is in the operating system state. Problem state processes in the virtual machine must obtain segment loading services from operating system state processes. Similarly, page protection may be altered only from the operating system state.

A bit associated with each segment register indicates access privilege. This bit is not directly related to virtual problem, operating system, or VRM state. It is referred to as the unprivileged bit because when the bit is set (has a value of one), page-protected references through the register are less privileged than when the bit is cleared (has a value of zero). A virtual machine may effectively relate this bit to virtual machine problem or operating system state, but the machine itself has the option of having this bit automatically tied to virtual machine problem or operating system state. Otherwise, this bit can be modified only from operating system state.

Access privilege to a segment may be dependent on the setting of this bit. For example, when the bit is set for a particular segment and register, access to the segment through the register is more restricted than when the bit is not set. Access privileges for a segment may be defined on a page basis with respect to this bit. For either entire segments or individual pages, the following protection settings are supported:

- No access in unprivileged state, read/write access in privileged state
- Read/only access in both unprivileged and privileged states
- Read/only in unprivileged state, read/write access in privileged state
- Read/write access in both unprivileged and privileged states.

When page level protection is selected and a protection violation occurs, a program check interrupt indicating a protection exception will be given to the virtual machine.

Mapped Page Ranges

A mapped page range gives you the capability to define the relationship between the pages of a segment and logical disk blocks in the minidisk. Once defined, the virtual address space of a mapped page range is referenced in a normal fashion by the processor. Accessing the logical disk blocks is accomplished by load, store, and processor instruction fetches. The paging supervisor manages the transfer of the logical disk blocks to and from main storage.

The virtual machine must attach a minidisk in the normal fashion before creating a mapped segment on a minidisk. The virtual machine is responsible for the integrity of the minidisk in terms of conflicting accesses to blocks on the minidisk, whether those accesses are implicit as a result of mapped segments or explicit as a result of **Start I/O SVCs**.

The virtual machine must also unmap or destroy the minidisk's segment before closing the minidisk.

Three types of mapped page ranges can be defined. They are Read/Write (RW), Write-New (W), and Copy-on-Write (CW).

RW References to virtual addresses cause data to be paged-in without explicit disk reads and buffering. Changes to the data are implicitly written back to the disk when the underlying pages are paged out. You can explicitly tell VRM to update the disk by way of a virtual memory SVC.

W Data that did not previously exist on the disk can be written out by the paging system without the original contents of the disk blocks being read in first. This mode can be used to move data from the paging space minidisk (previously mapped CW) to the virtual machine's file system minidisk.

CW This type is similar to RW, except that when data is paged out it is *not* written back to the same disk block(s) it originally came from. Instead, it is written to the VRM page space. You must explicitly preserve this data if you require a permanent copy.

A read-only page range can be set up as a special case of a read-write page range. To set up a read-only page, map a page range as RW, then prevent the pages from being written to with page protection functions. A page range of this type can be used, for example, to map read-only data and program code.

Segment Sharing Between Virtual Machines

The preceding discussion assumes that segments are accessible from only one virtual machine, and this is normally the case. However, virtual machines can pass messages in segments, and thus two virtual machines can have access to one segment. A virtual machine that has created a segment (either directly or by a copy operation) may transmit the segment identifier to a second virtual machine using the mechanisms described in "Virtual Machine Communications" on page 1-17. The recipient may then retransmit the segment ID to another virtual machine.

Chapter 3. VRM Programming Environment

CONTENTS

About This Chapter 3-3
VRM Internal Characteristics 3-4
  Naming Conventions 3-6
Device Management 3-8
  Accessing a Device 3-9
  Managing Physical Resources 3-10
Process Management 3-13
  Process Creation and Termination 3-15
  Exception Handling 3-16
Queue Management 3-18
Memory Management 3-20
  Segment Register Utilization 3-21
  Direct Memory Access Support 3-26
Semaphore Management 3-28
Timer Management 3-29
Program Management 3-30
Minidisk Management 3-31
  Disk Space Allocation 3-31
  Virtual Memory Page Space 3-32
  Bad Block Management 3-32

About This Chapter

This goes into more detail on the internal components of the VRM and how they work together in the RT PC. This chapter defines the naming conventions used by the VRM, describes the input/output subsystem, and then discusses the component management tasks for which the VRM is responsible.

VRM Internal Characteristics

As stated, the **Virtual Resource Manager** (VRM) is a collection of processes, device drivers and runtime routines that extend and control hardware functions for an operating system. Figure 3-1 provides a more detailed look at the relationship between the primary VRM components.

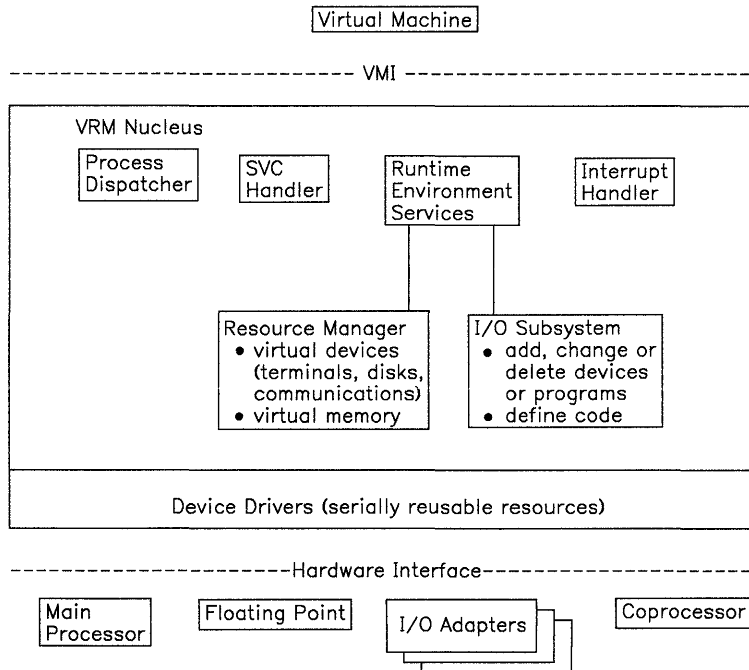


Figure 3-1. Virtual Resource Manager Structure

The RT PC is an "open" system in that code and devices can be dynamically added to the system. This open-system philosophy relies on well-defined conventions and protocols between the operating system and VRM drivers. For example, because VRM drivers handle all hardware I/O requests for the operating system, the VRM driver must know the format of operating system-generated requests, when to return a virtual interrupt, whether the interrupt should be solicited or unsolicited, and so on.

Together, processes and device drivers form that part of the VRM known as the input/output subsystem. Figure 3-2 on page 3-5 shows the control flow of the input/output subsystem.

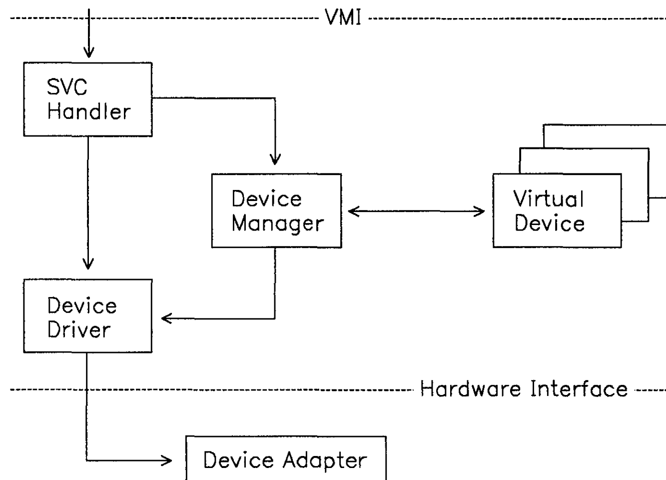


Figure 3-2. VRM I/O Subsystem

The input/output subsystem (IOS) provides a common device management and communication strategy for processes and device drivers. Virtual machines (which are themselves processes) communicate with devices (or device managers) by way of SVC instructions. A VRM SVC handler directs the SVC to the appropriate device driver or manager, or passes the SVC back to the virtual machine, depending on the SVC code.

A device driver, which is interrupt-driven, executes in a more restricted environment than a process. Many VRM functions are inaccessible to device drivers. For example, device drivers can use the queue, exception handling, and timer services of the runtime environment, but they cannot create processes, nor can they use semaphores. Also, device drivers cannot manage existing memory and have no heap storage support.

The VRM is designed so that most of the other VRM components work together to maintain the common device interface of the IOS.

The primary VRM components and the function each provides are summarized in Figure 3-3 on page 3-6.

Element	Function Provided
Processes	Asynchronous multiprogramming
Queues	Message passing, synchronization
Minidisks	Logical partitions of fixed disks
Virtual Memory	Expanded use of existing memory resources
Semaphores	Mutual exclusion, synchronization
Exception Handlers	Abnormal condition processing
Program Manager	Facility to copy, bind VRM modules
Debugger	Diagnose errors in VRM code
Timer	Time interval, time of day

Figure 3-3. Virtual Resource Manager Elements

Naming Conventions

The VRM recognizes devices and code modules by a 16-bit logical name. An input/output device number (IODN) uniquely identifies a device, and an input/output code number (IOCN) identifies the code.

Typically, the virtual machine assigns the IOCNs and IODNs. For virtual devices, the VRM assigns the IODN and returns the IODN to the virtual machine. You must assign an IOCN to code modules you install in the VRM. You can assign an IODN to a device you add to the system or have the VRM assign it for you. If you assign an IODN or IOCN that is already in use or is reserved, you will receive an error. See "Define Device SVC" on page 4-51 and "Define Code SVC" on page 4-49 for more information.

Within the VRM, however, components are known by 32-bit encoded identifiers. Each time the virtual machine assigns an IODN or IOCN, the VRM defines a corresponding 32-bit ID. IODNs and IOCNs can be considered 16-bit numerical names for code modules and devices. The 32-bit IDs have a more specific significance. Each is a bit-encoded modifier that provides additional information for a component. This information includes a type (to ensure that the requested function matches the type of element), generation (to ensure the validity of the identifier), and index (used to find the address of the corresponding control block).

The VRM assigns these 32-bit identifiers to the following objects:

- Processes (PID)
- Interrupt handlers (SLIH ID)
- Queues (QID)
- Semaphores (SID)
- Paths (path ID)
- Modules (MID)
- Devices (DID).

The VRM ensures that requested actions are appropriate for the object. For example, the program management function for binding external references in a module requires a module ID as input; use of another type of object name results in an error.

Because the VRM assigns identifiers dynamically, the value of the identifiers is not known until the code executes. You cannot "hard code" identifiers. Note that the 16-bit IOCNs and IODNs do not have the same value as the 32-bit MIDs and DIDs; they are related by internal VRM tables.

The path identifier associated with an IODN is returned from the **Attach Device SVC**. This value identifies the link between a virtual machine and a device. Virtual machines use the path identifier instead of the IODN when they issue the **Start I/O SVC, Send Command SVC**, **Cancel I/O SVC**, or **Detach Device SVC**.

Subsequent sections provide more details about the named objects and their associated actions.

Device Management

VRM device management, together with queue management, allows the addition of new devices to the system and communication between existing devices.

Each physical device is controlled by a device driver. A device driver consists of subroutines that are called by the VRM to handle the following operations for a device:

- Definition
- Initialization
- I/O initiation
- Interrupt handling
- Exception handling
- Termination.

Figure 3-4 and Figure 3-5 on page 3-9 show the relationships between the pertinent hardware components, including the standard or advanced processor cards, memory, busses, and optional devices.

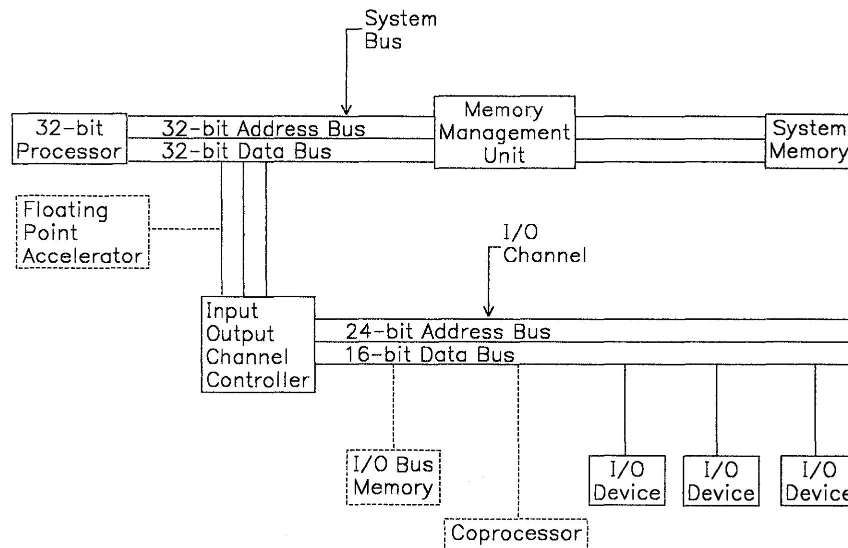


Figure 3-4. RT PC Hardware Structure with Standard Processor Card. The dotted lines represent specific optional devices available for use with RT PC.

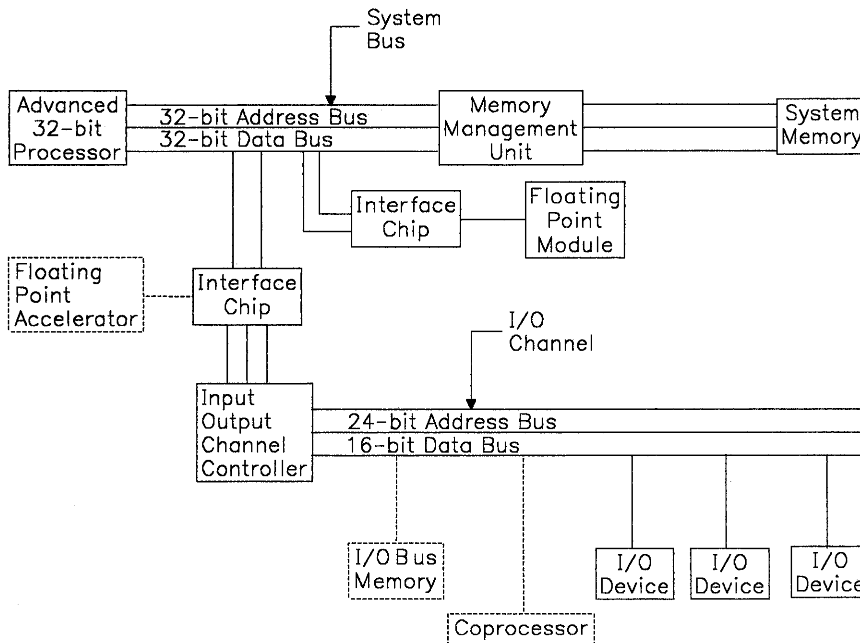


Figure 3-5. RT PC Hardware Structure with Advanced Processor Card. The dotted lines represent specific optional devices available for use with RT PC.

Note the system bus and the I/O channel shown in the preceding figures. Either processor can directly access either of the addresses on the system bus or input/output channel. Addresses from the I/O channel are 24 bits wide and addresses from the system bus are 32 bits wide. The ***input/output channel controller*** (IOCC) is the gateway between the system bus components (32-bit processor, memory management unit, and system memory) and the I/O bus-resident devices. All I/O devices are attached to the I/O bus.

Accessing a Device

A device must have an IODN assigned to it before a virtual machine can use it. The IODN for a device driver or process is assigned to the device by the virtual machine that issues the VMI **Define Device SVC**. Any code supplied by IBM is defined at IPL or initialization time, but code subsequently added to the VRM must first be defined from a virtual machine by way of a **Define Code SVC**. For virtual devices, a device manager calls "Define Device (_defind)" on page 5-110 for IODN assignment.

Each device has a query device structure (QDS). A QDS contains information describing the device and its current status. A device's QDS can be looked at with the **_qryds** call.

Device management initializes and maintains the QDS information that defines the device. Queue management automatically updates the operation completion information at the completion of an operation. A device driver or process is responsible for updating the device characteristics and error log fields in the define device structure (DDS). The DDS is a control block that contains all the pertinent information about a particular device. The DDS contains information on hardware and device characteristics, as well as device error logging functions. For more information on the role of the DDS for IBM-supplied devices or devices you may add to the system, see *VRM Device Support.*

Managing Physical Resources

The VRM manages the following physical resources:

- Adapters
- Interrupt levels
- Memory
- Direct memory access (DMA) channels.

The VRM identifies an adapter by combining the adapter type, port number, and the input/output base address from the device's DDS. The VRM ensures that only one device driver uses a device at any one time.

The VRM uses the interrupt type field in the device's DDS to determine how to allocate interrupt levels to device drivers. A non-shared interrupt level is allocated to only one driver at any one time.

The VRM uses several types of registers to manage its resources. Segment registers control virtual memory access and are described in "Segment Register Utilization" on page 3-21. System control registers reflect the status of certain parts of the system. The VRM maintains some virtual system control registers to keep track of multiple processes and device drivers.

System control registers are defined as follows:

- Counter source

  The counter source register contains a 32-bit value that is loaded into the counter when the counter is decremented from zero to one.

- Counter

  The counter register contains a 32-bit countdown counter that is decremented from an external source. When the counter is decremented from zero to one, the value from the Counter Source is loaded into the Counter.

- Timer status

  This 8-bit status register tells if any interrupts have been reported by the timer, and if so, how many.

- Exception control (ECR)

  The exception control register indicates in bits 4-7 the number of current exceptions. The address of the exception stack is contained in bits 8-31. The exception stack is shown in the following figure.

Offset	Field
0	Exception Control Word
4	Exception Address
8	Exception Data (store only)
12	Reserved

Figure 3-6. Exception Control Register Format

  The fields in the preceding figure are defined as follows:

  - Exception Control Word – contains the attributes of the **load** or **store** instruction.
  - Exception Address – indicates the base address of the exception.
  - Exception Data (**store** only) – contains the data that was the object of the **store**. For **load** instructions, this word is reserved.

- Multiplier quotient register (MQ)

  The multiplier quotient register provides an extension for a general purpose register to accommodate the result of certain mathematical operations.

- Machine check

  The machine check status register provides a means for reporting hardware malfunctions.

- Program check

  The program check status register provides a means for reporting certain programming errors.

- Interrupt request buffer (IRB)

  The interrupt request buffer allows interrupt requests to be generated under program control.

- Instruction address register (IAR)

  This register contains the address of the next instruction to be executed.

- Interrupt control status register (ICS)

  This register determines the state of the machine. Functions provided by this register include interrupt masking, enabling/disabling address translation, and the processor execution level.

- Condition status (CS)

  The condition status register contains information about the results of certain instructions.

For more information on the specifics of these registers, see *IBM RT PC Assembler Language Reference.*

Certain device types require direct memory access capabilities. See "Direct Memory Access Support" on page 3-26 for information on how devices utilize DMA facilities.

Process Management

A VRM ***process*** is a distinct entity that receives time from a processor to execute one or more programs. Processes work with device drivers, the other main components in the VRM input/output subsystem, to provide a device management interface to the operating system.

Multiple processes are supported in the VRM. Processes are dispatched (allowed to execute) based on priority levels, and round robin within the same level. The VRM supports 16 priority levels, from 0 (high) to 15 (low). Processes exist in the VRM to support ***multiprogramming*** below the virtual machine level. Multiprogramming is asynchronous execution of multiple operations. Virtual machines themselves are a special type of VRM process. VRM processes allow a component to wait for a condition, such as I/O completion, while the processor dispatches another process. All access to process management from the virtual machine is indirect. For example, a **Define Device SVC** can cause a device manager process to be created and dispatched.

VRM processes execute with supervisor state privilege. In other words, a process has access to any instruction, any data, and any I/O device in the system. Due to the open-coded nature of the VRM, you can install your own processes, each with supervisor state privilege. Obviously a component with supervisor state privilege can inadvertently compromise system integrity. The VRM can do little to ensure system integrity once processes or device drivers are installed. Therefore, IBM recommends you have a thorough understanding of the architecture of both the VMI and VRM before you install components into the VRM.

VRM process management design is characterized by:

- Efficient process switching
- Queueing that:
  - Provides general primitives for message passing and synchronization
  - Avoids complex, special-purpose function.
- Ability to lock shared resources.

A process can be in one of four states. They are:

- Ready
- Running
- Waiting
- Terminated.

Figure 3-7 on page 3-14 describes the relationship between process states.

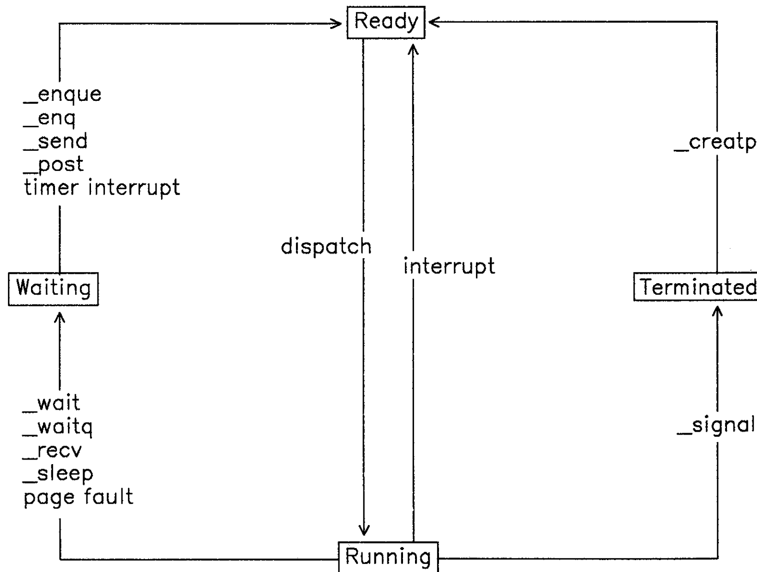


Figure 3-7. VRM Process States

When a process is created, the VRM typically initializes the process and places it in the ready state. The VRM then dispatches the process, which begins running.

A process that is running may be interrupted, at which time the VRM places it back in the pool of ready processes. The process may also be placed in a wait state. A process can wait for any of the following reasons:

- Event (such as the arrival of an element in a queue)
- Semaphore
- Page fault
- Timer.

Figure 3-8 on page 3-15 shows the call or interrupt type that can change a process' state from ready to waiting and vice versa.

Process State	Event Wait	Semaphore Wait	Timer Wait	Page Wait
Ready to waiting	_wait, _waitq	_recv	_sleep	I/O page fault
Waiting to ready	_enque, _post	_send	timer interrupt	page I/O complete

Figure 3-8. Rules for Process State Change

If the process is no longer needed or you want to free the resources used by a process, you can end the process. All of these process management functions are defined in "Process Management" on page 5-6.

Processes are dispatched on a priority basis. Within any one priority level, the VRM dispatches processes on a round-robin basis. The scheduling is preemptive; whenever a process enters the ready state due to the action of another process or device driver, the VRM dispatches the process with the highest priority. This may not be the same process that was preempted.

Although the process dispatcher controls which process executes, interrupts can affect the duration of that execution. Each interrupt level has a corresponding interrupt handler. When an interrupt occurs, the processor transfers control to the interrupt handler assigned to that interrupt level. After clearing the interrupt, the interrupt handler exits from the interrupt level. If another interrupt is pending, the corresponding interrupt handler is called. When all interrupts are cleared, control returns to the process dispatcher. Priority dispatching of processes resumes.

Process Creation and Termination

The five management functions for processes are:

- Change Process Attributes (**_change**)
- Create Process (**_creatp**)
- Initialize Process (**_initp**)
- Query Process ID (**_queryi**)
- Signal Process (**_signal**).

Figure 3-7 on page 3-14 indicates that the create process function changes a process from the terminated state to the ready state. Actually, this is a two-step process with an intermediate uninitialized state.

The **_creatp** routine reserves resources, creates the control blocks and a queue for the process, and informs the VRM of the new process.

The addition of the uninitialized state allows you to set up additional linkages to the new process or allocate additional resources before the process is first dispatched.

The **_initp** routine then places the process into the ready state for the first time, but only after you have a chance to set up the process with the resources you want.

The first time a process is dispatched, it appears as if the VRM process management component has called the process' main entry point as a subroutine. This entry point typically performs some initialization work, then enters a work loop. The work loop is characterized by:

- Waiting (for queues, semaphores, or the interval timer) until a work request arrives
- Performing the requested work
- Waiting for the next request.

A process terminates simply by returning to its caller. The process may execute a clean-up routine, then return to its caller (the process management component).

You can force a process to terminate with the **_signal** routine. A signal causes a process to stop what it had been doing and execute its exception handler routine.

The **_change** routine allows you to define additional entry points for certain module types.

The **_queryi** routine provides a means of finding the ID of a particular process. This ID is used for reference in many other calls.

Exception Handling

Processes can optionally define an exception-handling entry point. Another process can initiate execution of exception handler code by using the signal function. The exception handler routine can release resources to break deadlocks or perform emergency clean-up in situations such as imminent power down. Another use of the exception handler is for timer interrupt notification. For example, when a process wants to know when an interval of time expires, the exception handler notifies the process of this event.

The exception handler function is implemented as a software interrupt. The VRM saves the state of the process and transfers control to the exception entry point. A 32-bit mask passed as a parameter indicates the cause of the exception.

Both processes and device drivers can receive timer interrupts through the exception handler facility. "Timer Management" on page 3-29 describes how to request a timer interrupt.

Abnormal Termination

If a process is signalled to terminate, it typically issues commands to its associated device drivers (or other processes) telling them to terminate as well. The process then returns from the exception with a return code of 0. Process management recognizes this return code as an acknowledgement that the process is ready to terminate and treats the situation as if the process had returned normally from its main entry point.

During termination, process management releases the resources owned by the process. However, pages explicitly pinned by the terminating process must be released by the process' exception handler.

Queue Management

VRM processes and device drivers need to communicate work requests or send messages to each other. VRM queue management provides this facility.

The structures used to send work requests or virtual interrupts are called queue elements. Reserved areas of virtual memory provide the space for the allocation of these elements. The VRM memory management facility pins the real memory pages that back up this virtual memory. Therefore, you won't get a page fault when receiving queue elements.

The available queue management resources can conceivably be exhausted. If this unlikely event occurs, the process attempting to create a new queue or queue element must simply wait until the necessary resources are freed.

The VRM assigns a 32-bit identifier to each queue when it is created. In addition to this QID, a path ID indicates the component that enqueued an element (see "Enqueue Element (_enque, _enq)" on page 5-32). You establish a path when you attach to a queue, and you destroy a path when you detach from a queue. See "Attach Queue (_attchq)" on page 5-17 and "Detach Queue (_detchq)" on page 5-31 for more information.

Queue elements can be categorized as requests or acknowledgements. Request queue elements include two types for input/output operations, one type for communication between cooperating VRM components and one type for control in the input/output subsystem. The control queue element informs a device driver that the preceding I/O request was the last for the specified path. Although control queue elements have no user interface, device drivers must recognize the type 4 control queue element. See "Enqueue Element (_enque, _enq)" on page 5-32 for the format and definition of the other three request queue element types.

Acknowledgement queue elements are usually generated as a result of dequeueing a request queue element (although you can explicitly generate an acknowledgement). Acknowledgement queue elements are the elements used to implement virtual interrupts. The VRM can keep track of multiple virtual interrupts occurring on the same interrupt level with the queue management facility.

In general, the queue management functions used most often are:

- Enqueue an element
- Read the first element in a specified queue
- Dequeue an element.

The enqueueing or dequeueing of an element is referred to as an ***event***. Queue management keeps track of events with ***event control bits*** (ECB). Each queue served by a process has an ECB. When an element arrives in the queue, an ECB is turned on. Each process has 32 ECBs. ECBs 24-31 are reserved, but bits 0-23 can be used by queues.

Most VRM command and interrupt functions rely on queues as the basic means of communication. For example, each device (identified by its IODN) has a queue. A virtual machine makes an I/O request by issuing a **Send Command SVC** or a **Start I/O SVC**.

When the VRM receives either of these I/O SVCs, it builds a queue element that includes command-specific data (for the **Send Command SVC**) or a pointer to command-specific data (for the **Start I/O SVC**). The VRM sends this queue element to the specified device's queue. Within the VRM, a VRM component can simulate these SVCs by setting up a queue element and explicitly calling queue management to send the element to a specified device.

Most device drivers have a check parameters subroutine that can verify the validity of the queue element. If the check parameters subroutine detects an error, the device driver can reject the queue element. See *VRM Device Support* for more information on the check parameters subroutine.

Just as each device has a queue to receive commands, each virtual machine has a queue to receive virtual interrupts from the VRM. When a virtual machine's queue receives a virtual interrupt, the virtual machine treats the interrupt as a hardware interrupt. When a VRM process (such as device manager) receives a virtual interrupt, it is treated like a request queue element (**_readq** and **_deque** process it).

In the case of the **Start I/O SVC**, queue management automatically pins the memory area containing the command control block and buffers referenced by the command elements. For the **Send Command SVC**, queue management pins the command extension. After the device driver completes the request, it dequeues the element from its queue. During dequeue, queue management does the following:

- Updates the query device structure with operation results
- Unpins the command control block and buffers
- Enqueues a virtual interrupt (if one was requested) to the virtual machine.

Memory Management

The VRM provides virtual machines with paged virtual memory. This paging support is primarily hidden from virtual machines, such that virtual machines can treat virtual memory as physical memory with highly variable access times.

When the VRM tries to access a page that is not in memory, a page fault results. When a page fault occurs, the page fault handler searches the inverted page table (IPT) for an unused page frame. The IPT is pinned in memory and contains one entry for each page frame of real memory. If the page fault handler finds an unused page frame, it initiates an I/O request to load the page. If no unused page frame is found, the page fault handler uses an algorithm to select a page frame. If the page frame is modified, another I/O request is issued to write the contents of the page to the page space. After the page is written out, an I/O request loads the page frame with its new value.

Another table maintained by the virtual memory manager is the external page table (XPT). The XPT has one entry for each virtual page of memory. It maps pages in the virtual address space to blocks on the fixed disk. Because of its large size, this table is designed to be pageable.

Figure 3-9 on page 3-21 shows the real memory layout of the VRM.

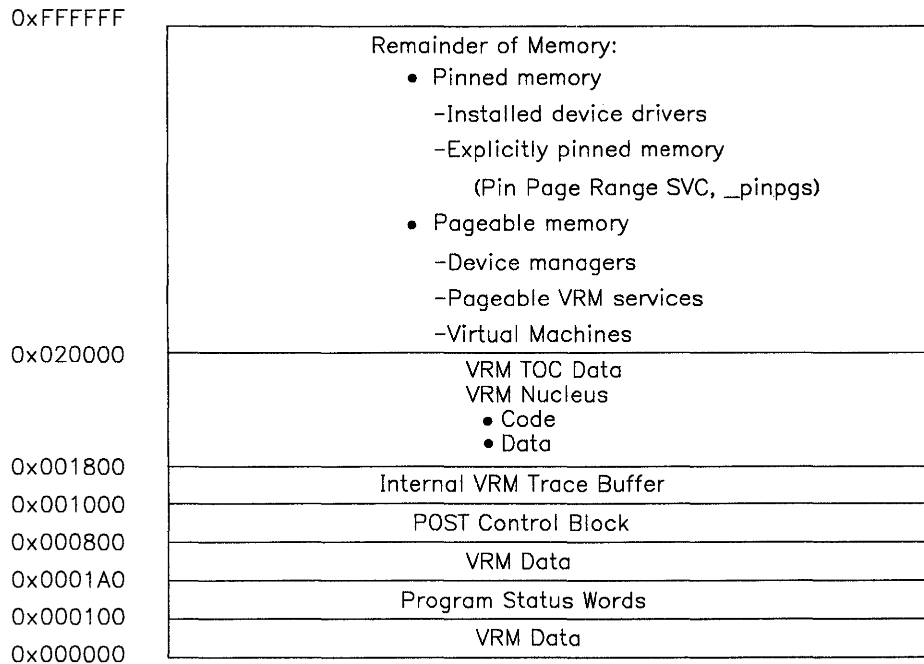


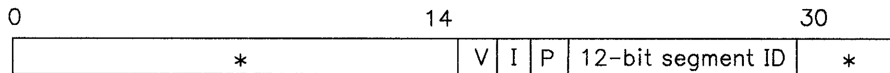
Figure 3-9. VRM Real Memory Mapping. Note that address 0x020000 is an approximation of the amount of memory required to contain the VRM nucleus and is not an exact value.

Segment Register Utilization

Virtual memory is divided into ***segments***, which are linearly-addressable spaces of one or more 2K-byte pages up to a maximum size of 2^{28} bytes. The exact size and protection characteristics of the segments are established when you create a segment with **Create Segment SVC**. The VRM returns a 12-bit segment identifier.

The VRM maintains 16 (0-15) segment registers to access virtual memory, but reserves registers 14 and 15 for its own use.

Figure 3-10 on page 3-22 shows the significant segment register bits.



∗ Set equal to zero

Figure 3-10. Segment Register Bit Map

In the preceding figure, bits 0 through 14 are reserved. Bit 15, the V bit, indicates when set that the segment ID is valid and that the memory management unit will respond to this address. The I bit indicates whether the segment can be accessed by the devices on the I/O channel. The P bit indicates whether the segment can be accessed by the 32-bit processor. Note that the I and P bits use reverse logic: when either bit equals one, it means no access by the I/O channel devices or the 32-bit processor. When either bit equals zero, access by the respective components is allowed.

Bits 18 through 29 contain the segment ID. Bits 30 and 31 are reserved.

A 32-bit effective address presented to RT PC hardware consists of the four high-order bits specifying the segment register and the next 28 bits specifying an offset from zero into the segment.

The following sections define the different ways that virtual machines, VRM processes, and VRM device drivers use the segment registers. Depending on the environment from which segments are accessed, different segment registers are saved and/or restored after certain operations.

Figure 3-11 on page 3-23 provides a virtual memory map of the VRM and summarizes how segment registers are used by the various VRM components.

Address		Segment Register
0xFFFF FFFF	Floating-Point Accelerator (option) Space	
0xFF00 0000	FPA2 DMA Mode	
0xFE00 0000	MC 68881 Non-Assist Mode	
0xFD00 0000	MC 68881 Assist Mode	
0xFC00 0000	Reserved	15
0xF4FF FFFF	Bus Memory Space	
0xF400 0000	Reserved	
0xF0FF FFFF	Bus I/O Space	
0xF000 0000		
0xEFFF FFFF	Maps addresses coming in from the I/O Bus (such as from the 286 coprocessor option)	14
0xE000 0000		
0xDFFF FFFF	For Processes: Used by VRM services For Device Drivers: Can be used, but may be changed by VRM services	12-13
0xC000 0000		
0xBFFF FFFF	For Processes: 4-10-Process-defined 11-Process Stack For Device Drivers: 4-10-Must be saved and restored after use 11 is reserved	4-11
0x4000 0000		
0x3FFF FFFF	Reserved for the VRM	3
0x3000 0000		
0x2FFF 0000	Used by Device Drivers or Managers	1-2
0x1000 0000		
0x0FFF FFFF	Reserved for VRM Nucleus	0
0x0000 0000		

Figure 3-11. VRM Virtual Memory Map/Segment Register Conventions

In general, segment register 15 always accesses the I/O bus, where all I/O devices and bus memory reside. Access to this space is typically limited to supervisor state. Do not change

the value in segment register 15. The following table shows how segment register 15 maps to specific areas of memory for certain components.

Address Range	Size	Usage
F000 0000 - F0FF FFFF	16 MB	Bus I/O map
F100 0000 - F3FF FFFF	48 MB	Reserved
F400 0000 - F4FF FFFF	16 MB	Bus memory map
F500 0000 - FEFF FFFF	110 MB	Reserved
FC00 0000 - FCFF FFFF	16 MB	MC68881 Assist Mode
FD00 0000 - FDFF FFFF	16 MB	MC68881 Non-Assist Mode
FE00 0000 - FEFF FFFF	16 MB	FPA2 DMA Mode
FF00 0000 - FFFF FFFF	16 MB	Map for optional floating-point hardware

Figure 3-12. Segment Register 15 Mapping

Segment register 14 is dedicated to mapping virtual machine references coming in from I/O bus devices, such as the optional coprocessor or adapters that use DMA. Do not change the value in segment register 14.

Segment registers 1 and 2 are used by check parameters routines (optional process or device driver subroutines) and I/O initiate routines (a required subroutine for device drivers). Segment register 2 points to the address of the data area and segment register 1 points to the command control block.

Because the dispatcher does not save the segment registers when it dispatches a new process or device driver, the runtime routines **_ssr**, **_rsr**, or **_lsr** must be used to manipulate segment register contents rather than using I/O instructions to change register contents.

To explicitly load a segment ID into a register, use **_lsr**.

Virtual Machine Segment Register Conventions

Each virtual machine is a unique VRM process that executes in problem state. Virtual machines use the segment registers as follows:

Register	**Usage**
0	Set up when the virtual machine is IPLed
1-13	Defined by the virtual machine
14	Maps addresses coming in from the I/O bus
15	Maps addresses going out to the I/O bus.

Note that the VRM places further restrictions on register 0.

VRM Process Segment Register Conventions

VRM processes allow multiprocessing below the VMI. Examples of VRM processes include device managers and protocol procedures. VRM processes use the segment registers as follows:

Register	Usage
0	Reserved for the VRM and used to address code and static data
1-2	Reserved for check parameters and I/O initiate routines
3	Reserved
4-10	Process-defined
11	Stack area
12-13	Used by VRM services
14	Maps addresses coming in from I/O bus
15	Maps addresses going out to the I/O bus.

A VRM process must define a convention for using segment registers 4-10. Registers 4-10 must be shared by the main process code, the exception handler, and any routines bound to the process with **_bind**. The process should not change registers 0-3 or 11-15.

If the process has a check parameters routine associated with it, the check parameters routine can change registers 1 and 2 without restoring them. Check parameters routines can also change registers 4-10 if the registers are restored to their original value after use.

The value of segment registers 1 and 2 changes if the process calls either **_enque** or **_deque**.

VRM Device Driver Segment Register Conventions

Device drivers provide an interface to hardware adapters. Device drivers use the segment registers as follows:

Register	Usage
0	Reserved for the VRM and used to address code and static data
1-2	Can be used by device drivers
3	Reserved
4-10	Must be saved and restored when used by a device driver
11	Reserved

12-13	Can be used by a device driver, but may also be changed by certain VRM services
14	Maps addresses coming in from I/O bus
15	Maps addresses going out to the I/O bus.

Most device driver subroutines, with the exception of the define device subroutine, can use segment registers 1 and 2 without restoring them. They can also change segment registers 4 through 10 if the register contents are restored after use. The define device subroutine cannot change any of the segment registers. See *VRM Device Support* for more information on device driver subroutines.

Device drivers must not change the other segment registers.

Direct Memory Access Support

Direct memory access (DMA) refers to an I/O adapter that can read from or write to system memory directly (without processor intervention). With this capability, the processor and the I/O device can execute in parallel.

Maximum system performance and coprocessor compatibility determine the VRM services provided for DMA support.

The VRM provides eight DMA channels of I/O bus arbitration and translation control words (TCWs) to handle address translation from DMA devices on the I/O bus. TCWs exist in the input/output channel controller and perform the following functions:

- Supply the appropriate number of high-order address bits to form the required address
- Determine if a memory reference is directed to system or bus-attached memory
- Relocate within system memory any specific memory references made by I/O channel-resident DMA devices.

See *IBM RT PC Hardware Technical Reference* for a detailed description of the input/output channel controller and the translation control words.

TCWs support two address translation modes, page mode and region mode. They are defined as follows:

Page — In page mode, each TCW entry maps a page (2K bytes) on the I/O channel to a page in system memory. As many as 512 TCWs (64 per channel) can be used in page mode. Both system and alternate DMA controllers can operate in page mode. The VRM also provides a subchannel mechanism for alternate DMA controllers. Page mode TCWs are mapped each time a DMA controller calls **_stdma** for a DMA transfer. For more information on how the VRM handles page mode DMA devices, see "Start Direct Memory Access Transfer (_stdma)" on page 5-97.

Region In region mode, each TCW entry maps a 32K-byte area on the I/O channel to a 32K-byte area in system memory. As many as 512 TCWs, shared by all channels, can be used in region mode. This mode can be used only by alternate controllers, such as the coprocessor option. See "Map System Memory (_mapsys)" on page 5-113 and "Set up region mode (_dmamov)" on page 5-117 for more information on region mode DMA devices.

DMA for processes running in the system processor typically use page mode, and the coprocessor option always uses region mode.

The VRM uses the DMA type field in the device's DDS to determine how to allocate DMA channels. Channel 8 is intended for use by the coprocessor. A non-shared DMA channel is allocated to only one device driver at any one time. A shared DMA channel can be allocated to more than one driver under the following conditions:

- The interrupt handlers have the same DMA type values.
- DMA subchannels are prearbitrated by the same alternate DMA controller.

The VRM provides support for device drivers to perform programmed I/O using real or virtual addresses, and DMA transfers to or from real or virtual addresses in system memory.

DMA for processes running in the system processor typically use page mode for translating DMA accesses. Here TCWs solve alignment problems and correct for implied scatter created by the virtual memory manager.

The coprocessor uses region mode for translating memory addresses. Alternate DMA devices under direct control of the coprocessor are given effective address spaces identical to the coprocessor when the devices are allocated to the coprocessor with **_assignd**. This copies the TCWs associated with the coprocessor's channel into the TCWs that correspond to the system DMA device and provides the device with an identical effective memory address space as the coprocessor.

System DMA devices are not under direct control of the coprocessor. Input/output operations to these devices are trapped by code in the Processor and Memory Management Card that emulates the requested function. These devices are typically mapped using page mode with an address generated from the coprocessor's page register and DMA address. These transfers are mapped using **_stdma** for each DMA transfer. This copies the TCWs associated with the coprocessor's channel into the TCWs that correspond to the system DMA device and provides the device with an identical effective memory address space as the coprocessor.

Semaphore Management

Processes typically share resources, such as data structures in commonly addressable areas of memory and certain hardware devices. A process can be preempted in the midst of updating a shared data structure by a process with a higher process priority. ***Semaphores*** provide a mechanism for locking resources to processes until a task is complete.

Certain programming conventions must be followed for semaphores to lock resources. The basic rule for locking resources with semaphores is that you receive status from a semaphore before using a resource and you send status to a semaphore after using a resource. The receive function checks the semaphore's count. If non-zero, the count is decreased and the calling process remains in the running state. If the count is zero, the calling process enters the wait state. Multiple processes can wait for the same semaphore. The send function increases the semaphore's count. If the count changed from zero to one, the highest priority process waiting on the semaphore (if any) is allowed to proceed.

The send and receive functions provide prioritized mutual exclusion, not simply serialization. This is achieved because waiting processes are released in order of priority instead of arrival sequence. See "Receive Semaphore (_recv)" on page 5-65 and "Send Semaphore (_send)" on page 5-66 for more information.

Although only one process can serve a queue and a single process can serve multiple queues, the inverse is true for shared resources. Multiple processes can wait until a single resource is available, but a single process can wait for only one resource at a time.

Pacing functions can be implemented using semaphores with count values other than just zero or one. For example, in an application with 10 available buffers shared by several processes, a semaphore is initialized with a count of 10. Each process receives from the semaphore before using a buffer, decreasing the semaphore's count. When the process returns the buffer to the pool, the process sends to the semaphore, increasing the semaphore's count. If the count ever becomes zero, subsequent requests by a process for a buffer must wait until a buffer becomes available.

The send and receive functions by themselves do not work for nested locks. For example, if a process receives from a semaphore, then calls a subroutine that attempts to receive from the same semaphore, the process would wait for itself.

A nested locking function can be built with a subroutine using a semaphore and two data areas. The data areas keep track of the ID of the lock owner and the nesting count.

Timer Management

In addition to the timer service SVCs available to virtual machines, interval timer support is available to VRM components. Three functions are provided: set, cancel, and wait. Interrupt handlers can use the set and cancel functions, but cannot use wait.

A device timer function is also provided. The device timer function detects timeout conditions in a device driver. The device timer is similar to setting an interval timer to interrupt the component periodically. However, the time interval is related to the processing of a queue and is started and stopped automatically as elements are enqueued and dequeued.

During initialization, a device driver can set up the device timer with a value measured in half seconds and associate the time interval with a queue.

When the I/O initiate entry point of a device driver is called, an interval timer request is set. Upon dequeue of the element, the interval timer request is cancelled. If the time interval expires before the element is dequeued, the device is considered to have timed out.

Setting the device timer causes an automatic interval timer request each time a queue element begins processing for the specified queue.

Sometimes the processing of a queued request by a device driver can span multiple I/O operations and consume a large amount of time. For example, a request to format a disk requires many individual I/O operations before the disk device driver dequeues the request. In this case, it is desirable to measure the interval between a command to the adapter and the interrupt it generates, rather than the entire duration of the operation. Therefore, the device timer can be reset to its original value, stopped, and restarted.

Time-of-day support is available in that VRM components can read the memory mapped time-of-day clock value. However, no time-of-day comparison function is offered. Instead, notification can be requested after a specific interval expires. The interval is expressed in timer units of 975.562 microseconds.

Memory-mapped timer information consists of four words. They are:

- Time of year
- Time of year that the VRM was IPLed

  Both of the preceding values are unsigned integers indicating the number of seconds since January 1, 1970.

- Number of seconds since the VRM was IPLed
- Extended time of year (a count with a frequency of 60 Hz).

Program Management

VRM program management provides services for copying and binding program modules. The modules must conform to the VRM-compatible module format and the VRM runtime environment. Appendix B, "TOC Object Module Information" on page B-1 describes the VRM-compatible format. The modules must reside on the VRM minidisk, or they must have been installed by a virtual machine through the **Define Code SVC**.

Each module has a module identifier (MID) that uniquely identifies the module's IOCN.

Each module also has a module type determined by the characteristics of the module's read/write section. Program management uses the module type to determine how to ensure that a logically complete copy of the module exists for each process. The three module types are:

Reentrant
: A reentrant module can be shared by multiple processes without duplicating the read/write section. These modules can contain static data when semaphores are used to ensure mutually exclusive access to the static variables. The static variable can have initial values because only one copy of the read/write section exists.

Serially reusable
: A serially reusable module is less restrictive than a reentrant module. Because the read/write section can contain static data, this section must be duplicated for each process that calls a module's procedure. By duplicating the read/write section, each process thinks it has a private copy of the entire module. An original copy of the read/write section is not maintained and therefore a new copy may not have the correct initial value for each static variable.

One-Use
: This module type can have only one user and is not shareable.

The primary program management functions are provided by "Copy Module (_copy)" on page 5-78, "Bind Module (_bind)" on page 5-76, "Allocate memory (_mallc)" on page 5-75, and "Free Allocated Memory (_mfree)" on page 5-80. The copy module function is employed to give each process a logically complete copy of the module. For a reentrant program, the copy consists of a shared copy of the read only section and a shared copy of the read/write section. The copy for a serially reusable program consists of a shared copy of the read only section and a unique copy of the read/write section.

The bind module function resolves references to external procedures or external static data. The **Define Code SVC** automatically binds each module to the appropriate runtime routines. The bind module function, therefore, should be used only to extend the runtime capabilities, such as allowing devices to be configured from multiple modules.

A process must have its own logically complete copy of a module if the process wants to call a procedure or access a variable from that module. Therefore, serially-reusable modules should be bound only to other serially-reusable modules owned by the same process or to a reentrant module. A reentrant module can be bound only to other reentrant modules.

Minidisk Management

Minidisks are designed as partitions of fixed disks instead of virtual disks. The minidisk manager partitions a fixed disk into one or more minidisks, where each minidisk is a contiguous allocation of blocks on the fixed disk. Therefore the virtual machine's allocation routine can take advantage of physical proximity to improve performance.

Warning: If you bypass the minidisk manager to access the underlying physical disk, you may receive unpredictable results if other users are accessing the physical disk by way of the minidisk manager.

Disk Space Allocation

Minidisks are allocated as physically adjacent partitions. As illustrated in Figure 3-13, the page space for virtual memory is a separate paging minidisk. Typically, minidisks are allocated from the next adjacent empty space after the last existing minidisk, with three exceptions. Those exceptions are:

- Because minidisks cannot span fixed disks, a minidisk is allocated on the next fixed disk boundary if it does not fit in the space remaining on the current fixed disk.
- If you have more than one fixed disk, you can specify on which disk to place a new minidisk. This would be desirable, for example, to place a virtual machine's data minidisk on a different fixed disk than the VRM or paging minidisk. This can prevent conflicts in arm scheduling between the paging system and the virtual machine's data management subsystem.
- You can specify the relative position of a new minidisk on a physical disk. Each physical disk is divided into three contiguous areas starting at the first sector through the last sector on the disk. Thus, if you specify the middle section, the new minidisk would be allocated somewhere in the middle third of the physical disk. The three areas on the disk are not physical boundaries, and minidisks can extend across them.

When a minidisk is deleted, its space is marked as free. If contiguous minidisks become deleted, their space is joined into a single unit.

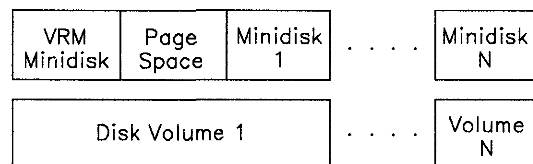


Figure 3-13. Minidisk Map

Fragmentation can occur, but the problem should not be severe since the allocation and deletion of minidisks should be infrequent. Disk space that becomes overly fragmented can be compressed by dumping the minidisks to tape and reloading. Minidisks are intended to be used as virtual disks, not as individual files.

Virtual Memory Page Space

If all paging space is ever consumed, the system abends.

When you install the VRM, you can specify the size to reserve as paging space. See *IBM RT PC Installing and Customizing the AIX Operating System* for more information.

Bad Block Management

When the physical disk is originally formatted, any existing bad blocks are noted and never allocated. Logically, they do not exist.

When a block goes bad as it is being written, an alternate block is assigned and used instead. When a block goes bad as it is being read, the virtual machine is notified by virtual interrupt, and the data, up to the point of error, is presented for possible recovery.

Chapter 4. System Control Instructions

CONTENTS

About This Chapter 4-5
Load Program Status Instruction 4-6
Supervisor Call Instructions 4-7
Execution Control SVCs 4-9
  Allocate Floating-Point Register Sets SVC 4-10
  Dispatch SVC 4-11
  Free Floating-Point Register Sets SVC 4-12
  No Operation SVC 4-13
  Post SVC 4-14
  Query Floating-Point Register Sets SVC 4-15
  Return SVC 4-16
  Return From Interrupt SVC 4-17
  Set Interrupt SVC 4-18
  Set Timer SVC 4-19
  Soft Interrupt SVC 4-21
  Virtual Machine Wait SVC 4-22
Memory Management SVCs 4-23
  Change Segment Size SVC 4-24
  Clear Segment Registers SVC 4-25
  Copy Segment SVC 4-26
  Create Segment SVC 4-27
  Destroy Segment SVC 4-29
  Discard Page Range SVC 4-30
  Load Segment Registers SVC 4-32
  Map Page Range SVC 4-34
  Pin Page Range SVC 4-37
  Protect Pages SVC 4-38
  Purge Page Range SVC 4-39
  Purge Segments SVC 4-41
  Query Page Protect SVC 4-43
  UnMap Page Range SVC 4-44
  UnPin Page Range SVC 4-45
Input/Output SVCs 4-46
  Attach Device SVC 4-47
  Cancel I/O SVC 4-48
  Define Code SVC 4-49
  Define Device SVC 4-51
  Detach Device SVC 4-57
  Query Device SVC 4-58
  Ring Queue Get Word SVC 4-61
  Ring Queue Put Word SVC 4-62
  Send Command SVC 4-63
  Start I/O SVC 4-67
Virtual Machine Communications SVCs 4-72

Send Address Message SVC 4-73
Send Immediate Message SVC 4-74
Set Message Receive SVC 4-75
Machine Control SVCs 4-76
Debug a Virtual Machine SVC 4-77
IPL Virtual Machine SVC 4-78
Machine Identification SVC 4-82
Query Virtual Machine SVC 4-83
Re-IPL VRM SVC 4-85
Terminate Virtual Machine SVC 4-86
Update VRM SVC 4-87
NVRAM Control SVCs 4-88
Read NVRAM SVC 4-89
Write Data to NVRAM SVC 4-90

About This Chapter

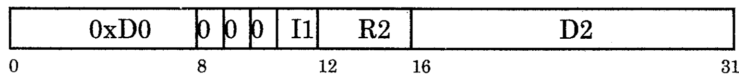
This chapter describes the instructions that control the execution state of the virtual machine. These include a virtualized load program status (LPS) instruction and the supervisor call (SVC) instruction. Because of the number of SVC variations that can be handled by the VRM, the majority of this chapter deals with SVCs. Each SVC definition includes a discussion of calling register conventions, hexadecimal representation of SVC codes, and return register conventions. Some instructions are followed by a "Comments" section, which elaborates on points made in the preceding text. For a description of the SVCs directed to the virtual terminal subsystem, see *VRM Device Support*.

If you have the AIX Operating System and are programming in the C language, please note that the SVCs and their parameters are found in the AIX Operating System file **ksvc.h**. Some of the C language structures used in these SVCs can be found in the AIX Operating System files **kcfg.h** and **kio.h**.

Load Program Status Instruction

The **load program status instruction (lps)** is privileged to the operating system. This instruction can change the state of the virtual machine from operating system to problem, and it can change the level of interrupt processing.

The format of the **lps** instruction is as follows:



In the preceding figure, please note that I1, R2, and D2 represent variables and the other fields reflect actual values. Pertinent variables used to describe the **lps** instruction are defined as follows:

R2 This denotes a general purpose register used as the second operand. R2 must be a hexadecimal integer in the range 0x0 through 0xF.

D2 This indicates a displacement from a base address.

For the **lps** instruction, 0/(R2) plus the sign-extended D2 field is used as a pointer to an area of memory that is treated as a PSB. If R2 equals 0, add nothing to the sign-extended D2 field. R2 may equal 0x1 through 0xF, representing the value of any of the registers 1 through 15. In this case, add the contents of the corresponding register to the sign-extended D2 field. The IAR is replaced by the word at offset 0 of the PSB. The contents of the virtual interrupt control status (VICS) register are replaced by the word at offset 4. The contents of the condition status (CS) register are replaced by the byte at offset 9.

D0 represents the operation code of this instruction. I1 can have a value of 0 or 1. If I1 equals 0, the execution level and bits 0-9 of the VICS are not modified. If I1 equals 1, the VRM scans bits 0-9 of the word at offset 4 for the highest priority one bit. If the VRM finds a bit equal to one, it sets the execution level to the level of the one bit found, and then sets the bit to zero. If the VRM finds no bit equal to one, it sets the execution level to 7 and sets ICS bits 0-9 to zero. Either way, the VRM reflects the execution level at the halfword at location 0xE4 and stores the updated VICS at location 0xE0.

You must execute this instruction only from operating system state, or the virtual machine will receive a privileged instruction exception.

Supervisor Call Instructions

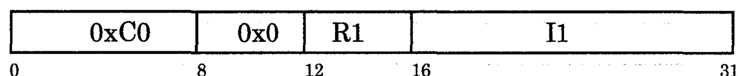
The rest of this chapter describes the supervisor calls directed to the VRM. You can issue the VRM SVCs only from virtual machine operating system state except where specifically stated otherwise.

An SVC causes a trap to the VRM. The processor state changes from unprivileged to privileged. The VRM SVC handler examines the SVC code. The SVC code is the low-order 16 bits of the 32-bit sum of 0/(R1) plus 16 zeroes // I1.

The SVC code not only tells the VRM what function is requested, it also tells the VRM whether to handle the SVC itself or to pass the SVC to the operating system for execution.

SVCs with function code > 32,767 are VRM SVCs. SVCs with function code < 32,768 are operating system SVCs.

The format of the SVC instruction is as follows:



In the preceding figure, please note that R1 and I1 represent variables and the other fields contain actual values. These variables are defined as follows:

R1 This denotes a general purpose register used as the first operand. R1 must be a hexadecimal integer in the range 0x0 through 0xF.

I1 This denotes a field of immediate data used as the first operand.

C0 represents the operation code for this instruction. The I1 variable is a halfword padded to the left with zeroes which is added to zero if R1 equals zero. R1 may also be a value in the range 0x1 through 0xF, representing the registers 1 through 15. In this case, the value of the corresponding register is added to the value of 16 zeroes // I1.

For passthrough SVCs (those SVCs routed by the VRM SVC handler back to the operating system for execution), the updated contents of the instruction address register (IAR) replace the word that begins at main storage location 0x268. The contents of the virtual interrupt control status register (VICS) replace the word that begins at 0x26C. The contents of the condition status register (CS) replace the byte at main storage location 0x271. The SVC code (the low-order 16 bits of the 32-bit sum of 0/(R1) plus 16 zeroes // I1) is stored into the word that begins at 0x27C.

The word beginning at 0x274 replaces the contents of the IAR. The byte beginning at 0x279 replaces the contents of bits 24-31 of the VICS. Reserved bits in either the IAR or ICS may be set to unpredictable values.

The VRM SVC handler further divides incoming VRM SVCs into two types. Type 1 SVCs execute synchronously with the virtual machine and do not require a complete context save. Type 1 SVCs

perform relatively simple functions that do not explicitly wait for input or output. Type 2 SVCs may execute synchronously or asynchronously with the virtual machine and require a full context save. Type 2 SVCs typically require some input or output to be performed. Figure 4-1 shows how the SVCs are divided.

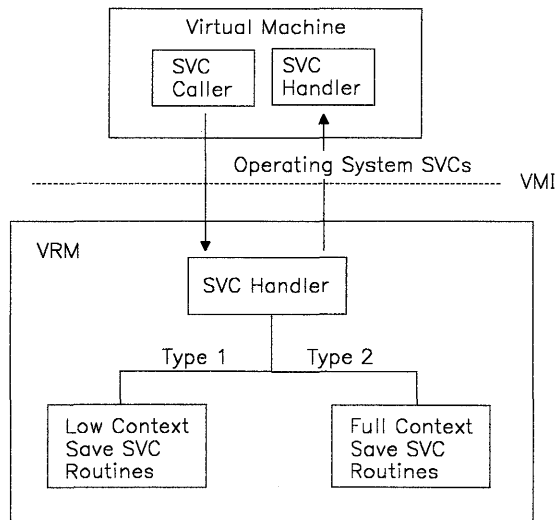


Figure 4-1. SVC Types

Execution Control SVCs

The following SVC instructions control the execution state of the virtual machine. These instructions modify certain virtual machine control registers and control the processing of virtual interrupts. If the memory-mapped VICS or execution level has been changed, execution of one of these SVCs causes the VRM to recognize the change. The **No Operation SVC** does nothing more than update this virtual machine control information.

The VRM recognizes the following execution control SVCs from the virtual machine:

- Allocate Floating-Point Register Sets
- Dispatch
- Free Floating-Point Register Sets
- No Operation
- Post SVC
- Query Floating-Point Register Sets
- Return
- Return from Interrupt
- Set Interrupt
- Set Timer
- Soft Interrupt
- Virtual Machine Wait.

Allocate Floating-Point Register Sets SVC

Description: This SVC allocates a floating-point register set for a virtual machine with the optional Floating-Point Accelerator installed.

SVC Code: 0x**FFB0**

Calling Register Conventions:

GPR2 = Number of register sets requested.

GPR3 = Address of word containing the allocation mask.

Return Codes: contained in GPR2 and GPR3

GPR2 — Return Codes

-1 = Resources unavailable.
-2 = Floating-Point Accelerator not installed.
0 = Successful completion.
20 = Protection violation storing into the allocation mask word.

GPR3 — Address of returned floating-point register sets mask

The word whose address is in GPR3 is modified to contain a mask representing the floating-point register sets allocated. If the Floating-Point Accelerator is not installed, GPR3 is not used.

Comments: If you change the contents of 0xD7 (which tells you which floating-point register set you are using), you must execute one of the execution control SVCs. Any of these instructions causes the VRM to examine the memory-mapped fields and update states, levels, and so on as required.

This SVC returns 20 whenever the virtual machine cannot store into the word addressed by GPR3. In this case, no register sets are allocated.

If the return code is zero, the requested number of register sets was allocated. If the return code is -1, fewer than the requested number of register sets were allocated.

For return codes 0 and -1, the word pointed to by GPR3 contains a bit mask indicating which register sets were allocated. Each 1 bit in the mask represents an allocated register set. The most significant bit corresponds to register set 0. The least significant bit corresponds to register set 31.

Dispatch SVC

Description: A virtual machine usually runs in the operating system state where certain SVCs are privileged. When the dispatching mechanism of an operating system needs to change to the problem state and give control to a program, it issues the **Dispatch SVC**.

SVC Code: 0x**FFFA**

Calling Register Conventions: none.

Return Codes: none.

Comments: The program status is set as if a **Return From Interrupt SVC** were issued on level 7.

Bits 0-9 of the ICS are set to zero, and the execution level is set to level 7. The updated ICS is stored in the word at location 0xE0, and the new level is stored in the halfword at location 0xE4.

The dispatched program can use the "Return SVC" on page 4-16 to return control to the operating system.

If this SVC needs to restart an exception reported as a program check or machine communications interrupt, the 20 or 36 bytes of restart data saved for the interrupt should be copied to location 0x2D4 before this SVC is executed. If the value of the byte at 0x2D4 is greater than two, an illegal operation code virtual program check is reported to the virtual machine.

The VRM sets the byte at 0x2D4 to zero as a result of this SVC.

Free Floating-Point Register Sets SVC

Description: This SVC frees floating-point register sets for a machine with the optional Floating-Point Accelerator installed.

SVC Code: 0x**FFAF**

Calling Register Conventions:

GPR2 = Bit mask indicating which register sets to free.

Return Codes: contained in GPR2

0 = Successful
4 = Attempt to free register sets not allocated.

Comments: Any time you change the floating-point register set allocated to a virtual machine, you must issue an execution control SVC (such as the **No Operation SVC**) to effect the change. Any of these instructions causes the VRM to examine the memory-mapped fields and update states, levels, and so on as required.

For systems without the floating-point accelerator, this SVC returns 0 in GPR2 only when GPR2 contained 0 when the SVC was issued; otherwise, GPR2 contains 4.

Return code 0 means all specified register sets were freed. Return code 4 means all specified register sets allocated to the virtual machine were freed. For example, if the virtual machine has register sets 5, 7, 8, 9, 12, and 15 and attempts to free sets 5, 6, 7, and 8, sets 5, 7, and 8 will be freed, but the virtual machine also receives a return code of 4.

The format of GPR2 on entry to this SVC matches the format of the mask returned from the **Allocate Floating-Point Register Sets SVC**.

No Operation SVC

Description: This SVC allows a virtual machine to invoke the VRM to update memory-mapped control information.

For example, if you change the floating-point register set assigned to your virtual machine, you can use the **No Operation SVC** to check and update the memory-mapped "virtual registers" of the virtual machine.

You might also use this SVC after changing any of the values in the VICS register.

SVC Code: 0x**FFD6**

Calling Register Conventions: none.

Return Codes: none.

Post SVC

Description: This SVC allows a virtual machine to send an event control bit (ECB) mask to a VRM process (such as a device manager). The mask conventions must be known by the cooperating virtual machine and VRM processes. Note that the VRM routine **_post** ("Post ECB (_post)" on page 5-37) reserves bits 24-31 of GPR3 for use by VRM queue management, and then assigns ECB numbers from the higher values (starting with bit 23) to the lower values (bit 0). Avoid using ECBs that are already defined by VRM queue management.

SVC Code: 0x**FFBA**

Calling Register Conventions:

GPR2 = ID of the path from the virtual machine to the VRM process.

GPR3 = 32-bit ECB mask that identifies an action known by both virtual machine and VRM process. See **Description** above and "Post ECB (_post)" on page 5-37 for restrictions on ECB mask assignment.

Return Codes: contained in GPR2

0 = Successful.

16 = Invalid path ID, or the path does not attach to a process.

Query Floating-Point Register Sets SVC

Description: This SVC returns a bit mask in GPR2 that indicates which floating-point register sets are allocated to the virtual machine making the query. You cannot use this SVC to query the floating-point register sets of another virtual machine.

SVC Code: 0x**FFB1**

Calling Register Conventions: none.

Return Codes: contained in GPR2

GPR2 contains a bit mask indicating the register sets allocated to the virtual machine making the query. When any GPR2 bits equal one, the corresponding floating-point register set is allocated to the virtual machine. For example, if GPR2 bits 0, 5, and 23 equal one, floating-point register sets 0, 5, and 23 are allocated to the virtual machine. If GPR2 contains only zeroes, no floating-point register sets are allocated to the virtual machine issuing the query, or the system does not have the floating-point accelerator option.

Return SVC

Description: A dispatched program returns control to the operating system by issuing the **Return SVC**.

The **Return SVC** can be issued in problem state.

SVC Code: 0x**FFF9**

Calling Register Conventions: none.

Return Codes: none.

Comments: **Return SVC** sets the program status as if a **Soft Interrupt SVC** were issued on level 7.

The old program status is stored in the level 7 program status block.

The **Return SVC** differs from **Soft Interrupt SVC** in that the interrupt is always taken immediately.

Return SVC sets the execution level to 7 and clears bits 0-9 of the ICS. The updated ICS bits 0-9 are stored in the halfword at location 0xE0, and the new level is stored in the halfword at location 0xE2.

Return From Interrupt SVC

Description: An interrupt handler in an operating system exits from the interrupt level with the **Return From Interrupt SVC**.

SVC Code: 0x**FFFC**

Calling Register Conventions: none.

Return Codes: none.

Comments: VRM resets the program status from the program status block for the current execution level as determined by the contents of the halfword at location 0xE4 in the virtual machine. If location 0xE4 has an invalid execution level value, the level 7 PSB is used. The IAR is replaced with the old IAR field, bits 16-31 of the ICS are replaced with bits 16-31 of the old ICS field, and the CS is replaced with the old CS field.

Bits 0-9 of the old ICS field are scanned to determine the new execution level. Bits 0-9 of the old ICS are replaced with the current contents of bits 0-9 of the ICS with the new level bit set to zero. The updated ICS bits 0-9 are stored in the halfword at location 0xE0, and the new level is stored in the halfword at location 0xE4.

If the execution level is -1 (machine communications) or -2 (program check), the VRM checks the byte at 0x28C or 0x2B0 to determine if there is restart data for an exception. If the first byte of the PSB extension for a machine communications or program check is not equal to zero, the VRM must restart a process interrupted by a previous exception.

When the SVC completes, the VRM sets the first byte of the appropriate restart area (0x28C or 0x2B0) back to zero.

If this SVC is issued from execution level 7, the level 7 PSB extension may be used to restart an exception. In this case, the byte at location 0x2D4 indicates the presence and amount of restart data. The following values are defined for 0x2D4:

- 0 = no restart data
- 1 = 20 bytes of restart data at 0x2D4
- 2 = 36 bytes of restart data at 0x2D4.

Again, when the SVC completes, the VRM sets the first byte of the restart area (0x2D4) back to zero.

If the value of 0x28C, 0x2B0, or 0x2D4 is greater than two, an illegal operation code virtual program check is reported to the virtual machine.

Set Interrupt SVC

Description: An operating system can enable or disable interrupts on the seven interrupt levels with the **Set Interrupt SVC**.

SVC Code: 0x**FFFE**

Calling register conventions:

GPR2 = Interrupt control options

Bit 6 = Change bit 30 of the ICS.
Bit 7 = Change bit 31 of the ICS.
Bit 14 = Value for bit 30 of the ICS.
Bit 15 = Value for bit 31 of the ICS.

Mask — Each bit corresponds to a bit in bits 16-23 of the ICS. If any of GPR2 bits 16-23 is a one, the value in the corresponding bit position of GPR2 bits 24-31 is taken and placed into the appropriate ICS bit position 16-23. For example, if GPR2 bit 18 (position 3) is a one, the value found in position 3 of GPR2 bits 24-31 (bit 26) is placed into ICS bit 18.

Value — Bits 24 through 31 contain the values for bits 16-23 of the ICS.

Return Codes: contained in GPR2

0 = Successful completion.

Set Timer SVC

Description: The **Set Timer SVC** maintains the virtual machine timer and associated virtual machine registers.

SVC Code: 0x**FFFF**

Calling register conventions:

GPR2 = Timer Options

Bit 0 =	The timer is enabled when bit 0 is set to one.
Bit 1 =	The timer is disabled when bit 1 is set to one.
Bit 2 =	The priority level and sublevel for timer interrupts are initialized when bit 2 is set to one.
Bit 3 =	The interval timer source register is changed or initialized when bit 3 is set to one. GPR3 must be loaded with the interval time value.
Bit 4 =	The current time for all virtual machines is set when bit 4 is set to one. GPR4 must be loaded with the time from midnight Jan. 1, 1970, in seconds.
Bit 5 =	The current time for only the requesting virtual machine is set when bit 5 is set to one. GPR4 must be loaded with the time from 01/01/70 in seconds.
Bit 6 =	Timer interrupts are disabled, and the timer interrupt level and sublevel values for this virtual machine are cleared to 0. Notice that the operation specified by this bit reverses the operation specified by bit 2.
Bit 7 =	The 60 hz extended time-of-day counter is enabled when bit 7 = 1.
Bit 8 =	The 60 hz extended time-of-day counter is disabled when bit 8 = 1.
Bits 9-20 =	Reserved.
Bits 21-23 =	The priority level for timer interrupts (binary 000 to binary 110).
Bits 24-31 =	The sublevel for timer interrupts (0 to 255).

GPR3 = Interval time value. The interval time value contains an integer number of timer units. A timer unit equals 16.6 milliseconds.

GPR4 = Current time. This field contains the number of elapsed seconds since January 1, 1970.

Return Codes: contained in GPR2

16 = Invalid interrupt specification, or attempt to enable timer with source bit (bit 3) set to zero.

Comments: Status and data values in the program status block when a timer interrupt occurs are:

Status Flags	Timer interrupt (bit 1 = 1).
Overrun Count	Number of timer overruns.
Status Word	Value in the virtual machine timer source register.
Data Words	0

The interval value in GPR3 should not exceed 2^{24}. The first 8 bits in the register are ignored.

Multiple actions can be initiated with one SVC. For example, the interval timer could be set up, the timer enabled, and current time of day set with one SVC call if the correct information is set up in the appropriate registers.

The set current time for requesting virtual machine option is not supported. Specifying this option sets the current time for all virtual machines.

The three timer states are:

- Disabled
- Enabled
- Enabled with interrupts.

If the timer is disabled, it does not count. If the timer is enabled, it is counting. In order to get interrupts, a valid interrupt level and sublevel must be specified or previously specified. The initial value for the real time of year and real time of IPL is initialized at IPL to the current value in the real time clock. All other values are initialized to zero at IPL with the timer disabled.

The value of the interval time is stored in the timer source register and is used as the counter value. If the timer is re-enabled it continues counting from when it was disabled. If a new value is entered for the interval time, the new value is stored into the timer source register.

When changing the status of the 60 hz extended time-of-day counter, the change is implemented on the next tick of the clock. When enabled, the counter may not start for up to 1 second. When disabled, the counter may not stop for up to 16.6 milliseconds.

Soft Interrupt SVC

Description: An operating system can initiate software interrupts on any of the seven interrupt levels with the **Soft Interrupt SVC**.

SVC Code: 0x**FFFD**

Calling Register Conventions:

GPR2 = Level — Contains the level to initiate a software interrupt in bits 13 through 15.

Sublevel — Contains the sublevel to initiate a software interrupt in bits 24 through 31.

GPR3 = Status Word — Contains the value to be placed in the status word in the program status block (offset 0x14) at the time of the interrupt.

GPR4 = Data Word — Contains the value to be placed in the first data word in the program status block (offset 0x18) at the time of the interrupt.

GPR5 = Data Word — Contains the value to be placed in the second data word in the program status block (offset 0x1C) at the time of the interrupt.

GPR6 = Data Word — Contains the value to be placed in the third data word in the program status block (offset 0x20) at time of interrupt.

Return Codes: contained in GPR2

16 = Level illegal or sublevel does not exist.

Comments: The status and data values in the program status block when a software interrupt occurs are:

Status Flags	Software interrupt (bit 0 = 1).
Overrun Count	0
Status Word	See GPR3
Data Words	See GPR4, GPR5, GPR6

The status and data words on hardware interrupts relieve the operating system from doing extra I/O to determine the cause of the interrupt.

The status and data words on software interrupts can emulate the hardware or provide a means of giving processing input with the interrupt.

Virtual Machine Wait SVC

Description: An operating system can put itself in a wait state by issuing the **Virtual Machine Wait SVC**.

SVC Code: 0x**FFFB**

Calling Register Conventions: none.

Return Codes: contained in GPR2

0 = Successful completion.

Comments: **Virtual Machine Wait SVC** puts the virtual machine in a wait state and sets the address of the next sequential instruction as the virtual machine's current IAR. ICS bit 31 is set to zero. On the next interrupt, VRM puts the old program status in the PSB.

The virtual machine timer changes a virtual machine from the wait state to the ready state.

Memory Management SVCs

The VRM creates a segment at virtual machine IPL time. The first page in the segment controls the interaction between the VRM and the virtual machine. This page is pinned in memory.

The VRM recognizes the following memory management SVCs from the virtual machine:

- Change Segment Size
- Clear Segment Registers
- Copy Segment
- Create Segment
- Destroy Segment
- Discard Page Range
- Load Segment Registers
- Map Page Range
- Pin Page Range
- Protect Pages
- Purge Page Range
- Purge Segments
- Query Page Protect
- Unmap Page Range
- Unpin Page Range.

All values in parameter structures for Memory Management SVCs are input parameters unless a superscript 0 (^{0}) follows the parameter name. These parameters are return parameters.

Do not try to alter the segment that contains the POST control block (PCB) with a memory-management SVC. The only memory-management SVCs that can be successfully issued for the PCB segment are **Load Segment Registers SVC** and **Query Page Protect SVC**.

Change Segment Size SVC

Description: The size of the named segment is changed to the size specified (or the next larger page multiple). The size may increase or decrease and may be either toward displacement 0 or maximum displacement, depending on the origin used when the segment was created. The size may be decreased to zero without destroying the segment. New pages, if any, are initialized to binary zeroes. Protection characteristics for any new pages are set according to the segment's protection.

SVC Code: 0x**FFEE**

Calling Register Conventions:

GPR2 = SegId. An unsigned integer that identifies the segment.

GPR3 = Size. The new size of the segment in bytes. If the size is not an integral multiple of the page size, the size is increased to the next multiple.

Return Codes: contained in GPR2

-1 = Insufficient resources.
0 = Successful completion.
8 = Segment does not exist.
12 = Invalid value for the size parameter.
14 = Segment ID is loaded in register 0 and the specified size is zero.
18 = I/O operation in progress to this segment and the size specified would decrease the segment size.

Comments: Pages mapped Read/Write or Write New are not paged-out as a result of this SVC. The virtual machine should first issue a **Purge Page Range SVC** to update the disk.

The contents of the segment, except for page 0, are undefined when GPR2 contains 14 on return.

Contents of a segment's pages deleted by a decrease operation are undefined when GPR2 contains 18. The virtual machine may retry this SVC after all the I/O operations have completed.

Clear Segment Registers SVC

Description: The **Clear Segment Registers SVC** clears the contents of one or more segment registers. You do not have to use this SVC before loading a new value into a register. The primary use of this SVC is to prevent a multitasking operating system from using a segment register loaded for another of its tasks.

SVC Code: 0x**FFE4**

Calling Register Conventions:

GPR2 = Segment register mask

This right-justified mask indicates the registers you want to clear. If you want to clear register 1, for example, GPR2 would contain the value 0x4000. To clear register 2, GPR2 must contain the value 0x2000.

A virtual machine cannot clear segment registers 0, 14, or 15. Any attempt to clear these registers is ignored.

Return Codes: contained in GPR2

0 = Successful completion.

Copy Segment SVC

Description: This SVC creates a new segment that is a copy of an existing segment. The protection characteristics of the new segment match those of the original. The new segment's identifier is returned in the ToSegID parameter.

SVC Code: 0x**FFEF**

Calling Register Conventions:

GPR2 = Contains a 32-bit address pointing to a 4-byte word-aligned structure containing parameters.

The parameter structure is:

Field	Bytes
FromSegID	bytes 0-1
ToSegID [0]	bytes 2-3

The parameters are defined as follows:

- FromSegID — An unsigned integer identifying the source segment.
- ToSegID — An unsigned integer identifying the destination segment.

Return Codes: contained in GPR2

- -1 = Insufficient resources or all available segment IDs in use.
- 0 = Successful completion.
- 8 = Segment does not exist in this virtual machine.
- 20 = Invalid parameter list address.

Comments: The VRM does not protect the virtual machine from copying segments that contain pages mapped Read/Write or Write New.

Also, pages pinned in the source segment are not automatically pinned in the destination segment.

Create Segment SVC

Description **Create Segment SVC** creates a segment and returns the segment ID (parameter SegId). The segment is initialized to binary zeroes. Page-level protection is specified for the entire segment according to the protection parameter. The **Protect Pages SVC** may be used to affect page-specific protection.

SVC Code: 0x**FFF2**

Calling Register Conventions:

GPR2 = Contains a 32-bit address pointing to an 8-byte word-aligned structure containing parameters.

The parameter structure is:

Field	Bytes
SegId	bytes 0-1
Inverted	byte 2
Protection	byte 3
Size	bytes 4-7

The parameters are defined as follows:

- SegId — This returned value is an unsigned integer that identifies the segment.
- Inverted — A bit indicating, if set, that the segment begins at the maximum displacement value, 2^{28}-1, rather than at displacement 0. If this bit is set and the segment grows, it grows toward displacement 0. Similarly, if the segment shrinks, it shrinks toward the maximum displacement.
- Protection — Segment protection
  - 0 = No access in unprivileged state; read/write access in privileged state.
  - 1 = Read only access in unprivileged state, read/write access in privileged state.
  - 2 = Read/write access in both unprivileged and privileged states.
  - 3 = Read only access in both unprivileged and privileged states.
- Size — The size of the segment in bytes. If the size is not an integral multiple of the page size (2048 bytes), the size is increased to the next multiple.

Return Codes: contained in GPR2

-1 =	Insufficient resources or all available segment IDs in use.
0 =	Successful completion.
12 =	Invalid parameter (inverted, protection, or size)
20 =	Invalid parameter list address.

Destroy Segment SVC

Description: This SVC destroys the specified segment. If the segment ID is loaded in a segment register, the register is cleared.

If the segment was being shared, either with another virtual machine or with a VRM component, the segment is still destroyed. Subsequent use of the segment results in a program check interrupt.

SVC Code: 0x**FFF1**

Calling Register Conventions:

GPR2 = SegId

Contains the unsigned integer that identifies the segment to be destroyed.

Return Codes: contained in GPR2

0 =	Successful completion.
8 =	Segment does not exist.
14 =	Segment ID is loaded in register 0.
18 =	Segment was not destroyed due to I/O operation in progress.

Comments: Pages mapped Read/Write or Write New are not paged-out to the specified minidisk as a result of this SVC. The virtual machine should first issue a **Purge Page Range SVC** to update the disk.

Contents of the specified segment are undefined when the VRM returns code 18. The virtual machine may retry this SVC after all I/O operations have completed.

The contents of the segment, with the exception of page 0, are undefined when GPR2 contains 14 on return.

Discard Page Range SVC

Description: This SVC synchronizes the contents of a range of pages with the contents of the fixed disk that corresponds to the page range. It should be used when the contents of the fixed disk to which a page range is mapped has been updated with a **Start I/O SVC**.

The **Discard Page Range SVC** works only for segments mapped with the **Map Page Range SVC**. If issued on unmapped segments, this SVC returns zero but does nothing.

For mapped pages, the **Discard Page Range SVC** works as follows:

- Copy-on-Write

  Page frames for pages mapped copy-on-write are discarded only if they have not been altered since mapped.
- Read/Write

  Page frames for pages mapped read/write are discarded only if they have not been altered since the last page-in by the virtual memory manager.
- Write New

  Page frames for pages mapped write new are discarded only if:
  - The virtual memory manager writes the page to disk after it is mapped.
  - The page has not been altered since it was written.

Note that this SVC does not release page frames that have been altered.

Subsequent references to a virtual address in this range result in a disk read (page-in) from the corresponding fixed-disk address for pages that were discarded.

This SVC should not be used on segments that have been copied with the **Copy Segment SVC**.

SVC Code: 0x**FFD2**

Calling Register Conventions:

GPR2 = SegId

This register contains the unsigned integer that identifies the subject segment.

GPR3 = First page

This register contains an integer that indicates the first page of the range.

GPR4 = Number of pages

This register contains an integer that indicates how many pages are in the range.

Return Codes: contained in GPR2

0 =	Successful completion.
8 =	Segment does not exist.
12 =	First page or number of pages invalid.
14 =	Segment ID is loaded in register 0 and page 0 was included in the range.
18 =	I/O in progress to the specified page range or the page range contains pinned pages.

Comments: Return code 18 indicates that one or more of the pages in the range are currently pinned due to I/O activity or the **Pin Page Range SVC**. The virtual machine can retry this SVC after all I/O is complete or the **Unpin Page Range SVC** is issued, or both.

Load Segment Registers SVC

Description: The **Load Segment Registers SVC** loads one or more segments into segment registers. Specified bit settings for page level privilege are also indicated.

SVC Code: 0x**FFE5**

Calling Register Conventions:

GPR2 = Points to a word-aligned structure containing parameters. The size of the structure is 4 times Number+4 bytes.

The parameter structure is:

Field	Bytes
Number	bytes 0-3
SegID	bytes 4-5 (first register loaded)
SegReg	byte 6
Protection	byte 7
. . .	
SegID	bytes 4*(#-1)+4 through 4*(#-1)+5 (last)
SegReg	byte 4*(#-1)+6
Protection	byte 4*(#-1)+7

The parameters are defined as follows:

- Number — The number of segments and/or registers to be loaded.
- SegId — An unsigned integers that identifies the segment. A segment ID of zero may be used to indicate that a segment register is loaded with an undefined value, effectively clearing the register. (A virtual machine is never allowed to have zero for one of its segment IDs.)
- SegReg — 4-bit integers identifying the register.
- Protection — Segment protection. Possible values are:

  0 = Page level protection with unprivileged bit cleared.
  1 = Page level protection with unprivileged bit set.
  3 = Page level protection with unprivileged bit tied to virtual machine operating system/problem state. When the virtual machine is in OS state, the unprivileged bit is cleared. When the virtual machine is in problem state, the unprivileged bit is set.

Return Codes: contained in GPR2

0 = Successful

Comments: A virtual machine may not change the segment identifier loaded into segment register 0. It can load segment register 0 to change its protection. A virtual machine may never load segment registers 14 or 15, however.

The VRM does not update the segment register when an invalid segment register number or protection is specified.

Because this SVC rounds the parameter address down to the nearest word-aligned address before using it, you must ensure that the structure is properly aligned.

Map Page Range SVC

Description: The **Map Page Range SVC** allows a virtual machine to establish a mapping between a range of pages in a segment and blocks on a minidisk. The pages are a contiguous set of virtual addresses, but the blocks are not necessarily contiguous on the minidisk. The block ranges must hold an integral multiple of one or more pages. Also, the area covered by the sum of the block ranges must be equal to the area covered by the range of pages.

All pages in the range must already exist in the segment; the segment is not automatically extended.

The page range specification is the same regardless of whether the segment is inverted.

The VRM supports paging activity on up to 64 minidisks at a time. This includes paging minidisks and any minidisks that have segments mapped to them. If you exceed this limit, you will receive a return code of -1 (insufficient resources).

Note: You can map pages only to minidisks that support bad block relocation and have a block size defined as 512 bytes.

SVC Code: 0x**FFC3**

Calling Register Conventions:

GPR2 = Contains a 32-bit address pointing to a variable-length word-aligned structure containing parameters.

The parameter structure is shown in the following figure.

Field	Bytes
SegID	bytes 0-1
Mode	byte 2
Reserved	byte 3
FirstPage	bytes 4-7
NumberPages	bytes 8-11
IODN	bytes 12-13
NumberRanges	bytes 14-15
StartBlock	bytes 16-19 (block 1)
NumBlocks	bytes 20-23
. . .	
StartBlock	bytes n-(n+3) (block n)
NumBlocks	bytes (n+4)-(n+7)

The parameters are defined as follows:

- SegId — This unsigned value identifies the segment.
- Mode — Options available:

  1 = Write New
  2 = Read/Write (Update)
  3 = Copy on Write

- The next byte is reserved and must be set equal to zero.
- FirstPage — This integer identifies the first page in the range (offset into the segment).
- NumberPages — This integer specifies the number of pages in the range.
- IODN — This 16-bit integer specifies the minidisk's IODN.
- NumberRanges — This integer indicates how many page ranges are specified in this structure.

- StartBlock — This integer specifies the offset of the first block in a range into the minidisk specified by the IODN. Each starting block must start on a 2K-byte boundary. Therefore the two least significant bits of the starting block must be set equal to zero.
- NumBlocks — This integer specifies the number of blocks in the range. The number of blocks must be an integer multiple of 2K increments.

Return Codes: contained in GPR2

-1 = Insufficient resources.
0 = Successful completion.
8 = Invalid segment identifier.
12 = Starting Block or Number of Blocks values are invalid.
12 = The mode is out of range.
12 = FirstPage or NoPages values are invalid.
12 = Block range doesn't match page range.
14 = Segment ID 0 is loaded in register 0 and page 0 was included in the range.
18 = I/O in progress to the specified page range or the page range contains pinned pages.
20 = Invalid parameter list address.
24 = Virtual machine unable to open the minidisk.
24 = Virtual machine opened the minidisk with insufficient privilege.
32 = Invalid minidisk IODN.
64 = Blocks out of range (exceed the end of the minidisk).

Comments: The VRM checks access authority to the minidisk only when the pages are mapped. The virtual machine issuing this SVC must have the specified minidisk open to map a file copy on write. Also, it must have the specified minidisk open for write access to map a file write new or read/write. All virtual machines sharing the segment have the same access rights as the virtual machine that mapped the pages. If a virtual machine requires exclusive use of a minidisk, the machine must open the minidisk for exclusive use, keep the minidisk open while it maps the pages, an not share the mapped segments.

The virtual machine must unmap or destroy the segments before it closes the minidisk, because the VRM does not automatically unmap the pages when the minidisk is closed.

Pages mapped read/write or write new are not paged out to the specified minidisk as a result of this SVC. The virtual machine should first issue a **Purge Page Range SVC** to update the disk.

GPR2 contains 18 on return only if one or more of the specified pages are pinned due to I/O activity or the **Pin Page Range SVC**. The pages specified prior to the pinned page are mapped, all others are not mapped. The virtual machine can retry this SVC after all the I/O operations have completed or the pages are unpinned with the **Unpin Page Range SVC**.

Pin Page Range SVC

Description: The **Pin Page Range SVC** ensures that a page fault does not occur for a range of pages. The function is always performed synchronously. If a page is not in primary memory, it is brought into memory. A count is kept for each page of the number of times this function has been performed for this page without a corresponding **UnPin Page Range SVC** for the page. If two virtual machines ask that a page be pinned and one asks that it be released, the page remains pinned until the second virtual machine asks for release. A call for already pinned pages has no effect on those pages except to increase the counter.

SVC Code: 0x**FFE3**

Calling Register Conventions:

GPR2 = SegId

This value identifies the segment.

GPR3 = FirstPage

This unsigned integer identifies the first page of the range.

GPR4 = NoPages

This unsigned integer specifies the number of pages in the range.

Return Codes: contained in GPR2

- -1 = Insufficient resources or the page is already pinned 255 times.
- 0 = Successful completion.
- 8 = Segment does not exist.
- 12 = Invalid FirstPage or NoPages values.

Comments: Each pinned page has an associated count in the range 0-255. This count is incremented by a **Pin Page Range SVC** and decremented by an **Unpin Page Range SVC**. A page is unpinned when the count changes from 1 to 0. The count can be no greater than 255 and no less than 0 for this instruction to work properly. All virtual machines that share the segment share the same count field.

Protect Pages SVC

Description: You can change the protection characteristics of a range of pages with this SVC.

SVC Code: 0xFFEA

Calling Register Conventions:

GPR2 = SegId

The unsigned integer that identifies the segment.

GPR3 = FirstPage

The page index (displacement/2048) of first page

GPR4 = NoPages

The number of pages affected.

GPR5 = Protection

0 = No access in unprivileged state; read/write access in privileged state.
1 = Read only access in unprivileged state; read/write access in privileged state.
2 = Read/write access in both unprivileged and privileged states.
3 = Read only access in both unprivileged and privileged states.

Return Codes: contained in GPR2

0 = Successful completion.
8 = Segment does not exist.
12 = FirstPage, NoPages, or protection parameters invalid.
14 = Segment ID is loaded in register 0 and an invalid Protection setting was given for page 0.
18 = I/O in progress to this segment, protection not changed.

Comments: The virtual machine must ensure that all pages changed by an I/O read operation have read/write privileged page protection.

Return code 18 indicates that protection was not changed for one or more of the pages because I/O was in progress to the pages. Protection was changed for the other pages in the range.

Purge Page Range SVC

Description: The **Purge Page Range SVC** is used to write a range of pages in a segment to backing store. You can use this SVC with the following options for notification of I/O completion:

- Asynchronous, no completion notification
- Synchronous
- Asynchronous, notify on completion

Regardless of the notification method selected, the VRM writes any changed pages in the range to backing store.

This SVC is not an atomic operation, so the VRM does not ensure that the pages are consistent on the fixed disk in the case of multiple writers or non-synchronous operation.

You can also specify whether the page frames written to the fixed disk should be released by the virtual memory manager when the I/O is complete. This allows you to distinguish whether this SVC is just updating backing store or is swapping a portion of a segment out of primary memory.

SVC Code: 0x**FFE0**

Calling Register Conventions:

GPR2 = SegId

This unsigned integer identifies the segment.

GPR3 = FirstPage

This integer identifies the first page of the range.

GPR4 = NoPages

This integer gives the number of pages in the range.

GPR5 = Option flags

This register contains options flags in bits 0 and 30-31. Bits 1-29 are reserved (set to zero).

- Bit 0 - release page frames on I/O completion

  When bit 0 is set to 0, the virtual memory manager releases all in-memory page frames specified in the call. The page frames are released immediately if they are unmodified, and after the I/O is complete if they are modified.

  When bit 0 is set to 1, the virtual memory manager does not release any of the page frames, but writes the modified page frames to backing store and leaves the pages in memory.

- Bits 30-31 - notification type

  0 = Asynchronous, no notification
  1 = Synchronous
  2 = Asynchronous, notify on completion

GPR6 = Notify ID address

This value is a halfword-aligned address where the unique identifier assigned by this SVC is returned if you request the asynchronous notify option. The identifier is a halfword value. If the value of the identifier is 0xFFFF, no notification will be sent. All other values indicate that I/O was required and notification will be sent.

The virtual machine receives a single machine communications interrupt specifying the identifier after all pages have completed page-out I/O. The virtual machine is allowed to proceed while the page-outs take place. The completion interrupt is sent to the virtual machine even if paging interrupts are disabled.

Return Codes: contained in GPR2

0 = Successful completion.
8 = Segment does not exist.
12 = FirstPage, NoPages, or options out of range.
14 = Segment ID is loaded in register 0 and page 0 was included in range.
18 = I/O in progress to this segment or the segment contains pinned pages.
20 = Invalid notify ID address.

Comments: Return code 18 indicates that one or more of the changed pages are currently pinned due to I/O activity or the **Pin Page Range SVC**. The virtual machine can retry the SVC after all I/O operations complete or the pages are unpinned with the **Unpin Page Range SVC**, or both.

Purge Segments SVC

Description: The **Purge Segments SVC** writes one or more segments to backing store. You can use this SVC with the following options for notification of I/O completion:

- Asynchronous, no completion notification
- Synchronous
- Asynchronous, notify on completion

Regardless of the notification method selected, the VRM writes any changed pages in the range to backing store.

This SVC is not an atomic operation, so the VRM does not ensure that the pages are consistent on the fixed disk in the case of multiple writers or non-synchronous operation.

SVC Code: 0x**FFD3**

Calling Register Conventions:

GPR2 = Number

This unsigned integer indicates the number of segments to purge.

GPR3 = SegID Array

This integer contains a pointer to a halfword-aligned structure containing a list of segment IDs to purge. Each entry in this structure is a two-byte segment ID.

GPR4 = Option flags

This register contains options flags in bits 0 and 30-31. Bits 1-29 are reserved (set to zero).

- Bit 0 - release page frames on I/O completion

  When bit 0 is set to 0, the virtual memory manager releases all in-memory page frames specified in the call. The page frames are released immediately if they are unmodified, and after the I/O is complete if they are modified.

  When bit 0 is set to 1, the virtual memory manager does not release any of the page frames, but writes the modified page frames to backing store and leaves the pages in memory.

- Bits 30-31 - notification type

  0 = Asynchronous, no notification
  1 = Synchronous
  2 = Asynchronous, notify on completion

GPR5 = Notify ID address

This value is a halfword-aligned address where the unique identifier assigned by this SVC is returned if you request the asynchronous notify option. The identifier is a halfword value. If the value of the identifier is 0xFFFF, no notification is sent. All other values indicate that I/O was required and notification will be sent.

The virtual machine receives a single machine communications interrupt specifying the identifier after all pages have completed page-out I/O. The virtual machine is allowed to proceed while the page-outs take place. The completion interrupt is sent to the virtual machine even if paging interrupts are disabled.

Return Codes: contained in GPR2

0 = Successful completion.
8 = Invalid segment identifier.
12 = Invalid option.
14 = One of the input segment IDs is loaded in register 0.
18 = I/O in progress to one of the segments.
20 = Invalid notify ID address, or invalid segID list address.

Comments: Return code 18 indicates that one or more of the changed pages are currently pinned due to I/O activity or the **Pin Page Range SVC**. The virtual machine can retry the SVC after all I/O operations complete or the pages are unpinned with the **Unpin Page Range SVC**, or both.

Query Page Protect SVC

Description: You can determine the page-level protection of a page with the **Query Page Protect SVC**.

SVC Code: 0xFFE9

Calling Register Conventions:

GPR2 = Contains a word-aligned 32-bit address pointing to a 7-byte word-aligned structure containing parameters.

The parameter structure is shown in the following figure. A superscript 0 (^{0}) indicates a returned value.

Field	Bytes
Page	bytes 0-3
SegID	bytes 4-5
Protection ^{0}	byte 6

The parameters are defined as follows:

- Page — Index (displacement/2048) of page
- SegID — Subject segment ID
- Protection — Read/write protection

  0 = No access in unprivileged state; read/write access in privileged state.

  1 = Read-only access in unprivileged state; read/write access in privileged state.

  2 = Read/write access in both privileged and unprivileged states.

  3 = Read-only access in both privileged and unprivileged states.

Return Codes: contained in GPR2

0 = Successful completion.
8 = Segment does not exist.
12 = Page out of range.
20 = Invalid parameter list address.

UnMap Page Range SVC

Description: The **UnMap Page Range SVC** unmaps pages mapped by the **Map Page Range SVC**. The specified pages are not written to their respective pages in the virtual machine's file system after this SVC executes. The contents of the specified pages are undefined.

SVC Code: 0x**FFB2**

Calling Register Conventions:

GPR2 = SegID

This integer identifies the segment.

GPR3 = FirstPage

This integer identifies the first page of the range.

GPR4 = NoPages

This integer gives the number of pages in the range.

Return Codes: contained in GPR2

0 = Successful completion.
8 = Segment does not exist in this virtual machine.
12 = FirstPage, page range, or both are invalid.

Comments: Pages mapped read/write or write new are not paged-out to the specified minidisk as a result of this SVC. The virtual machine should first issue a **Purge Page Range SVC** to update the disk.

UnPin Page Range SVC

Description: The **UnPin Page Range SVC** decreases the counter of pin page range calls for each page in the range. If the counter becomes zero and you try to unpin a page range, you will receive a -1 return code from the SVC.

SVC Code: 0x**FFE2**

Calling Register Conventions:

GPR2 = SegId

This integer identifies the segment.

GPR3 = FirstPage

This integer identifies the first page of the range.

GPR4 = NoPages

This integer gives the number of pages in the range.

Return Codes: contained in GPR2

-1 = Pin count already zero.
0 = Successful completion.
8 = Invalid segment identifier.
12 = Invalid FirstPage or NoPages values.

Input/Output SVCs

The VRM recognizes the following input/output SVCs from the virtual machine:

- Attach Device
- Cancel I/O
- Define Code
- Define Device
- Detach Device
- Query Device
- Ring Queue Get Word
- Ring Queue Put Word
- Send Command
- Start I/O.

Attach Device SVC

Description: The **Attach Device SVC** establishes linkage from an operating system to a device. Some devices may be shared by two or more virtual machines. Each virtual machine using a shared device must be attached to that device. The **Attach Device SVC** is a synchronous operation.

SVC Code: 0x**FFF7**

Calling Register Conventions:

GPR2 = IODN

This register contains the input/output device number (IODN) in bits 16 through 31.

Level/Sublevel — The interrupt level and sublevel to be used when notifying the virtual machine of an interrupt from this device are contained in bits 5 through 7 and bits 8 through 15, respectively.

GPR3 = Maximum Number Of Interrupts

This register contains the maximum number of unsolicited interrupts allowed from this device before an overrun occurs. Only values from 0 to 15 are allowed.

GPR4 = Path identifier

This register contains the address of the returned path identifier. In this case, the path that connects the virtual machine to the device is returned.

Return Codes: contained in GPR2

- -1 = Insufficient resources.
- 0 = Successful completion.
- 4 = Device is currently attached to this virtual machine.
- 8 = Device is already attached to maximum number of virtual machines, or is being used by the coprocessor.
- 12 = The address in GPR4 is not word-aligned.
- 16 = Invalid IODN specified
- 20 = Invalid interrupt level/sublevel specified.
- 24 = Invalid pending interrupt maximum.

Cancel I/O SVC

Description: The **Cancel I/O SVC** is used to purge queued work requests. All operations currently queued to the specified device can be cancelled, or all interrupts currently queued to this operating system from the specified device can be cancelled, or both. An interrupt from the specified device in progress is allowed to complete.

The **Cancel I/O SVC** is considered to be synchronous, although an I/O operation in progress is not immediately halted.

SVC Code: 0x**FFF3**

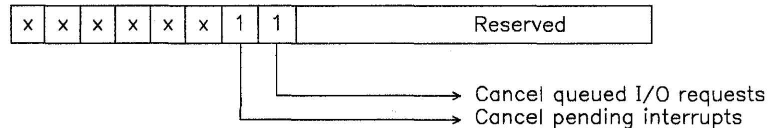
Calling Register Conventions:

GPR2 = IODN

This register contains the input/output device number.

GPR3 = Cancel I/O options

The **Cancel I/O SVC** options are defined as follows:



where:

- Cancel queued I/O requests — Bit 6 is set to cancel all I/O operations from this operating system to the specified device.
- Cancel pending interrupts — Bit 7 is set to cancel all pending interrupts for this operating system from the specified device. This option also cancels the interrupts that result from cancelling the queued I/O requests (Bit 6).

Cancelled I/O requests do not generate interrupts. However, GPR2 will contain the number of requests and/or interrupts cancelled.

GPR4 = Path identifier

Return Codes: contained in GPR2

n = number of operations cancelled.

Comments: Interrupts generated as a result of cancelling queued I/O requests have an operation result of 0x8000. The operation identification information describes the cancelled operation, and all other operation completion information is zero.

Define Code SVC

Description: The **Define Code SVC** allows you to add code to the VRM to support unique devices. The code must be in the VRM-compatible module format. Also, the module must conform to the VRM runtime environment. You must first load the module into memory and then execute **Define Code SVC.** The SVC then relocates the component into a special VRM address space. The module must be designed to be relocatable.

You must assign a unique input/output code number (IOCN) to your module. IOCNs 0 through 1,023 (inclusive) are reserved for devices defined at VRM IPL time. If you select a reserved IOCN for any code you may install into the VRM, you will receive an error return code of 16.

This SVC is synchronous and privileged to the operating system state.

SVC Code: 0x**FFD9**

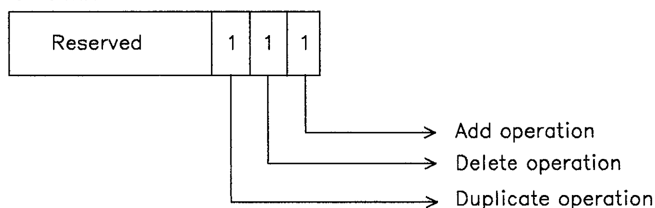
Calling Register Conventions:

GPR2 = IOCN

The input/output code number (IOCN) for the optional component occupies bits 16 through 31. This number is used by the **Define Device SVC** to associate the device code with the device.

GPR3 = Define options

The define options for this service occupy bits 16 through 31. This parameter identifies the requested code definition option. The parameter format is as follows:



The parameters are defined as follows:

- Add — This option is used to add code to the VRM. The IOCN must not already exist to use this option.
- Delete — This option is used to delete code from the VRM. The IOCN must already exist to use this option.

- Duplicate — This option copies part of the existing component specified by the IOCN. In effect, you create an additional IOCN for an existing module. Thus, two components with separate IOCNs can share part of a module, minimizing the resources required to support both components.

GPR4 = Module start address

This register contains the start address for the page-aligned module to be added to the VRM. The component code must be contiguous in virtual memory, starting at the specified address. This parameter is valid only when you specify the add option.

A shared module is indicated by the duplicate option. In this case, GPR4 (16-31) contains the IOCN of the module to be copied to create the new component. This parameter is valid only when you specify the duplicate option.

GPR5 = Module length

This register indicates the length of the module you want to add to the VRM. This parameter is valid only when you specify the add option.

Return Codes: contained in GPR2

-1 = Insufficient resources.
0 = Successful completion.
12 = Not added, invalid start address specified.
16 = Invalid IOCN specified.
24 = Invalid option specified.
28 = Not added, invalid module format or data.
32 = Not deleted, in use by one or more VRM components.

Comments: The VRM takes the real resources (page frames and page space) that contain the module and effectively clears the virtual memory that contains the module. Therefore, the module must be aligned at a page (2K byte) boundary and not mixed in a page with other data.

The read/write section of the module must not have pages protected read-only in both privileged and unprivileged state.

The module defined with this SVC cannot be mapped to a minidisk because this SVC does not support modules that are mapped files.

Define Device SVC

Description: The **Define Device SVC** lets you add a device to the system. This instruction is synchronous and privileged to the operating system state.

For most devices, you specify a unique input/output device number (IODN) as an input parameter. When a device manager creates an instance of a virtual device, however, the IODN is returned.

If you select an IODN that is reserved or that does not match your device type, you will receive an error return code of 16.

The following table defines IODN conventions.

Query All	IODN 0 is reserved for the query-all form of the **Query Device SVC**. No device can have this IODN.
Drivers/Managers	IODNs in the range from 1 through 16,383 are reserved to identify device drivers and device managers. These IODNs are provided as input to the **Define Device SVC**. IODNs 1 through 1,023 (inclusive) are reserved for devices defined at VRM IPL time.
Minidisks	IODNs in the range from 16,384 through 32,767 are reserved to identify minidisks. These IODNs are provided as input to the minidisk manager.
Virtual devices	IODNs in the range from 32,768 through 65,535 are reserved to identify virtual devices. Here the IODN is returned by the device manager that creates the virtual device.

SVC Code: 0x**FFF8**

Calling Register Conventions:

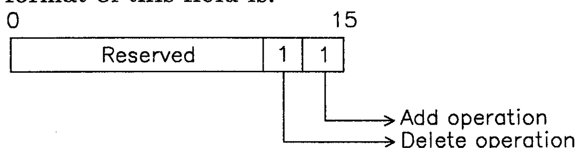
GPR2 = This register contains the address of the define device structure as shown in Figure 4-2 on page 4-52. This structure must be aligned on a fullword boundary.

Offset		
0	IODN	IOCN
4	Define options	Device type
8	Device name	
12	Reserved	
16	Offset to hardware characteristics	
20	Offset to device characteristics	
24	Offset to error log	
28	Device-dependent information	
N		

Figure 4-2. Define Device Structure — Header

The fields within the define device structure are:

- IODN — A 16-bit I/O device number the VRM uses to reference this device for all SVCs issued at the VMI.
- IOCN — A 16-bit I/O code number used to link this device to the code supporting it.
- Define options — A 16-bit field identifying the requested definition option. The format of this field is:

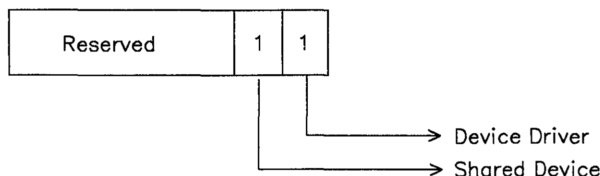


  where:

  – Add — This option is used to add a device definition to the VRM. The IODN must not already exist to use this option.

  – Delete — This option is used to delete a device definition from the VRM. The IODN must already exist to use this option.

- Device type — This 16-bit field identifies the characteristics of the device being defined. The format of this field is shown in the following figure.

  where:

  - Device Driver — A one in this bit specifies a device driver. A zero indicates a device manager.
  - Shared Device — A one in this bit specifies that the device can be shared by multiple virtual machines. A zero indicates a nonshared device, which may be attached to only one virtual machine at a time.

- Device Name — This is a 32-bit field ignored by the VRM. It is saved, unmodified, in the query device structure associated with the device. It can be used to create conventional names that identify devices.
- Offset to Hardware Characteristics — This is a 32-bit field that specifies the offset from the start of the Define Device Structure to the device's hardware characteristics. A value of zero indicates that the device does not have a hardware characteristics section. Only device drivers have such a section.
- Offset to Device Characteristics — This is a 32-bit field that specifies the offset from the start of the Define Device Structure to the device characteristics. A value of zero indicates that the device does not have a device characteristics section.
- Offset to Error Log — This is a 32-bit field that specifies the offset from the start of the Define Device Structure to the error log section. A value of zero indicates that the device does not have an error log section.
- Device-dependent Information — This section is not used by device managers, but is required for device drivers. These fields may include the device's hardware characteristics, device characteristics and error log. All device types do not require all of these fields. For example, several devices have no error log sections. All device drivers must have a hardware characteristics field. (See Figure 4-3 on page 4-54.)

  Device characteristics are the parameters that control a device. Because different devices need different control, the fields in this section and the length of the section vary from driver to driver.

  An Error Log section is kept for all devices available from IBM, but is optional for devices you may install. This field may be used to monitor the number and severity of certain error conditions.

Each of these sections must start on a word boundary and the first 4 bytes in each of these areas is its length in words. No free space is allowed between any of these sections.

Return Codes: contained in GPR2

-1 = Insufficient resources
0 = Successful completion.
8 = Add and Delete not allowed in the same request.
12 = DDS not on word boundary or invalid address.
12 = DDS offsets invalid.
12 = DDS not accepted by device.
16 = Invalid IODN specified.
20 = Invalid IOCN specified.
24 = Not deleted, attached by one or more users.

Comments: Note that this service does not create additional copies of components. The same IOCN should be specified for multiple devices only when each of the devices is connected to the same adapter or the component is coded to be re-entrant.

Device drivers require a hardware characteristics section. Figure 4-3 describes the format of this section.

Offset	Field
0	DDS header
28	Length (in words) of hardware characteristics
32	Internal device type
36	I/O port address (base)
40	I/O port addresses (number)
44	Bus memory start address (RAM)
48	Bus memory end address (RAM)
52	DMA type
56	One interrupt definition (16 bytes) for each interrupt level supported by the device
N	Bus memory start address (ROM)
N+4	Bus memory end address (ROM)
N+8	

Figure 4-3. Define Device Structure — Hardware Characteristics

The hardware characteristics fields are defined on the following page.

- Length — Specifies the length of the hardware characteristics in words. This value is determined by the following formula:

  ((28/4) + (4* Number of Interrupts)
  (+ 2 for devices with ROM addresses)

- Internal device type — Specifies the type of device. If bit 0 of this word is set to 0, the device has ROM addresses that are specified after the interrupt definition fields. The format of the internal device type field is defined as follows:

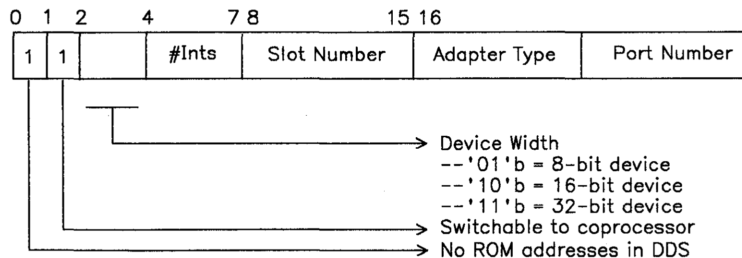


Figure 4-4. Internal Device Type

- DMA type — Specifies the device's DMA support. The format of this field is defined as follows:

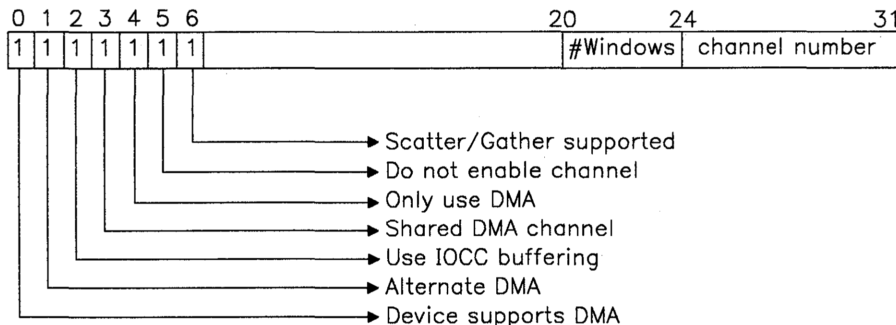


Figure 4-5. DMA Type

- Interrupt Type — Specifies the device's interrupt support. The class field determines the relative importance of the device's interrupt. Each interrupt level may have only one associated class. The classes are defined as follows:

  0 = Devices that overrun but have no recovery mechanism.
  1 = Devices that overrun but can recover.
  2 = Performance-sensitive devices that do not overrun.
  3 = Devices that do not overrun and are not performance-sensitive.

The format of the interrupt type field is as follows:

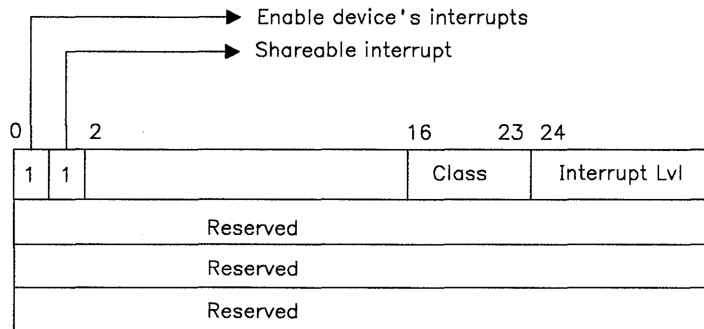


Figure 4-6. Interrupt Type

The Define Device Structure for a delete operation must be at least 28 bytes long. The fields in the first 6 bytes must correctly describe the device to be deleted.

Detach Device SVC

Description: The **Detach Device SVC** terminates linkage from an operating system to a previously attached device. The **Detach Device SVC** is a synchronous operation.

SVC Code: 0x**FFF6**

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the device from which to detach.

GPR3 = Path identifier

This register contains the identifier of the path that connects the virtual machine to the device.

Return Codes: contained in GPR2

0 = Successful completion.
16 = Invalid path identifier specified.

Comments: This service blocks the virtual machine until the I/O currently enqueued to the device is complete. A multitasking operating system may not want to issue this command until all I/O for the device is complete.

Query Device SVC

Description: The **Query Device SVC** is used to obtain information concerning a device or I/O operations associated with the specified device. An IODN referenced in the query structure and equal to zero is interpreted as a request for all IODNs currently defined. This SVC is a synchronous operation.

SVC Code: 0x**FFF5**

Calling Register Conventions:

GPR2 = Contains the virtual address of a query device structure (QDS). The QDS must be aligned on a fullword boundary.

The format of the QDS is defined as follows on input to the **Query Device SVC**:

Offset		
0	IODN	Query Options
4	QDS Length	
8	Remainder of Query buffer :	
N		

The fields within a QDS are defined as follows:

- Input/Output Device Number (IODN) — The IODN is used by the operating system to specify a particular I/O device.
- Query options — This field defines what device information is to be returned. Possible values for this field are:

  0xnnn1 = Returns the hardware characteristics.
  0xnnn2 = Returns the device characteristics.
  0xnnn4 = Returns the error log.
  0xnnn7 = Returns all of the above.

- QDS length — This field contains the length in bytes of the rest of the QDS. The total length of the QDS is this value plus 8.

On return, the QDS is of the same format as the define device structure in Figure 4-2 on page 4-52. The first 28 bytes of the QDS contain the information defined for the first 28 bytes of the DDS. The define options have been replaced by the following information:

0xnnn1 = Hardware characteristics returned.
0xnnn2 = Device characteristics returned.
0xnnn4 = Error log returned.
0xnnn7 = Returns all of the above.

The QDS starting at offset 28 contains the operation completion information for the last operation completed by the device for this virtual machine. The information requested by the query options comes next.

The general format of the operation completion information is shown in the following chart. Refer to the "Send Command SVC" on page 4-63 and "Start I/O SVC" on page 4-67 sections for the exact data in these fields.

Offset			
28	Reserved	Status Flags	Overrun Count
32	Status Word		
36	Data Word 1		
40	Data Word 2		
44	Data Word 3		
48			

Figure 4-7. Operation Completion Information

The operation completion information provided by the VRM is defined as follows:

- Status Flags — Save the status flags from the operation most recently completed.
- Status Word — Save the status word from the operation most recently completed.
- Data Word 1 — Save the first data word from the operation most recently completed.
- Data Word 2 — Save the second data word from the operation most recently completed.
- Data Word 3 — Save the third data word from the operation most recently completed.

Information describing all the devices in the system is returned when an IODN of zero is specified. Since the number of IODNs in the system can exceed the length of the buffer, the query options field contains the starting IODN to be returned. Therefore, the requestor can obtain all of the defined IODNs by using several **Query Device SVCs**. The returned IODNs are in ascending order. The format of the returned information is shown in Figure 4-8 on page 4-60.

Offset			
8	Number of IODNs defined	Number of IODNs returned	
12	IODN	IOCN	This section is repeated for each returned IODN.
16	Device Name		
20	- - -		

Figure 4-8. Query Device Structure (IODN = 0)

Return Codes: contained in GPR2

0 =	Successful completion.
12 =	QDS not on full word boundary or invalid address.
16 =	Invalid IODN specified.
20 =	QDS not long enough.

Ring Queue Get Word SVC

Description: The **Ring Queue Get Word SVC** is used to obtain a fullword of data from a ring queue created in the VRM with **_rqc**. (See "Ring Queue Create (_rqc)" on page 5-92). The pointer to the queue must have been passed previously to the virtual machine by the cooperating VRM process that created the queue.

If there is no data in the queue, a -1 is returned in GPR2.

SVC Code: 0x**FFB9**

Calling Register Conventions:

GPR2 = Pointer to the VRM ring queue.

Return Codes: contained in GPR2

-1 = Ring queue empty.
-2 = Invalid ring queue pointer.
¬ -1 or -2 = Data from the queue.

Ring Queue Put Word SVC

Description: The **Ring Queue Put Word SVC** places a word of data in a VRM ring queue created with **_rqc**. (See "Ring Queue Create (_rqc)" on page 5-92). The pointer to the queue must have been passed previously to the virtual machine by the cooperating VRM process that created the queue.

If the queue is full, a -1 is returned in GPR2.

SVC Code: 0x**FFB8**

Calling Register Conventions:

GPR2 = Pointer to the VRM ring queue.

GPR3 = Fullword of data to be placed in the queue.

Return Codes: contained in GPR2

0 =	Successful completion.
-1 =	Ring queue full.
-2 =	Invalid ring queue pointer.

Send Command SVC

Description: The **Send Command SVC** initiates operations to a device, a device manager process or a virtual device. The command control information associated with this SVC controls the interaction with the device. This control information differs from the Start I/O CCB in both form and functions supported. It is designed for efficient transfer of small commands while also allowing larger commands to be transferred. It does not provide the scatter/gather data transfer function.

The status of the operation may be obtained in one of two ways. If interrupt on completion has been specified in the operation options, the status word, along with the status flags in the PSB, contains device-dependent information concerning the operation just completed. If no interrupts have been specified, the operating system can use the **Query Device SVC** to obtain the current status of the specified IODN.

The **Send Command SVC** is considered to be a synchronous operation, in that the request either has been queued for execution by the VRM or has been rejected before control is returned to that level of the operating system. If the request has been rejected, no interrupts are generated. The operating system can specify whether the transfer of data is to be synchronous.

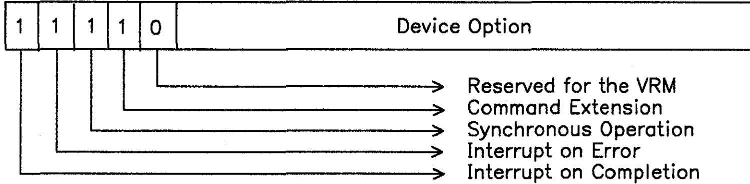
SVC Code: 0x**FFBF**

Calling Register Conventions:

GPR2 = IODN

This register contains the input/output device number for the device in bits 0 through 15. This operating system must have previously attached to this device with the **Attach Device SVC**.

Options — Contains the Operation and Device Options in bits 16 through 31. The Operation Options allow the operating system to define the relationship of the execution of an I/O operation to that operating system's execution. This field is shown in Figure 4-9 on page 4-64.



The operations options fields are defined as follows:

- Command Extension — This option specifies that a command extension exists and that its address is in GPR5 and length in GPR6.
- Synchronous Operation — This option specifies that the current operating system is placed in a wait state until the command is completed or an error is encountered.
- Interrupt on Error — This option specifies that the operating system is interrupted (on the interrupt level specified on the **Attach Device SVC**) if this command terminates with an error condition. If a command is terminated by a **Cancel I/O SVC**, the system considers the cancellation to be an error condition.
- Interrupt on Completion — This option specifies that the operating system is to be interrupted (on the interrupt level specified on the **Attach Device SVC)** when this command has been completed.

The Device Option field is 11-bits and contains the option that the device manager or driver uses. Few, if any, devices need all 11 bits to define I/O options. In this case, the bits not actually used to specify the device option may be used as flags (bit-encoded modifiers) to the device option. IBM-supplied device managers specify the device option in the low-order bits of the device option field.

The following device options are defined for all device managers:

0 = Configure real devices
1 = Create virtual/logical device
2 = Open virtual/logical device
3 = Close virtual/logical device
4 = Delete virtual/logical device.

Values between 5 and 2,047 are device-dependent options.

GPR3 = Contains command-specific data.

GPR4 = Contains command-specific data.

GPR5 = Contains the address of the command extension when the long form of the **Send Command SVC** is specified or command-specific data when the short form is specified.

GPR6 = Contains the length of the command extension when the long form of the **Send Command SVC** is specified or command-specific data when the short form is specified.

GPR7 = Path identifier

This register contains the identifier of the path connecting the virtual machine and the device.

Return Codes: contained in GPR2

-32768 = Operation cancelled.
-1 = Unable to pin pages due to insufficient real memory or the page is already pinned 255 times.
0 = Successful completion.
8 = Unable to pin pages due to invalid command extension segment identifier.
12 = Invalid command extension length or alignment error (command extension must be word-aligned).
16 = Invalid path identifier specified.
22 = VRM reserved bit set in operation options.
≥256 = Device-dependent return codes, operation rejected.
<0 = Device-dependent return codes, error during operation.

Comments: When a device driver is processing this request, the VRM pins the command extension.

I/O buffers used by this service can reside in bus I/O memory (addresses 0xF4000000 through 0xF4FFFFFF).

For synchronous operations, return codes >0 indicate that the operation was not initiated because an error was detected by the SVC handler (rc<256) or the device's check parameters routine (rc≥256). Return codes <0 indicate that the operation was initiated but terminated due to an error condition. Two of these return codes are common for all devices. A -1 indicates that the operation failed due to insufficient resources and a -32,768 means the operation was cancelled.

Asynchronous operations return a zero when the operation begins, but this does not necessarily mean the operation completed without error. The virtual machine must check the operation results field of the Send Command program status block to determine if the request was successful. A zero in the operation results field indicates successful completion, and a negative value means the operation was started but did not finish due to some error. If an asynchronous operation returns a positive value in GPR2, the operation never started and no PSB is generated. Asynchronous operations never return a negative value in GPR2.

Information describing the last operation completed by a device is kept in two places:

1. The program status block (PSB) for the interrupt level specified by the virtual machine when it is attached to the device
2. The device's query device structure (QDS).

   The QDS is updated unconditionally at the completion of an operation while the PSB is updated only if an interrupt is generated. The QDS operation completion information associated with a **Send Command SVC** is:

   Status Flags — 0x00010s00, where 's' indicates if the interrupt was solicited or unsolicited.

   Overrun Count — Set to 0 for solicited interrupts and set to overrun count for unsolicited interrupts.

   Status Word — Operation Result — Contains the return code for the operation in bits 0 through 15. An error is indicated if the value is negative (high-order bit is on).

   IODN — Contains the IODN of the device in bits 16 through 31.

   Data Word 1 — Operation options for the command in bits 0 through 15.

   Device-dependent data or the command extension segment ID in bits 16 through 31.

   Data Word 2 — Device-dependent data or the command extension address.

   Data Word 3 — Device-dependent data.

Figure 4-9 shows the format of the Send Command program status block.

Offset				
0	Old IAR			
4	Old ICS			
8	Reserved	Old CS	Reserved	Sublevels
12	New IAR			
16	Reserved	New ICS	Status Flags	Overrun Count
20	Operation Result		IODN	
24	Options		Device Dependent/CE Seg ID	
28	Device Dependent / Command Extension Address			
32	Device Dependent			
36				

Figure 4-9. Send Command Program Status Block

Start I/O SVC

Description: The **Start I/O SVC** initiates input/output operations to a device driver. The command control block (CCB) associated with the **Start I/O SVC** contains information used to control the execution and flow of the I/O operation.

Each I/O operation is an atomic operation with respect to the specified device. That is, all command elements in a CCB are executed before another Start I/O work request for that device is processed.

The status of the I/O operation is obtained in two ways. If interrupt on completion has been specified in the CCB, the status word, along with the status flags in the PSB, contains device-dependent information concerning the operation just completed. If no interrupts have been specified, the operating system can use the **Query Device SVC** to obtain the current status of the specified IODN.

The **Start I/O SVC** is considered to be a synchronous operation, in that the request either has been queued for execution by the the VRM or has been rejected before control is returned to that level of the operating system. If the request has been rejected, no interrupts are generated. The operating system has the option of transferring the data synchronously or asynchronously.

SVC Code: 0xFFF4

Calling Register Conventions:

GPR2 = Command control block

This register contains the virtual address of a command control block (CCB)

GPR3 = Path identifier

This register contains the identifier of the path connecting the virtual machine to the device.

The CCB contains the information to start an I/O transfer or command. The CCB is composed of two parts. The first is the command header and the second (which is optional) is the command element. I/O commands have no command elements. I/O transfers must have one command element for each transfer area. Thus, the command elements provide a chaining capability as shown in Figure 4-10 on page 4-68. The command header and all the command elements that define an I/O operation must be contiguous in storage.

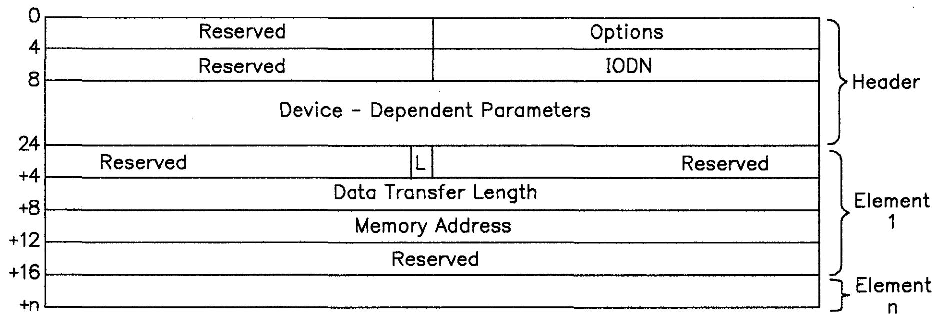
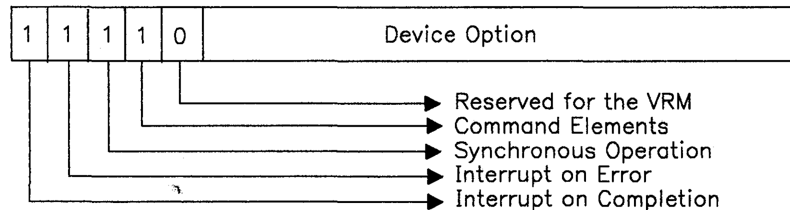


Figure 4-10. Command Control Block (CCB)

The command header contains information about the I/O operation in general. The command header must be aligned on a fullword boundary.

The fields of the CCB header are defined as follows:

- Operation options — This field allows the operating system to define the relationship of the execution of an I/O operation to that operating system's execution. This field is structured as follows:



The operation options fields are defined as follows:

– Command Element — This option specifies that the CCB contains one or more command elements.

– Synchronous Operation — This option specifies that the current operating system is to be placed in a wait state until the I/O operation is completed or an error is encountered.

– Interrupt on Error — This option specifies that the operating system is to be interrupted (on the interrupt level specified on the **Attach Device SVC**) if this I/O operation terminates with an error condition. When an I/O operation is terminated by a **Cancel I/O SVC**, the system considers the cancellation to be an error condition.

- Interrupt on Completion — This option specifies that the operating system is to be interrupted (on the interrupt level specified on the **Attach Device SVC**) when this I/O operation has been completed.
- Device Option — This 11-bit field contains the option that the device driver uses. Few, if any, devices need all 11 bits to define I/O options. In this case, the bits not actually used to specify the device option may be used as flags (bit-encoded modifiers) to the device option. IBM-supplied device drivers specify the device option in the low-order bits of the device option field.

  The following device options are defined for all device drivers:

  0 = Read
  1 = Write
  2 = Position
  3 = Format
  4 = Change characteristics.

  Values between 5 and 2,047 are device-dependent options.

- IODN — This field contains the 16-bit identifier for the specified I/O device. The operating system must have previously attached this device with the **Attach Device SVC.**
- Device Dependent Parameters — This field contains device-dependent parameters such as a disk block number. Even if the device does not require the full 16 bytes reserved for this field, the first command element, if required, must start at byte offset 24 from the start of the command header.

Each command element is used to define an area of memory for a data transfer request.

The command element fields are defined as follows:

- Link (L) — Bit 15, the link bit, has meaning only for CCBs that require one or more command elements. When bit 15 is equal to zero, this indicates the end of the command element chain.
- Data Transfer Length — This field contains the length of the data to be transferred in bytes.
- Memory Address — This field contains the address of the area to be used in this I/O transfer.

Return Codes: contained in GPR2

-32768 = Operation cancelled.
-1 = Unable to pin pages due to insufficient real memory or the page is already pinned 255 times.
0 = Successful completion.
8 = Unable to pin pages due to invalid CCB or buffer segment identifier.

12	= Unable to pin pages due to invalid CCB or buffer length, or alignment error (either CCB which must be word-aligned or an I/O buffer which may have alignment dependencies).
13	= Length specification error
14	= Invalid device/media characteristics data
16	= Invalid path identifier specified
22	= Invalid option
≥256	= Device-dependent return codes, operation rejected
<0	= Device-dependent return codes, error during operation.

Comments: When a device driver is processing the request, the VRM pins the CCB and buffers.

I/O buffers used by this service can reside in bus I/O memory (addresses 0xF4000000 through 0xF4FFFFFF).

For synchronous operations, return codes >0 indicate that the operation was not initiated because an error was detected by the SVC handler (rc<256) or the device's check parameters routine (rc≥256). Return codes <0 indicate that the operation was initiated but terminated due to an error condition. Two of these return codes are common for all devices. A -1 indicates that the operation failed due to insufficient resources, and a -32,768 indicates that the operation was cancelled. The return code is available in GPR2 after the SVC instruction completes.

Asynchronous operations return a zero when the operation begins, but this does not necessarily mean the operation completed without error. When the request completes, the virtual machine checks the operation results field of the Send Command program status block to determine if the request was successful. A zero in the operation results field indicates successful completion, and a negative value means the operation was started but did not finish due to some error. If an asynchronous operation returns a positive value in GPR2, the operation never started and no PSB is generated. Asynchronous operations never return a negative value in GPR2.

Information describing the last operation completed by a device is kept in two places, the program status block (PSB) for the interrupt level specified when the virtual machine attached to the device, and the device's query device structure (QDS). The QDS is updated unconditionally at the completion of an operation, while the PSB is updated only if an interrupt is generated. The QDS operation completion information from a **Start I/O SVC** is as follows:

Status Flags — 0x00100s00, where the 's' bit indicates if the interrupt was solicited or unsolicited.

Overrun Count — Set to 0 for solicited interrupts and set to overrun count for unsolicited interrupts.

Status Word — Operation Result — Contains the return code for the operation in bits 0 through 15. An error is indicated if the high-order bit is on.

IODN — Contains the IODN of the device in bits 16 through 31.

Data Word 1 — Contains device-dependent data in bits 0 through 15. Bits 16 through 31 of this word contain the segment ID of the CCB that was most recently completed.

Data Word 2 — CCB Address — Contains the address of the CCB most recently completed.

Data Word 3 — Contains device-dependent data.

The format of the Start I/O program status block is shown in Figure 4-11.

Offset				
0	Old IAR			
4	Old ICS			
8	Reserved	Old CS	Reserved	Sublevels
12	New IAR			
16	Reserved	New ICS	Status Flags	Overrun Count
20	Operation Result		IODN	
24	Device-dependent data		CCB Segment ID	
28	CCB Address			
32	Device-dependent data			
36				

Figure 4-11. Start I/O Program Status Block

Virtual Machine Communications SVCs

The VRM recognizes the following virtual machine communications SVCs from the virtual machine:

- Send Address Message
- Send Immediate Message
- Set Message Receive.

Send Address Message SVC

Description: The **Send Address Message SVC** sends an address message to the specified virtual machine. This is also the recommended way to share segments among virtual machines.

SVC Code: 0x**FFDE**

Calling Register Conventions:

GPR2 = Emergency

Bit 0 is set to 1 if this is an emergency message.

Virtual machine ID — Contains an integer identifying the virtual machine in bits 1 through 15.

SegId — Contains in bits 17-31 the ID of the segment in which the message can be found.

GPR3 = Displacement

This register contains a 28-bit displacement within the segment.

GPR4 = Length

This register contains the 28-bit length of the message in bytes.

GPR5 = Type

This register contains a 32-bit type code. The meaning of this field must be agreed upon by sender and recipient.

Return Codes: contained in GPR2

0 = Successful.
4 = Specified virtual machine does not have message receive enabled.
8 = Invalid segment or invalid displacement into the segment.
12 = Virtual machine ID does not exist.

Send Immediate Message SVC

Description: The **Send Immediate Message SVC** sends an immediate message to another virtual machine.

SVC Code: 0x**FFDF**

Calling Register Conventions:

GPR2 = Emergency

Bit 0 is set to 1 if this is an emergency message.

Virtual machine ID — Contains an integer identifying the virtual machine in bits 1 through 15.

Length — Contains the length of the message in bits 25 through 31.

GPRs 3-5 = Messages, as needed.

Return Codes: contained in GPR2

0 = Successful.
4 = Specified virtual machine does not have message receive enabled.
12 = Virtual machine ID does not exist.

Set Message Receive SVC

Description: The **Set Message Receive SVC** sets interrupt levels and sublevels for receiving messages.

SVC Code: 0x**FFDD**

Calling Register Conventions:

GPR2 = Normal Level. This register contains an integer between 0 and 6 representing the normal level. This value is found in bits 0 through 7.

Normal Sublevel — Contains an integer between 0 and 255 in bits 8 through 15.

Emergency Level — Contains an integer between 0 and 6 in bits 16 through 23.

Emergency Sublevel — Contains an integer between 0 and 255 in bits 24 through 31.

Return Codes: contained in GPR2

0 = Successful.
12 = One or more parameters out of range.

Receiving a Message from a Program Status Block

When a message is sent to a virtual machine, that machine receives an interrupt on the level and sublevel set with the **Set Message Receive SVC** (assuming that SVC has been issued and assuming the interrupt level is enabled). Figure 2-4 on page 2-17 shows the program status block (PSB) associated with the interrupt.

The purpose of this section is to describe the content of the status and data words of the PSB for these interrupts.

The three data words, at offsets 0x18, 0x1C, and 0x20 in the PSB have exactly the same content as GPR3, GPR4, and GPR5 as supplied with either the **Send Immediate Message SVC** or the **Send Address Message SVC**. The status word (at offset 14 hexadecimal) has content very similar to the content of GPR2 supplied with either the **Send Immediate Message SVC** or the **Send Address Message SVC**. The recipient finds the sender's virtual machine ID in bits 1-15 of the status word, rather than the recipient's ID. The rest of the status word has the same content as the value in GPR2 supplied by the sender.

Machine Control SVCs

The VRM recognizes the following machine control SVCs from the virtual machine:

- Debug a Virtual Machine
- IPL Virtual Machine
- Machine Identification
- Query Virtual Machine
- Re-IPL VRM
- Terminate Virtual Machine
- Update VRM.

Debug a Virtual Machine SVC

Description: A virtual machine can issue the **Debug SVC** to give the debugger control when it or any other virtual machine is next dispatched.

SVC Code: 0x**FFB6**

Calling Register Conventions:

GPR2 = The VMID of the virtual machine to be debugged.

Return Codes: contained in GPR2

0 = Successful completion.
16 = The virtual machine does not exist.

Comments: The debugger gets control when the virtual machine to be debugged is next dispatched. Therefore, a virtual machine that is waiting as the result of issuing a **VM Wait** command does not cause debugger invocation until it gets a virtual interrupt.

To invoke the debugger when a virtual machine is first dispatched, that virtual machine should execute this SVC. Only IPLed virtual machines can use this SVC.

If the debugger is not already loaded, it is loaded when the SVC is issued. The virtual machine issuing the SVC waits until the load is complete.

IPL Virtual Machine SVC

Description: The **IPL Virtual Machine SVC** allows one virtual machine to IPL another. The virtual machine is IPLed from the specified IODN and given the requested VMID. This command is part of the machine control procedures (MCP) that can be used without a formal operating system. When the VRM is IPLed, the MCP typically issues this command to IPL at least one virtual machine.

SVC Code: 0x**FFC2**

Calling Register conventions:

GPR2 = IPL type and origin

These fields are defined as follows:

Bit	**Meaning**
0	1= Interrupt on error
1	1= Interrupt on completion
2	1= Synchronous IPL
16-31	IPL device — This specifies the IODN of the minidisk or diskette from which the IPL is done. The diskette or minidisk must contain a virtual machine IPL record at either block number 0 (if IPLing from a minidisk) or sector 0 (if from diskette). See Figure 4-12 on page 4-80.

GPR3 = Virtual machine ID

Register 3 must specify the VMID for the new virtual machine. Valid VMIDs are in the range 0-255. A VMID of zero requests the next available VMID.

GPR4 = This register contains the interrupt level in bits 21 through 23 and sublevel in 24 through 31 if GPR2 bits 0 or 1 are one.

Return Codes: contained in GPR2

- -1 = Insufficient resources.
- 0 = Virtual machine IPL successful. This does not include errors found by the virtual machine during its internal IPL process.
- 4 = Invalid parameter.
- 8 = Mount diskette.
- 10 = Device/Minidisk does not exist.
- 12 = IPL header is invalid. This refers to the contents of the virtual machine IPL record. See Figure 4-12 on page 4-80 for a description of the virtual machine IPL record.
- 16 = Error reading virtual machine IPL record.
- 24 = No more virtual machines can be supported.

28 = Duplicate VMID.
32 = An IPL was issued after the 're-IPL virtual machine' key sequence and before the virtual machine was automatically IPLed.

Comments: The **IPL Virtual Machine SVC** is privileged to the operating system state.

Upon successful return, the virtual machine will have been created and established as a VRM process. In the following discussion, the virtual machine's segment 0 is referred to as the IPL segment. When it is entered, the following environment exists:

- Page 0 in the IPL segment (segment register 0 contents) is pinned.
- The execution level is 7.
- The time of IPL is set.
- Timer is disabled.
- No interrupts (levels 0 through 6 and machine communications interrupts) are individually masked off.
- All level 0-6 interrupts and machine communications interrupts as a group are masked off (ICS bit 31 on).
- Page fault notification is disabled.
- The virtual machine comes up in operating system state.
- The VMID is the one specified when the **IPL Virtual Machine SVC** was issued.
- The virtual machine's segment 0 is created and loaded so that problem state tasks have no access to it. The virtual machine must issue a **Protect Page Range SVC** if such tasks are to access the segment.
- Upon entry, the following registers are set:

  GPR2 = The IODN of the device from which the virtual machine was IPLed.
  GPR3 = The ID of the virtual machine issuing the SVC, or 0 if IPLed from the VRM.
  GPR4 = The segment ID of the virtual machine's segment 0.
  GPR5 = The length of the virtual machine's segment 0.

If bits 0, 1, and 2 of GPR2 are zero when this SVC is issued, the IPL is done asynchronously, but no interrupt is returned.

Although the IPL header has an eight-character VM name field, only the first four characters are used. See Figure 4-13 on page 4-81 for the PSB format.

The IPL record is located on block 0 of the volume containing the bootstrap for the virtual machine being IPLed. It points to an area on the volume which, in most cases, contains the bootstrap loader for an operating system. The IPL record can be located on any 128-byte boundary within the first 512 bytes of block 0. Valid locations for the IPL record, therefore, are at offsets 0, 128, 256, or 384. The first valid boot record found at any of these locations is the one that is IPLed. The bootstrap itself must be contained in contiguous sectors.

Figure 4-12 on page 4-80 illustrates the IPL record format. Refer to Figure 4-13 on page 4-81 for a description of the PSB at IPL.

The program load address is the address, relative to logical segment 0, at which the bootstrap is loaded. You must be aware of the reserved page 0 locations. The entry point is the address, relative to the start of logical segment 0, at which the bootstrap gets control. The two lengths, at offsets 12 and 28, contain the length, in bytes, of the bootstrap and segment 0, respectively.

The six bytes at offset 20 contain format information. The last track field refers to the last track number on a cylinder (the number of tracks/cylinder - 1). The last sector gives the number of sectors on a track.

Offset	Field		
0	Record type		
4	Program entry point		
8	Program load address		
12	Program length		
16	Starting block number		
20	Bytes/Sector *	Last Track *	Last Sector *
24	Sectors/Block *	Reserved	
28	Segment 0- Initial Length		
32 . . 96	Reserved		
96	User-defined		
128			

Figure 4-12. IPL Record Format. Fields marked with an asterisk (*) are required only if the IPL record is on a diskette. Note that this 128-byte area can be located at any 128-byte offset (0, 128, 256, or 384) within the first 512 bytes of block 0.

Offset				
0	Old IAR			
+4	Old ICS			
+8	Reserved	Old CS	Reserved	Number of sublevels
+12	New IAR / Origin of sublevel vector			
+16	Reserved	New ICS	Status flags	Overrun count
+20	Return Code		IPL's IODN	
+24	IODN specified for VM		VMID	
+28	Reserved			
+32	Reserved			
+36				

Figure 4-13. Virtual Machine Program Status Block from IPL

Machine Identification SVC

Description: The **Machine Identification SVC** allows an application to determine the unique serial number and machine type associated with each RT PC.

The **Machine Identification SVC** can be issued in problem state.

SVC Code: 0x**FFD4**

Calling Register Conventions:

GPR2 = Contains a pointer to a word-aligned area in memory

GPR3 = Contains the length of the area pointed to by GPR2

Currently, the area is defined as 2 words (8 bytes). The first word is filled in with the unique machine identification number. The second word is filled in with a value to indicate the model type of the machine. A 0 in the second word indicates a 6150 and a -1 indicates a 6151.

Return Codes: contained in GPR2

0 = Successful completion.
4 = Unique ID may have been tampered with.
16 = Specified area is not the correct length or is not word-aligned.

Query Virtual Machine SVC

Description: The **Query Virtual Machine SVC** allows an operating system to identify:

- If a specific virtual machine is IPLed
- All IPLed virtual machines in the system.

The information is returned in a structure as shown in Figure 4-14 on page 4-84. The address of the structure is passed by way of the SVC.

SVC Code: 0x**FFD8**

Calling Register Conventions:

GPR2 = Query type

Query type is defined as follows:

Bit 0=1 Query specific VMID
Bit 1=1 Query specific VM name
Bit 2=1 Query all VMs
Bits 3-31 Reserved.

GPR3 = Address of the structure

GPR4 = VM name or ID

GPR2 bit 0=1 VMID is contained in GPR4 bits 24-31
GPR2 bit 1=1 Virtual machine name is contained in GPR4.

Return Codes: contained in GPR2

0 = Successful completion.
4 = No virtual machines satisfy this request.
16 = Invalid query option (GPR2 bits 0-7), or the specified VMID is invalid.
24 = The supplied structure is not long enough to satisfy the request.

Comments: The **Query Virtual Machine SVC** is privileged to the operating system state.

Before issuing this SVC, the number-of-elements field in the structure must be set to reflect the possible number of entries.

If the supplied structure is not long enough to contain all the virtual machines (return code = 24), the number-of-entries area in the structure tells how many virtual machines really exist. The SVC should be re-issued with the structure size increased to allow for that many virtual machines. See Figure 4-14 on page 4-84 for a description of the structure passed by the SVC.

A virtual machine might issue this SVC prior to using virtual machine communications to determine the identifier of a virtual machine for which the virtual machine name is known.

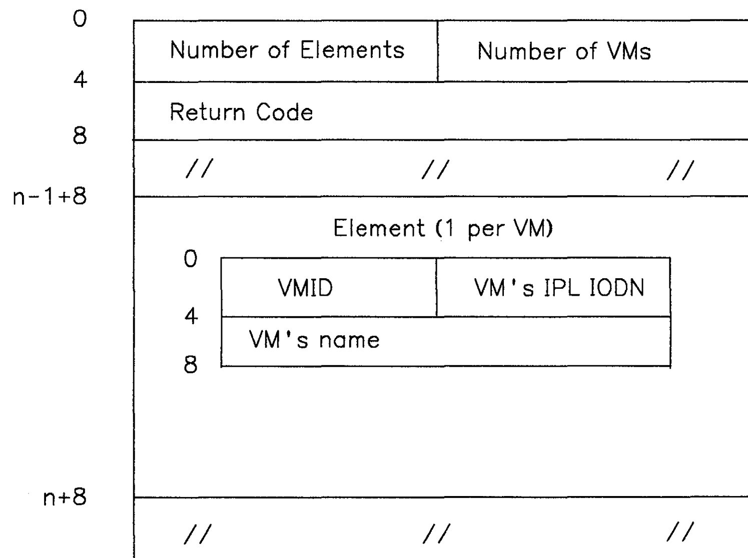


Figure 4-14. Structure Returned from "Query VM SVC"

Re-IPL VRM SVC

Description: The **Re-IPL VRM SVC** causes another IPL of the workstation, typically after you install code into or update the VRM.

SVC Code: 0x**FFD7**

Calling Register Conventions: none.

Return Codes: contained in GPR2

- 0 = Successful completion.
- 16 = Another virtual machine is active.

Comments: The **Re-IPL VRM SVC** is privileged to the operating system state.

Only the virtual machine that issued the **Re-IPL VRM SVC** may be active when this service is invoked.

This service returns to the caller only when an error condition prevents the VRM from re-IPLing the work station.

Terminate Virtual Machine SVC

Description: The **Terminate Virtual Machine SVC** allows a virtual machine to remove itself from the pool of dispatchable virtual machines. The machine control program can also use this command to terminate a virtual machine.

SVC Code: 0x**FFDC**

Calling Register Conventions:

GPR2 = Reason for termination

This parameter is set up as follows:

Bit 0	Reserved
Bit 1	Re-IPL this machine after termination
Bits 2-31	Reserved.

Return Codes: none.

Comments: The **Terminate Virtual Machine SVC** is privileged to the operating system state.

This instruction can be issued by a virtual machine which has determined to terminate itself for whatever reason. It is also the proper way to terminate after receiving notification of impending shutdown.

Update VRM SVC

Description: The VRM opens the VRM minidisk with read-only access (access rights = 0). The VRM then pages, on demand, the VRM code from this minidisk.

An update or install utility can stop the VRM from paging from the VRM minidisk by issuing the **Update VRM SVC**.

SVC Code: 0x**FFBD**

Calling Register Conventions: none.

Return Codes: contained in GPR2

0 = Successful completion.
16 = Another virtual machine is active.

Comments: The **Update VRM SVC** is privileged to the operating system state.

Only the virtual machine that issued the **Update VRM SVC** may be active when this service is invoked. Access rights for the VRM minidisk are changed to exclusive read/write if the virtual machine has the VRM minidisk open.

The VRM may page data or code directly from the VRM minidisk. This SVC causes the VRM to terminate paging from the VRM minidisk by copying all of the pages to the page space minidisk.

You must re-IPL the VRM after using this service to ensure updates take effect. See "Re-IPL VRM SVC" on page 4-85.

NVRAM Control SVCs

The VRM recognizes the following non-volatile random access memory (NVRAM) control SVCs from the virtual machine:

- Read NVRAM
- Write Data to NVRAM.

NVRAM starts at 0xF0008800 and is defined as shown in Figure 4-15.

Location	Data
F0008800 - F000880D	Real time clock data
F000880E - F000881D	Error log - 16 bytes
F000881E - F0008827	Usage log
F0008828 - F000882C	IPL source table
F000882D - F0008836	Reserved
F0008837	Dialog status
F0008838	Shutdown status
F0008839 - F000883A	Time data
F000883B - F000883C	IPL device IODN
F000883D	ACIS Operating System use
F000883E - F000883F	NVRAM CRC check bytes.

Figure 4-15. Layout of NVRAM

Read NVRAM SVC

Description: You can read up to 50 bytes starting at a specified offset in NVRAM with the **Read from NVRAM SVC**.

SVC Code: 0x**FFB5**

Calling Register Conventions:

GPR2 = Offset to first byte to read

This register contains the offset in bytes from the beginning of NVRAM to the first byte to be read. The first byte of NVRAM has offset zero.

GPR3 = Data area address

This register contains the address of the area in which the data will be placed.

GPR4 = Number of bytes to read

This register contains the number of bytes to read.

Return Codes: contained in GPR2

0 = Successful completion.

16 = An attempt was made to access outside of NVRAM.

Comments: The **Read Data from NVRAM SVC** is privileged to the operating system state.

Write Data to NVRAM SVC

Description: The **Write Data to NVRAM SVC** allows you to write data to NVRAM. The last two bytes of NVRAM are reserved. This means that, even though all 50 bytes can be read, you can only write to the first 48 bytes.

SVC Code: 0x**FFB4**

Calling Register Conventions:

GPR2 = Byte offset to place data

This register contains the byte offset into NVRAM of where to place the data. The first byte of NVRAM has offset zero.

GPR3 = Data's virtual address

This register contains the virtual address of the data to be written.

GPR4 = Number of bytes to write

This register contains the number of bytes to write.

Return Codes: contained in GPR2

0 = Successful completion.

16 = An attempt was made to write outside of NVRAM.

Comments: The **Write Data to NVRAM SVC** is privileged to the operating system state.

Chapter 5. Virtual Resource Manager Programming Interfaces

CONTENTS

About This Chapter 5-5
Process Management 5-6
  Change Attributes (_change) 5-7
  Create Process (_creatp) 5-10
  Initialize Process (_initp) 5-11
  Query ID (_queryi) 5-12
  Schedule Off-Level I/O Processing (_sio) 5-13
  Signal (_signal) 5-14
Queue Management 5-16
  Attach Queue (_attchq) 5-17
  Broadcast Virtual Interrupts (_bvint) 5-21
  Cancel Enqueue (_canclq) 5-22
  Create Queue (_creatq) 5-23
  Dequeue Element (_deque) 5-25
  Destroy Queue (_dstryq) 5-30
  Detach Queue (_detchq) 5-31
  Enqueue Element (_enque, _enq) 5-32
  Peek at Queue Element (_peekq) 5-36
  Post ECB (_post) 5-37
  Query Attached Path (_queryp) 5-38
  Read Queue Element (_readq) 5-39
  Wait for Event or Queue (_wait, _waitq) 5-40
Memory Management 5-42
  Attach Segment (_attchs) 5-43
  Convert Effective Address to Real Address (_cveara) 5-44
  Convert Virtual Address to Real Address (_qra) 5-45
  Detach Segment (_detchs) 5-46
  Load Byte (_loadb) 5-47
  Load Halfword (_loadh) 5-48
  Load Segment Register (_lsr) 5-49
  Load Word (_loadw) 5-50
  Move Buffer (_mvbuff) 5-51
  Pin Page Range (_pinpgs) 5-52
  Query Segment ID (_qsid) 5-53
  Restore Segment Register (_rsr) 5-54
  Save Segment Register (_ssr) 5-55
  Store Byte (_storeb) 5-56
  Store Halfword (_storeh) 5-57
  Store Word (_storew) 5-58
  Unpin Command Control Block (_upnccb) 5-59
  Unpin I/O Buffer (_upinio) 5-60
  Unpin Page Range (_unpin) 5-61
Semaphore Management 5-62
  Create Semaphore (_creats) 5-63

Destroy Semaphore (_dstrys) 5-64
Receive Semaphore (_recv) 5-65
Send Semaphore (_send) 5-66
Timer Management 5-67
Cancel Interval Timer (_cnltmr) 5-68
Control Device Timer (_ctldvt) 5-69
Set Device Timer (_setdvt) 5-70
Set Interval Timer (_settmr) 5-71
Wait for Interval Timer (_sleep) 5-73
Program Management 5-74
Allocate memory (_mallc) 5-75
Bind Module (_bind) 5-76
Copy Module (_copy) 5-78
Free Allocated Memory (_mfree) 5-80
Query Module ID (_querym) 5-81
Virtual Machine Control Procedures 5-82
Query Virtual Machine (_queryv) 5-83
Terminate Virtual Machine 5-85
Input/Output Procedures 5-86
Allocate a Buffer from the Buffer Pool (_bfget) 5-87
Free Allocated Buffer (_bffree) 5-88
Read Block (_rdblk) 5-89
Read Words (_rdwds) 5-90
Receive Data from Bus Memory (_erecv) 5-91
Ring Queue Create (_rqc) 5-92
Ring Queue Delete (_rqd) 5-94
Ring Queue Get Word (_rqgetw) 5-95
Ring Queue Put Word (_rqputw) 5-96
Start Direct Memory Access Transfer (_stdma) 5-97
Write Block (_wrblk) 5-100
Write Words (_wrwds) 5-101
Minidisk Management 5-102
Check for Bad Blocks (_chkblk) 5-103
Minidisk Bad Blocks (_badblk) 5-104
Minidisk Manager Check (_mdmchk) 5-105
Device Management 5-106
Allocate Device-Dependent Data (_dalct) 5-107
Assign a Device to the Coprocessor (_assign) 5-108
Change Bus Mask (_chgmsk) 5-109
Define Device (_defind) 5-110
Machine Identification (_uname) 5-112
Map System Memory (_mapsys) 5-113
Query Device Identifier (_queryd) 5-115
Query Device Status (_qryds) 5-116
Set up region mode (_dmamov) 5-117
VRM Trace and Error Process Interfaces 5-119

Error Process (errrecvr on/off) 5-119
Error Process Input Queue Element 5-119
Error Process Acknowledgement Queue Element 5-120
Error Entry Acknowledgement Queue Element 5-121
Trace Process 5-123
Trace Process Input Queue Element 5-123
Trace Process Acknowledgement Queue Element 5-124
Trace Process Input Queue Element 5-125
Trace Process Acknowledgement Queue Element 5-125
Trace Buffer Acknowledgement Queue Element 5-126
Event Monitoring 5-128
Generate Error Entry (_errvrm) 5-129
Generate Generic Trace Entry (_trcgen) 5-133
Generate Trace Entry (_trcvrm) 5-135
Internal VRM Trace (_vrmtrc) 5-137
Save/Get Dump Table Entry (_dmptbl) 5-140

About This Chapter

This chapter describes the runtime routines that manage processes, queues, semaphores, timers, and system storage. Each runtime routine includes a description of the call, the subroutine format and parameter sequences, and possible return values. In addition, the conditions that terminate the caller for any particular subroutine are described.

The subroutines are presented from the machine-level point of view. Parameters are described as data or addresses passed in general purpose registers 2, 3, 4, and 5. Note that only four such parameters can be passed in the general purpose registers. Subroutines that require more than four parameters must pass these parameters on the stack. These ***stack parameters*** are pointed to by an offset from the value in GPR1. GPR1 always points to the stack. A subroutine that uses a stack parameter would be expressed as follows:

```
GPR2 = parameter 1 data
GPR3 = parameter 2 data
GPR4 = parameter 3 data
GPR5 = parameter 4 data
(0(R1)) = parameter 5 data
(4(R1)) = parameter 6 data
```

In the preceding example, `(0(R1))` represents the stack parameter. `(R1)` indicates register 1, which points to the stack. `0` is the offset into the stack. For this example, the stack parameter is the value found at offset 0 from the address pointed to by R1. You may also see `(4(R1))`, which indicates a second stack parameter found at offset 4 from the beginning of the stack.

For more information on the register conventions and parameter-passing mechanisms used on RT PC, see "Subroutine Linkage Conventions" on page B-8.

Parameters that are less than 32 bits in length must be right-justified in the register with the high-order, unused bits set equal to zero.

Note to C language programmers:

All the VRM calls defined in this chapter are presented in machine language conventions. Please see Appendix D, "C Language VRM Subroutines" on page D-1 for the C language definition of these calls.

Process Management

The VRM contains mechanisms to control the multiple processes active at any one time. The process management runtime routines are:

- Change attributes
- Create process
- Initialize process
- Query ID
- Schedule off-level I/O processing
- Signal.

On the following pages, each function is described. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Change Attributes (_change)

Description:

The change function allows modification of a process or device driver attribute. Attributes that may be changed include:

- Process priority
- Entry points:
  - Check parameters
  - I/O initiate
  - Interrupt handler
  - Exception handler
  - Off-level I/O handler.

For more information on VRM process and device driver entry points, see *Virtual Resource Manager Device Support.*

The change function is typically used during the initialization routine, although it can be used at any time.

The main entry point of a module is used as the default entry point for all required functions in a device manager process or a device driver. This function can be used to change the required entry points and/or specify the optional check parameter, exception handler, and off-level handler entry points.

The ID parameter identifies the object being altered. The valid IDs and their associated entry points are:

- PID or DID
  - Exception handler (type 5x entry point)
- SLIH ID or DID
  - Interrupt handler (type 4x entry point)
  - Exception handler (type 5x entry point)
- QID or DID
  - Check parameter (type 2x entry point)
  - I/O initiate (type 3x entry point)[1]
- DID
  - Off-level I/O handler (type 7x entry point) or any of the above.

---

1 If queue served by a SLIH

The caller of this function must be the owner of the object being changed. For example, if a PID is specified it must be the caller's PID. If a QID is specified, the server of the queue must be the caller.

The next parameter, process priority, is an optional parameter that applies only to processes. The priority value applies to VRM and virtual machine processes. The values for this parameter are defined as follows:

0 = High priority

8 = Default priority for VRM process

12 = Default priority for virtual machine process

15 = Low priority.

If specified for a device driver, the priority parameter is ignored.

Entry points can be specified in an array of up to five entries. Each entry specifies the type of entry point, the module ID, and the address. Valid types include:

2 = Check parameter
3 = I/O initiate
4 = Interrupt handler
5 = Exception handler
7 = Off-level I/O handler.

If the off-level I/O handler entry point is modified, the contents of segment register 13 are written over and not saved.

Subroutine Call:

```
Return Code = _change (ID, flags, priority, entry pts., num elements)
```

Calling Register Conventions:

GPR2 = ID of the object to change
GPR3 = Bit flags
0x02 = change priority
0x04 = entry points input
GPR4 = Process priority
GPR5 = Word-aligned address of the entry point array shown in Figure 5-1 on page 5-9.
(0(R1)) = Number of array elements

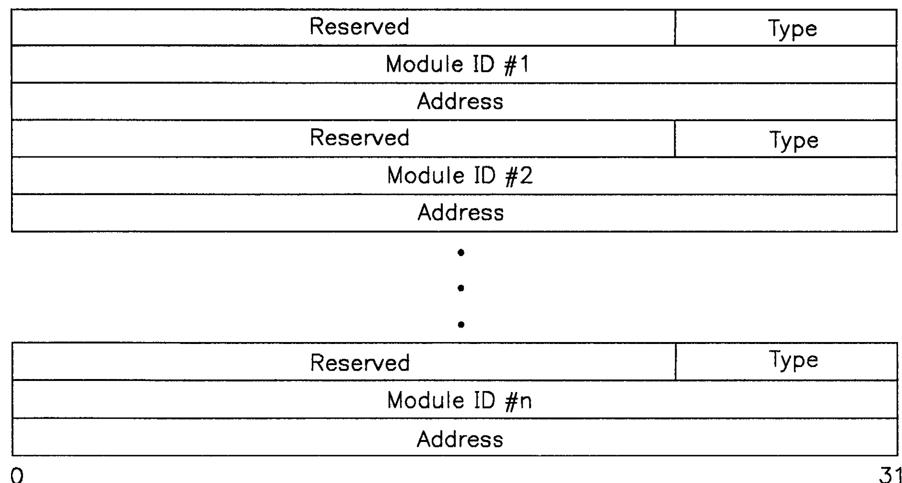


Figure 5-1. Entry Point Array Structure for _change

In the preceding figure, please note that 'address' has different meanings for IBM-supplied modules and modules you may have converted. For IBM-supplied modules, the address value is the new entry point address. For converted modules, the address points to the module's constant pool. The first entry in the constant pool is the address of the new entry point.

Return Codes: contained in GPR2

-1 = Insufficient resources.

This condition can occur only when a process is defining an exception handler entry point. In this case, the VRM must allocate a control block and there may not be enough space (resources) to do so.

0 = Successful.

Termination: The following conditions cause the system to abend:

- Caller specified an invalid ID or an ID not associated with caller
- Caller specified an invalid parameter (priority, flags, or type).

Create Process (_creatp)

Description: The create function changes a process from terminated to uninitialized. System control blocks are allocated for the process, and a segment is created to contain the process' stack.

You can assign an optional 4-byte logical name to the process when you create it. This name is not guaranteed to be unique and is used only as an aid when debugging (when finding control blocks in a dump, for example).

The process ID (PID) is returned if the creation is successful. The PID, as described in "Naming Conventions" on page 3-6, uniquely identifies the process and is required for assigning queues for the process to serve. The creation of these queues can be handled by the creator of the process or the new process itself.

Subroutine Call:

```
Return Code = _creatp (process name, PID);
```

Calling Register Conventions:

GPR2 = Optional unsigned 4-character process name
GPR3 = Address of the returned process ID. The process ID is a word value that is word-aligned in memory.

Return Codes: contained in GPR2

0 = Successful
-1 = Insufficient resources.

Termination: The system will abend if a device driver calls the create process function.

Initialize Process (_initp)

Description: Process initialization completes the transition of a process from terminated to ready. Prior to this point, the VRM has set up the resources the new process will use: process' code has been defined, the process was created, a queue was created for the process, entry points were bound, and so on.

The parameters used include the PID, to identify the process, and the MID, to identify the main entry point of the process.

After initialization, a process enters the ready state. When dispatched, the process begins executing instructions at its main entry point.

You can pass parameters to the process, such as the IDs of queues or semaphores. The structure of the parameters must be agreed upon by the process that calls this function and the process it initializes.

Note: The Initialize Process routine writes over, and does not save, the contents of segment register 12.

Subroutine Call:

```
Return Code = _initp (PID, MID, initialization parms, length of
                      parms, ret'd segment ID, ret'd segment offset);
```

Calling Register Conventions:

The values in registers 2 through 5 are input parameters, and the values in registers 6 and 7 are output parameters.

GPR2 = Process ID
GPR3 = Module ID
GPR4 = Word-aligned address of initialization parameters
GPR5 = Length of initialization parameters
(0(R1)) = The address of the parameters' segment ID. The segment ID is a halfword value that is halfword-aligned in memory. (If this parameter has a value of -1, no segment ID is returned.)
(4(R1)) = The address of parameters' segment offset. The segment offset is a word value that is word-aligned in memory. (If this value equals -1, no segment offset is returned.)

Return Codes: contained in GPR2

-1 = Insufficient resources.
0 = Successful.

Termination: The following conditions cause the system to abend:

- Initialize process function called by a device driver
- Caller specified an invalid process or module ID.

Query ID (_queryi)

Description: The query function allows a component to find out information about queues served by itself or another component.

The input parameter is an ID (either a PID, SLIH ID, or DID). The output is:

- A number from 0 to 24 indicating the number of queues served by the process or device driver
- An array containing an entry for each queue indicating:
  - The queue's ID (QID)
  - The ECB associated with the queue (none, if server is a device driver).

A component may need to ascertain its own ID in the following situations:

- When a process is executing its initialization routine
- When a common routine, shared among multiple processes or device drivers, is called.

A component can access the global variable **_curid** to determine its own ID.

Subroutine Call:

```
Return Code = _queryi (ID, # of queues, QID structure,
                           # of QID array elements);
```

Calling Register Conventions:

GPR2 = Server ID
GPR3 = Address of the returned number of queues. This is a word value aligned on a word boundary in memory.
GPR4 = Word-aligned address of the queue ID array

Each queue has an entry in the array. Each entry consists of two words, the queue's identifier (QID) and the event control bit (ECB) mask of the queue.
GPR5 = Number of elements in the QID array.

Return Codes: contained in GPR2

0 = Successful.

Termination: The following conditions cause the system to abend:

- Caller specified an invalid ID.

Schedule Off-Level I/O Processing (_sio)

Description: This function allows device drivers to schedule some of their work off of the interrupt level. Thus, if the second-level interrupt handler (SLIH) subroutine of a device driver must do some time-consuming processing and this work does not need to be performed immediately, the processing can be scheduled 'off level'.

This service allows SLIHs to run as fast as possible, avoiding interrupt-processing delays and overrun conditions.

You can call this routine to schedule work off-level, or you can give the address of the DDS as the return code from the SLIH subroutine.

Subroutine Call:

```
Return Code = _sio (DDS address);
```

Calling Register Conventions:

GPR2 = Word-aligned address of the device's DDS.

Return Codes: contained in GPR2

0 = Successful.

Signal (_signal)

Description: The concept of signalling an exception handler can be compared to causing a software interrupt. The primary function of a signal is to force another process to terminate, although it has other uses. For example, a 'wake-up' from timer services is sent using a signal (see "Timer Management" on page 3-29).

The target of a termination signal is always another process. To terminate itself, a process simply returns to its caller. If the target process has no defined exception handler, a VRM-supplied default routine executes instead. For termination signals, this routine attempts to release resources held by the process. For signals other than termination, the default routine does nothing.

A target ID and 32-bit signal mask are required input parameters. This mask tells you the kind of signal that was sent. When bit 0 equals 1, the signal is for termination. The next 7 bits are reserved, and you can define the remaining 24 bits for cooperating processes. Figure 5-2 illustrates the bit mask structure.

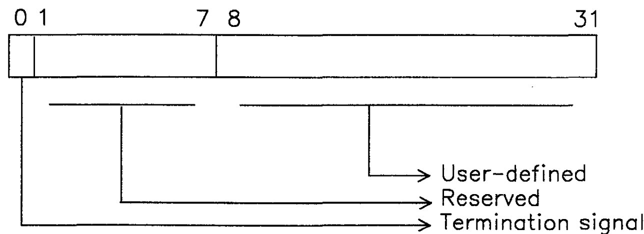


Figure 5-2. Signal Mask

If a process receives a signal but has not executed an exception handler routine and it receives another signal, the second signal mask is ORed into the first signal mask. Also, the process may receive another signal while the exception handler is being executed. In this case, after the exception handler returns, it is called again with the new signal mask.

Normally, the exception handler routine generates a return code of zero. If, however, a termination signal has been received and the exception handler does not want to terminate, it returns a -1. In this way, a process can reject a termination signal.

The signal function is also used to implement timer functions. The signal mask is used as a timer ID that uniquely identifies individual requests. To avoid confusion, processes should establish unique signal masks for timer requests.

Exception handler subroutines share some of the restrictions and limitations of hardware device drivers. For example, exception handlers cannot call any services that might put the caller in the process' wait state. In addition, any service that is

prohibited for use by an interrupt handler or device driver is also prohibited for an exception handler. The following services cannot be called by an exception handler:

- **_wait** or **_waitq**
- **_recv**
- **_sleep**

Subroutine Call:

```
Return Code = _signal (ID, mask);
```

Calling Register Conventions:

GPR2 = Target ID
GPR3 = Signal mask.

Return Codes: contained in GPR2

0 = Successful
4 = Invalid target ID.

Queue Management

The VRM has several mechanisms for controlling queues. The queue management runtime routines are:

- Attach queue
- Broadcast virtual interrupt
- Cancel enqueue
- Create queue
- Dequeue element
- Destroy queue
- Detach queue
- Enqueue element
- Peek at queue element
- Post event control bit
- Query attached path
- Read queue element
- Wait for event or queue.

Each function is described in the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Note: The 32-byte queue elements manipulated by queue management routines must be defined on fullword boundaries.

Attach Queue (_attchq)

Description: The attach function establishes a path between two communicating VRM components.

The three parameters associated with this function are:

- From ID
- To ID
- Acknowledgment information.

The From ID and the To ID are the IDs of the enqueuer and the server of the queue, respectively. The ID can be any one of the following:

- Process ID
- SLIH ID
- Queue ID
- Device ID.

Neither ID needs to be associated with the caller of this function. A device manager, for example, can establish a path between a virtual machine process and a device driver's queue.

If a PID or SLIH ID is specified for the To ID, a path is established to the first queue served by the component.

If a device ID is specified, the QID it is associated with is determined from the device directory.

The To ID parameter must be a device ID if I/O subsystem functions, such as CCB pinning, are required. When a device ID is used, the associated device is 'started' when the first user attaches to it (the user's initialization entry point is called).

The acknowledgment information consists of four parts, one of which is the acknowledgment type. The type parameter specifies acknowledgement information to be returned when the processing of a queue element is completed. The four type options are:

None: Acknowledge feature not used
Short: Completion acknowledged by posting an ECB
Long: Completion acknowledged by the return of a queue element
I/O: Completion acknowledged by a virtual interrupt.

Another part of the acknowledgment information is the interrupt depth counter. This parameter places a limit on the number of unsolicited virtual interrupts that can be outstanding at any given time. This parameter, which is valid only when attaching to a device, prevents runaway consumption of queue elements by a device in error situations. If the count is exceeded, the interrupt overrun count is increased. If zero is specified for this parameter, the parameter defaults to one. The largest valid interrupt depth count is 15.

The final two input parameters are for acknowledge information; their meanings are summarized in Figure 5-3.

Acknowledge type	Value	Parameter One	Parameter Two
None	0	n/a	n/a
Short	1	ECB mask	n/a
Long	2	Return QID	Enqueue priority
Interrupt	3	Return QID	Interrupt level and sublevel

Figure 5-3. Acknowledge Parameters

The ECB mask parameter is a 32-bit value with a single bit turned on and the rest zero.

The return QID parameter is a 32-bit value. If the return QID is zero, it defaults to the first queue associated with the From ID server.

Acknowledge parameter two is interpreted as an enqueue priority or as an interrupt level/sublevel value, depending on the acknowledge type. Both are 32-bit quantities, although only the least significant bits are used. The enqueue priority is a number from 0 to 15, occupying the last 4 bits of the parameter. It is described in more detail in "Enqueue Element (_enque, _enq)" on page 5-32. The level/sublevel parameter occupies the last 16 bits. Of these 16 bits, the first 8 are the virtual machine interrupt level (0-7) and the next 8 bits are the virtual machine interrupt sublevel (0-255).

In addition to the return code, the path name parameter (path) is also returned. Path is used by the enqueue function.

Subroutine Call:

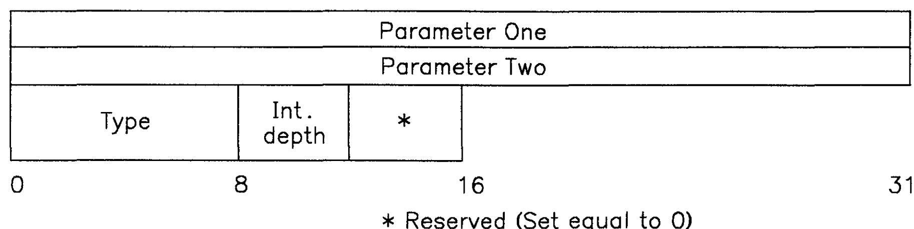
```
Return Code = _attchq (from ID, to ID, path, ack. parms);
```

Calling Register Conventions:

GPR2 = From ID
GPR3 = To ID

GPR4 = Address of the returned path identifier. This word value is word-aligned in memory.
GPR5 = Word-aligned address of acknowledge parameter structure.

The format of an acknowledge parameter structure is defined as follows:



Return Codes: contained in GPR2

-1 = Resources unavailable
0 = Successful
28 = Maximum number of To ID paths exceeded, path not attached.

Termination: The following conditions cause the system to abend:

- The attach function was called by a device driver.
- The caller specified an invalid ID.
- The caller specified an invalid acknowledge parameter.
- The caller specified a duplicate path.

Comments:

Using the same queue for incoming requests and for acknowledgments causes unpredictable results. For example, when a request is on the queue, that request is the 'active' element. When a synchronous **_enque** function is made, **_enque** waits for an active element to arrive in the acknowledgment queue. If the request queue and the acknowledgment queue are the same, **_enque** considers the operation complete due to the presence of the active request element. To avoid this situation, select one of these alternatives:

- Use different queues for requests and acknowledgments.
- Dequeue a request before waiting for an acknowledgment.
- Use a short acknowledgment path so you do not need an acknowledgment queue.

A device driver can request that a path be created if the specified 'from ID' parameter is a SLIH ID, the ID of a queue served by a SLIH, or a device ID associated with a device driver. For paths of this type, the following restrictions apply:

- The acknowledge type parameter must be 'No Ack' because a device driver cannot wait for acknowledgment of a request.

- If the queue associated with the 'to ID' has a check parameters routine, the routine will not be called for requests on this path because the routine may be outside of pinned memory.
- If a request queue element sent on this path is a send command or start I/O type, the memory associated with CCBs or buffers will not be pinned.
- To prevent a device driver from exhausting the VRM control block supply with unlimited requests, the VRM imposes a limit on the number of requests that can be queued concurrently on a path. The limit is defined as follows:
  - If the 'from ID' is a device ID, the limit can be from 1 to 16.
  - If the 'from ID' is a SLIH ID or a queue ID, the limit is 1.

  This path limit is not enforced for processes because the VRM can dynamically expand its supply of control blocks.

Broadcast Virtual Interrupts (_bvint)

Description: The Broadcast Virtual Interrupt function should be used primarily by device drivers. This functions tells all components attached to a device that a virtual interrupt has occurred. The device driver 'broadcasts' acknowledgement queue elements on all paths attached to a device. This function is similar to **_enque**, with the following differences:

- The queue element type **must** specify acknowledgement.

  The acknowledgements are "unsolicited" since they are not directly associated with a queued request.

- You do not have to fill in the path ID field of the queue element.

  The VRM determines the pertinent path IDs by looking at the device driver's queue.

- If a device driver has more than one queue, acknowledgements are sent to the paths attached to the **first** queue.

Note: Even if a device has more than 32 paths attached to it, the broadcast virtual interrupt function sends only 32 acknowledgements.

The format of the queue element is the same as for the **_enque** and **_deque** functions. See "Enqueue Element (_enque, _enq)" on page 5-32.

Subroutine Call:

```
Return Code = _bvint (queue element address);
```

Calling Register Conventions:

GPR2 = The word-aligned address of the queue element

Return Codes: contained in GPR2

-1 = Insufficient resources

0 = Successful.

Cancel Enqueue (_canclq)

Description: The cancel function is intended for abnormal termination conditions. Cancel allows a process to retrieve all elements queued by a particular path. If an element is being processed but has not been dequeued, then it cannot be cancelled. The elements are not returned, they are discarded.

Also, if the queue server is a virtual machine, interrupts can be cancelled. The option parameter determines whether requests, interrupts, or both types of elements should be retrieved. The return code indicates the number of queue elements cancelled.

Subroutine Call:

```
Return Code = _canclq (path, option);
```

Calling Register Conventions:

GPR2 = Path ID
GPR3 = Option (in binary)
01 = requests
10 = interrupts
11 = requests and interrupts.

Return Codes: contained in GPR2

n = Number of requests/interrupts cancelled.

Termination: The following conditions cause the system to abend:

- The cancel function was called by a device driver.
- The caller specified an invalid path name.

Create Queue (_creatq)

Description: Any process can create a queue, either for its own use or for another process or device driver. A device driver cannot create a queue because the allocation of the control blocks may result in a page fault.

After a queue has been created, its control blocks (the queue header and elements) reside in pinned memory. Therefore, page faults never occur as a result of using the queue.

The server's ID is the 32-bit identifier of the VRM component (process or device driver) that serves the queue.

The number of priority levels is a value in the range 0 to 15 that indicates the number of classes into which elements can be divided by the **_enque** function. This value, unrelated to hardware interrupt levels or device priority, controls the order in which queued requests are processed.

One of the returned parameters is the queue identifier (QID). The other returned parameter is the event control bit (ECB), which is used when waiting for the queue. Although the ECB is a number specifying 1 of 32 events, it is returned as a 32-bit mask with a single bit turned on. This bit mask is of the form required for the wait function.

Another input parameter is the maximum number of allowed paths. This value must be in the range 0 to 255 and is used, for example, when creating a queue for a serially-reusable device. The path limit is set to one, indicating that only one process at a time can attach to the queue. Specifying zero for this parameter implies no limit. Typically, this parameter is 1 for serially-reusable devices and 0 for virtual devices, but other values may be used. For example, the virtual terminal manager uses this parameter to control the number of active virtual terminals.

The module ID parameter is required if the queue's server is a device driver. The main entry point of the module is used as the default value for the following entry points:

- I/O initiation, called by:
  - The **_enque** function if the chaining of an element causes the queue to become non-empty
  - The first level interrupt handler (FLIH) if an element has been dequeued and more queue elements need processing.
- Second level interrupt handler (SLIH) called by the FLIH when a hardware interrupt is received.

The **_change** function can be used to change these default entry points, or to specify the following two optional entry points:

- Check parameter — Called when elements are being chained to the queue
- Exception handler — Called to report timeout conditions, if the device timer facility is used for the queue.

Queues served by processes require no entry points, and the MID parameter must be zero.

Subroutine Call:

```
Return Code = _creatq (server ID, priority levels,
                       path limit, MID, QID, ECB);
```

Calling Register Conventions:

GPR2	=	Server ID
GPR3	=	Priority level
GPR4	=	Maximum number of paths
GPR5	=	Module ID
(0(R1))	=	Address of returned queue ID. This word value is word-aligned in memory.
(4(R1))	=	Address of returned ECB mask. This word value is word-aligned in memory.

Return Codes:

-1 = Insufficient resources.
0 = Successful.

Termination: The following conditions cause the system to abend:

- The create queue function was called by a device driver.
- The caller specified an invalid server or module ID.

Create Queue (_creatq)

Description: Any process can create a queue, either for its own use or for another process or device driver. A device driver cannot create a queue because the allocation of the control blocks may result in a page fault.

After a queue has been created, its control blocks (the queue header and elements) reside in pinned memory. Therefore, page faults never occur as a result of using the queue.

The server's ID is the 32-bit identifier of the VRM component (process or device driver) that serves the queue.

The number of priority levels is a value in the range 0 to 15 that indicates the number of classes into which elements can be divided by the **_enque** function. This value, unrelated to hardware interrupt levels or device priority, controls the order in which queued requests are processed.

One of the returned parameters is the queue identifier (QID). The other returned parameter is the event control bit (ECB), which is used when waiting for the queue. Although the ECB is a number specifying 1 of 32 events, it is returned as a 32-bit mask with a single bit turned on. This bit mask is of the form required for the wait function.

Another input parameter is the maximum number of allowed paths. This value must be in the range 0 to 255 and is used, for example, when creating a queue for a serially-reusable device. The path limit is set to one, indicating that only one process at a time can attach to the queue. Specifying zero for this parameter implies no limit. Typically, this parameter is 1 for serially-reusable devices and 0 for virtual devices, but other values may be used. For example, the virtual terminal manager uses this parameter to control the number of active virtual terminals.

The module ID parameter is required if the queue's server is a device driver. The main entry point of the module is used as the default value for the following entry points:

- I/O initiation, called by:
  - The **_enque** function if the chaining of an element causes the queue to become non-empty
  - The first level interrupt handler (FLIH) if an element has been dequeued and more queue elements need processing.
- Second level interrupt handler (SLIH) called by the FLIH when a hardware interrupt is received.

The **_change** function can be used to change these default entry points, or to specify the following two optional entry points:

- Check parameter — Called when elements are being chained to the queue
- Exception handler — Called to report timeout conditions, if the device timer facility is used for the queue.

Queues served by processes require no entry points, and the MID parameter must be zero.

Subroutine Call:

```
Return Code = _creatq (server ID, priority levels,
                       path limit, MID, QID, ECB);
```

Calling Register Conventions:

GPR2 = Server ID
GPR3 = Priority level
GPR4 = Maximum number of paths
GPR5 = Module ID
(0(R1)) = Address of returned queue ID. This word value is word-aligned in memory.
(4(R1)) = Address of returned ECB mask. This word value is word-aligned in memory.

Return Codes:

0 = Successful.
28 = No more queues can be allocated for the specified server ID.

Termination: The following conditions cause the system to abend:

- The create queue function was called by a device driver.
- The caller specified an invalid server or module ID.

Acknowledgement to a general purpose request

Offset				
0	Reserved for system use			
4	Path ID			
8	Type = 0	Reserved *	Flags	Reserved
12	Operation results		User-defined	
16	User-defined			
20	User-defined			
24	User-defined			
28	User-defined			
32				

Figure 5-6. General Purpose Acknowledgement Queue Element

In the preceding figures, the fields marked with an * must contain the first byte of the request element's operation options field if the operations options field is used with the enqueue function.

While checking the type of information in the request element, queue management also examines the ID of the queue's server. If the server is a device driver and the request type is Start I/O or Send Command, the memory containing the CCB or command extension is unpinned.

Therefore, the device driver should not attempt to access a CCB or command extension after the associated request has been dequeued.

If the 'suppress acknowledgement' option is selected, the VRM does not return any information to the sender of the request, nor does it unpin the CCB or command extension. The queue's server is responsible for explicitly generating the acknowledgement (using **_enque**) and unpinning the CCB (using **_upnccb**) or command extension (using **_upinio**).

Note: A process cannot dequeue elements from queues belonging to other processes.

A path to a queue may be destroyed between the time an element is enqueued and dequeued. If this happens, no acknowledgement is generated when the element is dequeued. Instead, the element is discarded with no error.

When a device driver receives a detach queue element, all it has to do is dequeue it. The detach queue element is sent each time a path is detached. The detach queue element simply synchronizes the path detach and the completion of any pending request queue elements. IBM recommends not to use the suppress acknowledgment option when a detach queue element is dequeued because this situation can cause the requestor to become hung.

Subroutine Call:

```
Return Code = _deque  (QID, option, ack. QE);
```

Calling Register Conventions:

GPR2 = Queue ID
GPR3 = Option
- 0 = generate acknowledgement
- 1 = suppress acknowledgement
- 2 = override path level/sublevel

GPR4 = Word-aligned queue element address

(For option 1, if GPR4 = -1, no acknowledgement queue element is returned.)

Return Codes: contained in GPR2

0 = Successful
12 = Error, no element on queue.

Termination: The following conditions cause the system to abend:

- The caller specified an invalid queue ID.
- The acknowledge queue parameter is missing.
- The caller is not the queue's server.
- Buffers were not pinned.

Comments: **Generating virtual interrupts**

When a queue server finishes processing an element, the "Dequeue Element (_deque)" function is typically used to generate an acknowledgement to the sender of the element. However, as noted in "Enqueue Element (_enque, _enq)" on page 5-32, the acknowledge type element can be explicitly enqueued. This explicit enqueueing is particularly useful for generating unsolicited interrupts. Also, when a request has been dequeued with the suppress acknowledge option, enqueueing the acknowledge type element generates a solicited interrupt.

When returning an acknowledge queue element, either with **_enque** or **_deque**, the following information is required:

Byte Offset (decimal)	Value
8	Queue element type (0 = acknowledgement)
10	Flags
	0x80 Software interrupt
	0x40 Timer interrupt
	0x20 Start I/O interrupt
	0x10 Send command interrupt
	0x08 Virtual terminal interrupt
	0x04 Solicited interrupt
	0x02 Reserved
	0x01 Message from another virtual machine
12-13	Operation results.

Other fields, such as device-dependent information, should be filled in if applicable.

The following information, as noted in "Dequeue Element (_deque)" on page 5-25, is filled in automatically. However, if an interrupt is being explicitly enqueued, this information must be supplied by the caller.

Byte Offset (decimal)	Value
4-7	Path ID or zero.
9	First byte of operations option from the request
28-31	Interrupt level/sublevel (optional).

The fields in bytes 14-23 vary depending on the SVC in use. The values for **Start I/O SVC** are defined as follows:

14 - 15	IODN
18 - 19	CCB segment ID
20 - 23	CCB Address

The values for **Send Command SVC** are defined as follows:

14 - 15	IODN
16 - 17	Operation options (the first byte of this field duplicates byte 9)
18 - 19	Command extension segment ID or device dependant data
20 - 23	Command extension address or device dependant data

If the path ID field is zero, queue management enqueues the element on the first (or only) path to the queue's server.

If bit 0 of the interrupt level/sublevel field is zero, or if the input option does not override the level, the level is determined from the path. However, if bit 0 is a one and the input option is set to override (option 2), the path information is overridden. Bits 16-23 contain the interrupt level, and bits 24-31 contain the sublevel.

For solicited interrupts, the first byte of the request's operation options field indicates if:

- The operation is synchronous.
- An acknowledgement should be returned.
- An acknowledgement should be returned only if there has been an error (non-zero operation results).

Destroy Queue (_dstryq)

Description: Any process can destroy any queue. The queue can be destroyed while still in use, and the only validation required to destroy a queue is a valid QID.

The elements in a queue, if any, are dequeued when the queue is destroyed. Acknowledgements are not generated when the elements are dequeued. The process serving the queue is not informed that the queue was destroyed, unless it was waiting for the queue. If the process was waiting, it is returned to the ready state with ECB posted. Also, any paths involving the queue are destroyed.

Queues served by a process are automatically destroyed when the process is destroyed.

Subroutine Call:

```
Return Code = _dstryq (QID);
```

Calling Register Conventions:

GPR2 = Queue ID.

Return Codes: contained in GPR2

0 = Successful.

Termination: The following conditions cause the system to abend:

- The destroy queue function was called by a device driver.
- The caller specified an invalid QID.

Detach Queue (_detchq)

Description: The detach function invalidates the path between two communicating VRM components. The only required parameter is the ID of the path linking the components.

If the To ID in the path being detached is a device ID, the associated device is 'stopped'. A detach type queue element is placed in the device driver's queue. The **_detchq** function does not continue until the device driver dequeues the queue element. At this time, no other elements can be enqueued using this path. This prevents serialization problems because a device must complete all pending requests before detaching from the path. However, this may also cause excessive delay to the caller of **_detchq** if lengthy requests have yet to be processed. You should design device driver interfaces so that all I/O activity is done before you call this service. In addition, your device drivers must recognize request queue element type 4, which is an internal control queue element sent by the VRM to detach a device from a path.

For devices with multiple paths attached, a detach queue element is sent each time a path is detached. The device driver's termination routine is not called until the VRM detaches the last path, however.

Subroutine Call:

```
Return Code = _detchq (path);
```

Calling Register Conventions:

GPR2 = Path identifier.

Return Codes: contained in GPR2

0 = Successful.
4 = Invalid path identifier.

Termination: The system abends if the detach function is called by a device driver.

Enqueue Element (_enque, _enq)

Description: The enqueue function places an element into the target queue. The enqueuer builds a copy of the element to be enqueued and provides this data structure as an input parameter.

This function can be used by a VRM component to simulate the function of a **Start I/O SVC**, a **Send Command SVC**, or a virtual interrupt. Enqueue can also be used for simple process-to-process communication within the VRM. The type parameter in the queue element determines the operation to be performed.

All queue elements are 32 bytes long. The first word (4-bytes) is reserved for system use. The next word specifies the ID of the path on which the element is enqueued. Normally, the component doing the enqueue is the one identified by the From ID in the path. When enqueueing an acknowledgement, however, the path flow is reversed. The enqueueing component is identified by the To ID.

Starting with the third word, the common format is less evident. The first byte of word three specifies the queue element type. Five types of queue elements are defined:

0 - Acknowledgement
1 - Send Command
2 - Start I/O
3 - General Purpose
4 - Control.

VRM services generate control queue elements. The detach queue element is an example of a control queue element.

Note: Check parameter subroutines do not see control queue elements.

The second byte indicates the enqueue priority. The priority value can range from 0 to 15, with 0 designating highest priority. A value of 15 is always interpreted as the lowest possible priority. For example, for a queue defined with three priority levels, specifying 2 or 15 results in an element being queued to the lowest level.

The formats for the three request type queue elements are shown in Figure 5-8 on page 5-33, Figure 5-7 on page 5-33, and Figure 5-9 on page 5-33. Acknowledgement queue elements are shown in Figure 5-4 on page 5-26, Figure 5-5 on page 5-26, and Figure 5-6 on page 5-27. The fourth request type, control, is formatted similar to the general purpose type. The remainder of this discussion primarily involves request queue elements; refer to "Dequeue Element (_deque)" on page 5-25 for further details about acknowledgement elements.

Send Command Queue Element

Offset			
0	Reserved for system use		
4	Path ID		
8	Type = 1	Priority	Options
12	IODN	Command extension segment ID	
16	Parameter 1 (from GPR 3)		
20	Parameter 2 (from GPR 4)		
24	GPR 5 / Command extension address		
28	GPR 6 / Command extension length		
32			

Figure 5-7. Send Command Queue Element Format

Start I/O queue element

Offset			
0	Reserved for system use		
4	Path ID		
8	Type = 2	Priority	Options
12	IODN	CCB segment ID	
16	CCB address		
20	CCB length		
24	Reserved		
28	Reserved		
32			

Figure 5-8. Start I/O Queue Element Format

General purpose queue element

Offset			
0	Reserved for system use		
4	Path ID		
8	Type = 3	Priority	Options
12	User-defined		
16	User-defined		
20	User-defined		
24	User-defined		
28	User-defined		
32			

Figure 5-9. General Purpose Queue Element Format

Bytes three and four of the third word in the element are defined as operation options. The five high-order bits are significant to queue management. The parameter is interpreted as follows:

Bits	**Meaning**
0x8nnn	Acknowledge completion (error or successful).
0x4nnn	Acknowledge completion, errors only.
0x2nnn	Synchronous request.
0x1nnn	Chained control blocks.
0xn8nn	Control operation (VRM only).

The first two bits of the operation options field modify the acknowledge options specified when the queue was attached. For example, if the path specifies interrupt acknowledgement, the enqueue can request no interrupt, interrupt on error only, or unconditional interrupt. If the path does not specify interrupt acknowledgement, the enqueue cannot request an interrupt.

If the synchronous request bit is on, control does not return from the **_enque** function until the request element has been acknowledged. This performs in one step what can be done by enqueueing an element with the synchronous bit turned off, then calling the **_wait** or **_waitq** function (see "Wait for Event or Queue (_wait, _waitq)" on page 5-40).

Note: Only one of the first three operation bits should be turned on at any one time.

The synchronous bit is ignored if the path type specifies that no acknowledgement should be generated (path type = 0).

The chained control blocks bit is used in conjunction with start I/O and send command request elements only. For start I/O, the bit indicates one or more command elements in addition to the CCB command header. For send command requests, it means that there is a command extension. Also, if the server of the queue is a device driver, the memory containing the chained control blocks is pinned. This includes buffers referenced in the command elements.

I/O buffers used by the Send Command and Start I/O versions of this service can reside in bus I/O memory (addresses from 0xF4000000 to 0xF4FFFFFF). The segment ID of this area is 0xFFFF. For Start I/O requests, each command element of the CCB must contain the segment ID in bits 16-31 of the first word in the element.

The control bit is significant only in the control (type 4) queue element. If the control bit is set on, the remaining bits of the operations field specify the control operation. Currently, the only defined value is zero, which signifies a detach operation. An interrupt is not generated when a control element is dequeued, but the ECB associated with the acknowledgement queue is posted.

Queue management notifies the queue's server, if necessary, after an element is enqueued. This may involve posting an ECB or calling an I/O initiate routine.

Note: You can call this service in two ways (**_enq** and **_enque**). The only difference between the two calls is that **_enq** provides an error return code if the specified path ID is invalid. The **_enque** call abends the system when it encounters an invalid path ID.

Subroutine Call:

```
Return Code = _enque  (queue element);
                   or
Return Code = _enq (queue element);
```

Calling Register Conventions: (same for both calls)

GPR2 = Word-aligned address of the queue element.

Return Codes: contained in GPR2

- -1 = Unable to pin a Start I/O or Send Command queue element due to insufficient real memory, page pinned more than 255 times, or path limit restriction for SLIHs exceeded.
- -32768 = Operation cancelled.
- 0 = Successful.
- 4 = Invalid path ID (**_enq** only).
- 8 = Unable to pin Start I/O or Send Command buffer or CCB due to invalid segment identifier.
- 12 = Unable to pin Start I/O or Send Command buffer or CCB because of incorrect alignment or invalid length.
- 20 = Command control block is not word-aligned.
- ≥256 = Return code generated by check parameter routine.
- <0 = Results of a synchronous operation (operation results from acknowledgement queue element, sign extended).

Termination: The following conditions cause the system to abend:

- The caller specified an invalid type parameter.
- The caller specified an invalid path (**_enque** only):
  - Path ID invalid.
  - Path ID of zero not accepted.
  - Path used in wrong direction.

Peek at Queue Element (_peekq)

Description: This subroutine allows you to browse the contents of any but the top element in a queue without removing the element from the queue. Use "Read Queue Element (_readq)" on page 5-39 to browse the top element in a queue.

To peek at a queue element, specify the offset from the top of the queue of the element you wish to browse. To browse the next element to be processed after the current element, specify one as the queue element offset. The element will be copied to a space where it can be browsed.

The copied queue element includes the path ID used by the sender to enqueue the element. This value can be used with "Query Attached Path (_queryp)" on page 5-38 routine to determine the component that enqueued the element.

Although most queues process enqueued elements in first-in first-out order, some queues do not. Note that, for queues not processed FIFO, the order of the elements may change from the time you browsed an element at a particular offset. An example of a queue that does not process elements FIFO is a fixed-disk queue that uses arm scheduling algorithms to sort enqueued elements.

Subroutine Call:

```
Return Code = _peekq (QID, offset, addr. of copied QE);
```

Calling Register Conventions:

GPR2 = Queue ID
GPR3 = Offset from the top of the queue
GPR4 = Address of the copied queue element.

Return Codes: contained in GPR 2

0 = Successful
-1 = No queue element found at the specified offset
-4 = Invalid offset (offset must be greater than 0).

Termination: The following conditions cause the system to abend:

- The caller specified an invalid queue ID.

Post ECB (_post)

Description: The post event control bit (ECB) function is a primitive queue management operation used for notifying processes that are waiting for a particular event. A post is performed automatically during an enqueue and during a dequeue if a queue was attached with the acknowledge option.

Processes and device drivers can explicitly perform a post if the higher level queue functions are not required and their associated overhead is undesirable.

The ECB numbers must be known by the cooperating components, either through programming convention or passing of initialization parameters.

Note: Bits 24-31 are reserved for use by queue management and should not be used. Also, as queues are created for a process, queue management starts with the higher numbered ECBs and works downward towards zero. For example, the first queue served by a process is assigned bit 23, the second is assigned bit 22, the next is assigned bit 21, and so on.

Subroutine Call:

```
Return Code = _post (ECB, PID);
```

Calling Register Conventions:

GPR2 = ECB mask
GPR3 = Target process ID.

Return Codes: contained in GPR 2

0 = Successful.

Termination: The following conditions cause the system to abend:

- The caller specified an invalid process ID.

Query Attached Path (_queryp)

Description: The query function reveals the existence of a path established by the attach function and returns the path name or the IDs associated with a valid path ID. To retrieve a path ID, a From ID and To ID are input. These two IDs must be the two that were used to create the path. If a path ID is input, the From and To IDs must be zero. Conversely, if a From and To ID are input, the path ID must be zero.

A device driver can use this service, but only to pass a path ID.

Other returned information concerning the path includes the acknowledge type information and the To QID. Refer to "Attach Queue (_attchq)" on page 5-17 for details about these parameters.

Subroutine Call:

```
Return Code = _queryp (path data);
```

Calling Register Conventions:

GPR2 = Address of query path structure.

This structure is shown in Figure 5-10.

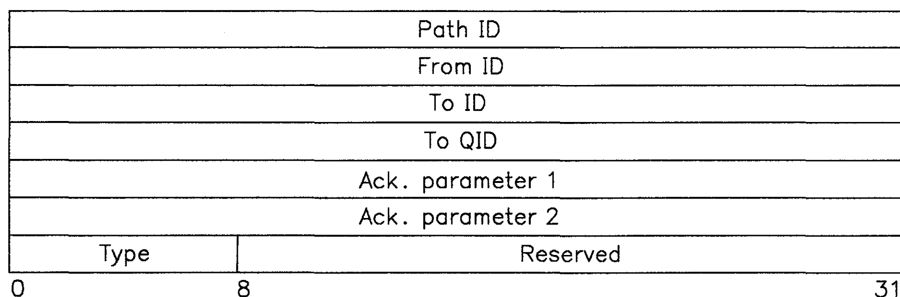


Figure 5-10. Query Path Structure

Return Codes: contained in GPR2

0 = Successful
4 = Invalid ID, or invalid parameters passed by a device driver
12 = Path doesn't exist.

Read Queue Element (_readq)

Description: The read function allows you to look at the contents of the top element in a queue without removing the element from the queue.

The path ID used by the sender to enqueue the element is returned in the queue element. This value can be used as input to the query attached path function to determine who sent the element.

When a queue element is read using **_readq**, it is marked as busy.

Subsequent **_readq** calls will read the same element. The element remains on top of the queue until dequeued. Any elements enqueued after the **_readq** but before the **_deque** are queued behind the busy element, even if they have a higher priority.

See the description of "Enqueue Element (_enque, _enq)" on page 5-32 for the formats of the various queue elements.

Subroutine Call:

```
Return Code = _readq (QID, queue element);
```

Calling Register Conventions:

GPR2 = Queue ID
GPR3 = Word-aligned address of returned queue element.

Return Codes: contained in GPR2

0 = Successful
12 = Error, no element on queue.

Termination: The VRM abends the system if the caller specifies an invalid queue ID.

Wait for Event or Queue (_wait, _waitq)

Description: The wait function forces a component to wait for the use of a queue or the occurrence of an event. The two forms are specific and general wait.

Specific wait is used to wait for a particular queue, and a process can wait for only one queue at a time. When the process is dispatched after the wait, a copy of the highest priority element in the queue is provided as a return parameter. The queue's ID determines which ECB to await.

With the general wait, however, a process can wait for one or more ECBs. The input parameter is a 32-bit mask, with each bit set equal to one representing an ECB to await. If the general wait is used to wait for an ECB associated with a queue, the process is notified when an element is enqueued. The returned mask indicates which event occurred, and the **_readq** function can be used to obtain the appropriate queue element.

The second parameter for the general wait is a mask of ECBs to clear when the wait is complete.

Note: If the ECBs are associated with queues, they should not be cleared. Those ECBs are the responsibility of queue management. However, you must clear the ECBs that are not associated with queues.

Subroutine Call:

```
Return Code =
 _wait (wait mask, clear mask, ret'd. ECB mask);
   - or -
 _waitq (QID, queue element);
```

Calling Register Conventions:

_wait

GPR2 = Mask of ECBs to await
GPR3 = Mask of ECBs to clear
GPR4 = Address of returned ECB mask. This word value is word-aligned in memory.

_waitq

GPR2 = Queue ID
GPR3 = Address of returned queue element. This word value is word-aligned in memory.

Return Codes: contained in GPR2

0 = Successful.

Termination: The following conditions abend the system:

- The wait function was called by a device driver.
- The caller specified an invalid queue ID.

Memory Management

The VRM has several mechanisms to control system memory. The memory management runtime routines include:

- Attach segment *
- Convert effective address to real address
- Convert virtual address to real address
- Detach segment *
- Load byte *
- Load halfword *
- Load segment register
- Load word *
- Move buffer *
- Pin page range
- Query segment ID
- Restore segment register
- Save segment register
- Store byte *
- Store halfword *
- Store word *
- Unpin command control block
- Unpin page range.

The routines followed by an asterisk (*) should be used to avoid system abends or data corruption caused by competing processes using the same segment. If a process destroys a segment while another process is writing into the segment, one of two things can happen. They are:

- A system abend will result if a process references a segment that no longer exists.
- Data corruption may result if the segment ID is immediately reallocated. In this case, the original process is writing to the same segment ID, but is actually writing over whatever data may be in the segment.

Although the **attach segment** and **detach segment** functions do not prevent a process from destroying a segment used by another process, they can prevent data corruption. The **attach segment** call prevents a segment ID from being reallocated if the segment is destroyed. This feature prevents corruption of another user's data. The segment ID of a segment that is destroyed while attached cannot be reallocated until **detach segment** calls bring the segment's attach count to zero.

The various **load** and **store** functions discussed here prevent system abends when a segment is destroyed while being read or written. These routines issue return codes (instead of abending the system) to indicate a reference to an invalid address.

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Attach Segment (_attchs)

Description: This routine prevents a segment ID from being reallocated when a segment in use by multiple processes is destroyed by one of the processes. Each segment ID has an attach count that indicates the number of attaches to the segment. When a process attaches to a segment, the attach count is incremented by one. As long as the attach count is greater than zero, the segment ID cannot be reallocated (even though the segment may have been destroyed). This maintains the integrity of data in the system because the segment ID cannot be reused until **detach segment** routines bring the attach count to zero (see "Detach Segment (_detchs)" on page 5-46).

This routine cannot be called by device drivers.

Subroutine Call:

```
Return Code = _attchs (segment ID);
```

Calling Register Conventions:

GPR2 = Segment ID (right justified)

Return Codes: contained in GPR2

-1 = Attach count exceeds 255
0 = Successful
8 = Invalid segment ID.

Convert Effective Address to Real Address (_cveara)

Description: This conversion function accepts a 32-bit effective address and returns its real address. An effective address is a 32-bit value of which the high-order 4 bits indicate the segment register and the low-order 28 bits specify the displacement into the segment. The low order 11 bits (page offset) are the same in both the effective address and the real address.

Subroutine Call:

```
Return Code = _cveara (eff. addr., real addr.);
```

Calling Register Conventions:

GPR2 = Effective address
GPR3 = Address of the returned real address. The returned address is a word value that is word-aligned in memory.

Return Codes: contained in GPR2

0 = Successful
8 = Effective address not found in memory.

Convert Virtual Address to Real Address (_qra)

Description: This conversion function accepts an effective address and a segment ID and returns the corresponding real address. Use this function instead of "Convert Effective Address to Real Address (_cveara)" when the segment is not currently loaded into a segment register. The four most significant bits of the effective address are ignored by this function.

Subroutine Call:

```
Return Code = _qra  (segment ID, eff. addr., real addr.);
```

Calling Register Conventions:

GPR2 = Segment ID
GPR3 = Effective address
GPR4 = Address of returned real address. The returned address is a word value that is word-aligned in memory.

Return Codes: contained in GPR2

0 = Successful
8 = Effective address not found in memory.

Detach Segment (_detchs)

Description: This routine decrements a segment ID's attach count by one. The VRM will not reassign a segment ID with an attach count greater than zero. Any time a segment's attach count equals zero and the segment is destroyed, the segment ID can be reused immediately.

This routine cannot be called by device drivers.

Subroutine Call:

```
Return Code = _detchs (segment ID);
```

Calling Register Conventions:

GPR2 = Segment ID (right justified)

Return Codes: contained in GPR2

-1 = Attach count underflow (more **detchs** routines issued than **attchs** routines)
0 = Successful
8 = Invalid segment ID.

Load Byte (_loadb)

Description: This routine loads a byte of data from the specified address. If the specified address is valid, the low-order byte of GPR2 on return will contain the desired value. If the address was invalid, GPR2 will contain a return code indicating this.

This operation does not check the input effective address and assumes the segment register is loaded with the proper segment ID.

This function cannot be called by device drivers.

Subroutine Call:

```
Return Code or Data = _loadb (effective address);
```

Calling Register Conventions:

GPR2 = Contains the address of the byte to load

Return Codes: contained in GPR2

If the address you attempted to load from is valid, the 3 high-order bytes of GPR2 contain zero and the low-order byte contains the desired value.

-1 = Input effective address invalid.

Load Halfword (_loadh)

Description: This routine loads a halfword of data from the specified address. If the specified address is valid, the low-order halfword of GPR2 on return will contain the desired value. If the address was invalid, GPR2 will contain a return code indicating this.

This routine has the same storage-alignment characteristics as the hardware load halfword instruction.

This operation does not check the input effective address and assumes the segment register is loaded with the proper segment ID.

This function cannot be called by device drivers.

Subroutine Call:

```
Return Code or Data = _loadh (effective address);
```

Calling Register Conventions:

GPR2 = Contains the address of the halfword to load

Return Codes: contained in GPR2

If the address you attempted to load from is valid, the high-order halfword of GPR2 contains zero and the low-order halfword contains the desired value.

-1 = Input effective address invalid.

Load Segment Register (_lsr)

Description: The load segment register function **should** be used to change segment registers. Although segments can be changed with certain machine-level instructions, the changes may revert to the previous value after a redispatch/restart because the VRM was not informed of the transaction. The dispatcher does not save segment registers, so this service is used to maintain the current values of each segment register. In this way the correct values are used when a process or device driver is restarted. The load segment register function is unaffected by page fault restrictions. The passed-in segment ID, segment register number, and the protection value are right-justified.

Note: Only segment registers 1 through 13 should be changed by **_lsr**. Never change the value in segment registers 0, 14 or 15.

For more information on the use of segment registers, see "Segment Register Utilization" on page 3-21.

Subroutine Call:

```
Return Code = _lsr  (segment ID, seg. reg.#, protection);
```

Calling Register Conventions:

GPR2 = Segment ID
GPR3 = Segment register number
GPR4 = Protection
- 0 = Page-level protection with unprivileged bit cleared
- 1 = Page-level protection with unprivileged bit set.

Return Codes: contained in GPR2

0 = Successful.

Load Word (_loadw)

Description: This routine loads a word of data from the specified address. If the address is invalid, a return code of -1 is passed back.

This routine has the same storage-alignment characteristics as the hardware load word instruction.

This operation does not check the input effective address and assumes the segment register is loaded with the proper segment ID.

This function cannot be called by device drivers.

Subroutine Call:

```
Data = _loadw (effective addr., pointer to return code);
```

Calling Register Conventions:

GPR2 = Contains the address of the word to load
GPR3 = Contains the address of the return code.

Return Data: contained in GPR2

If the return code (pointed to by GPR3) is zero, GPR2 contains the desired value. If the return code is -1 (meaning the input effective address was invalid), the contents of GPR2 on return are unpredictable.

Move Buffer (_mvbuff)

Description: This routine copies a buffer of any storage alignment into another buffer. The source buffer will not be altered unless the target buffer overlaps it. For example, if the source buffer is specified as 100 bytes beginning at address 00 and the copy is to be placed at address 90, 10 bytes of the source buffer will be written over by the **move buffer** routine.

If any of the addresses in either the source or target buffers are invalid, the system does not abend. Instead, GPR2 on return contains a -1 to indicate an invalid buffer address.

Note that the length of the target buffer is assumed to be equal to the length of the source buffer.

This operation does not check either input buffer address and assumes the segment registers (if different for the two buffers) are loaded with the proper segment IDs.

This function cannot be called by device drivers.

Subroutine Call:

```
Return Code = _mvbuff (target buff., res'd, source buff., length);
```

Calling Register Conventions:

GPR2 = Contains the effective address of the target buffer
GPR3 = Reserved
GPR4 = Contains the effective address of the source buffer
GPR5 = Contains the length of the source buffer.

Return Codes: contained in GPR2

-1 = Either source or target buffer address invalid
0 = Successful.

Pin Page Range (_pinpgs)

Description: The pin pages function ensures that page faults do not occur for a range of pages. A device driver cannot call this function.

Subroutine Call:

```
Return Code = _pinpgs (seg.ID, first page, # of pages);
```

Calling Regsister Conventions:

GPR2 = Segment ID of the segment containing the pages.
GPR3 = Page number of the first page to pin.
GPR4 = Number of pages to pin.

Return Codes: contained in GPR2

-1 = Unable to pin due to insufficient real memory or page pinned more than 255 times
0 = Successful completion
8 = Invalid segment identifier
12 = First page or number of pages invalid.

Query Segment ID (_qsid)

Description: The query segment ID function accepts a 32-bit effective address and returns its segment ID. The segment register number is simply the high-order 4 bits of the effective address, and the segment offset is the low-order 28 bits of the effective address.

Subroutine Call:

```
Return Code = _qsid  (eff.addr., SID);
```

Calling Register Conventions:

GPR2 = Effective address
GPR3 = Address of the returned segment ID. The segment ID is a halfword value that is halfword-aligned in memory.

Return Codes: contained in GPR2

0 = Successful
4 = Unknown effective address.

Restore Segment Register (_rsr)

Description: The restore function replaces the contents of a segment register saved with **_ssr** (see "Save Segment Register (_ssr)" on page 5-55). The segment register number should be in the range 1 through 10.

For more information on segment registers, see "Segment Register Utilization" on page 3-21.

Subroutine Call:

```
_rsr  (seg. reg. #, seg. reg. contents);
```

Calling Register Conventions:

GPR2 = Segment register number
GPR3 = Segment register contents.

Return Codes: none.

Save Segment Register (_ssr)

Description: The save function temporarily holds the contents of a segment register that will later be restored with **_rsr** (see "Restore Segment Register (_rsr)" on page 5-54). The segment register number should be in the range 1 through 10.

Subroutine Call:

```
Segment Register Value = _ssr (segment register number);
```

Calling Register Conventions:

GPR2 = Segment register number.

Return Codes: contained in GPR2

On return, GPR2 contains the segment register contents.

Store Byte (_storeb)

Description: This routine stores a byte of data into the address specified. If the specified address is invalid, the system does not abend. Instead, GPR2 contains a -1 on return to indicate this condition.

This operation does not check the input effective address and assumes the segment register is loaded with the proper segment ID.

This function cannot be called by device drivers.

Subroutine Call:

```
Return Code = _storeb (effective address, data);
```

Calling Register Conventions:

GPR2 = Contains the address at which to store the byte
GPR3 = Contains the byte to store, right justified

Return Codes: contained in GPR2

-1 = Input effective address invalid
0 = Successful

Store Halfword (_storeh)

Description: This routine stores a halfword of data into the address specified. If the specified address is invalid, the system does not abend. Instead, GPR2 contains a -1 on return to indicate this condition.

The **store halfword** function has the same storage alignment characteristics as the hardware store halfword instruction.

This operation does not check the input effective address and assumes the segment register is loaded with the proper segment ID.

This function cannot be called by device drivers.

Subroutine Call:

```
Return Code = _storeh (effective address, data);
```

Calling Register Conventions:

GPR2 = Contains the address at which to store the halfword
GPR3 = Contains the halfword to store, right justified

Return Codes: contained in GPR2

-1 = Input effective address invalid
0 = Successful

Store Word (_storew)

Description: This routine stores a word of data into the address specified. If the specified address is invalid, the system does not abend. Instead, GPR2 contains a -1 on return to indicate this condition.

The **store word** function has the same storage alignment characteristics as the hardware store word instruction.

This operation does not check the input effective address and assumes the segment register is loaded with the proper segment ID.

Subroutine Call:

```
Return Code = _storew (eff. address, data);
```

Calling Register Conventions:

GPR2 = Contains the address at which to store the word
GPR3 = Contains the word to store

Return Codes: contained in GPR2

-1 = Input effective address invalid
0 = Successful

Unpin Command Control Block (_upnccb)

Description: The unpin command control block function unpins the specified CCB and any data areas pointed to by the CCB.

A device driver must issue this routine when it enqueues an acknowledgment for a command that had been dequeued earlier without an acknowledgment. When a command is dequeued with acknowledgment, the **_dequeue** function automatically unpins the CCB for a device driver.

Subroutine Call:

```
Return Code = _upnccb (CCB seg.ID, CCB eff. addr.);
```

Calling Register Conventions:

GPR2 = Segment ID of the segment containing the CCB
GPR3 = Address of the word-aligned CCB.

Return Codes: contained in GPR2

-1 = Page not pinned.
0 = Successful completion.
8 = Invalid segment identifier.
12 = First page or number of pages invalid.

Unpin I/O Buffer (_upinio)

Description: This routine is used to unpin command extensions that were pinned by queue management for Send Command queue elements. These command extension areas are normally unpinned when the request is dequeued. However, if the request is dequeued with acknowledgment suppressed, the command extension is not unpinned and must be unpinned explicitly with this routine.

Note: The **_unpin** routine does not unpin command extensions.

Subroutine Call:

```
Return Code = _upinio  (seg.ID, start address, # of bytes);
```

Calling Register Conventions:

GPR2 = Segment ID of the subject segment
GPR3 = Starting address of the area to unpin
GPR4 = Number of bytes to unpin.

Return Codes: contained in GPR2

-1 = Pin count underflow
0 = Successful completion
8 = Invalid segment identifier
12 = Start address or number of bytes invalid.

Comments: All the pages that the specified buffer resides in are unpinned. Only the low-order 28 bits of the start address are processed (the segment register number field is ignored).

Unpin Page Range (_unpin)

Description: The unpin function decreases the count of pin page range calls for each page in the range. If the count becomes zero, the corresponding page may be removed from primary memory.

Subroutine Call:

```
Return Code = _unpin  (seg.ID, first page, # of pages);
```

Calling Register Conventions:

GPR2 = Segment ID of the segment containing the pages
GPR3 = First page to unpin
GPR4 = Number of pages to unpin.

Return Codes: contained in GPR2

-1 = Page not pinned
0 = Successful completion
8 = Invalid segment identifier
12 = First page or number of pages invalid.

Semaphore Management

The VRM has several mechanisms to control semaphores. The semaphore management runtime routines include:

- Create semaphore
- Destroy semaphore
- Receive semaphore
- Send semaphore.

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and possible return codes.

Create Semaphore (_creats)

Description: The create function establishes the existence of a semaphore. A 32-bit semaphore ID is returned if the function is successful.

Any process can create a semaphore, which can then be used by any other process.

A semaphore's count can be initialized to any value by the input parameter.

Subroutine Call:

```
Return Code = _creats (initial value, semaphore ID);
```

Calling Register Conventions:

GPR2 = Initial value of the semaphore
GPR3 = Address of the returned semaphore ID. The semaphore ID is a word value that is word-aligned in memory.

Return Codes: contained in GPR2

-1 = Insufficient resources
0 = Successful.

Termination: The VRM abends the system if the create function is called by a device driver.

Destroy Semaphore (_dstrys)

Description: The destroy function removes a semaphore from the VRM. Any process can destroy any semaphore. If any processes are waiting for the semaphore, the destroy operation is unsuccessful. As long as the semaphore ID is valid and no processes are waiting for the semaphore, a semaphore can be destroyed.

Subroutine Call:

```
Return Code = _dstrys (semaphore ID);
```

Calling Register Conventions:

GPR2 = Semaphore ID.

Return Codes: contained in GPR2

0 = Successful
20 = Unsuccessful, processes awaiting semaphore.

Termination: The following conditions cause the system to abend:

- The destroy function was called by a device driver.
- The caller specified an invalid semaphore ID.

Receive Semaphore (_recv)

Description: The receive function performs one of two options based on the count of the semaphore. If the count is zero, the requesting process is placed into the wait state and is queued to the semaphore control block. If the count is not zero, the count is decreased by 1 and the requesting process is allowed to remain in the running state.

Subroutine Call:

```
Return Code = _recv (semaphore ID);
```

Calling Register Conventions:

GPR2 = Semaphore ID.

Return Codes: contained in GPR2

0 = Successful.

Termination: The following conditions cause the system to abend:

- The receive function was called by a device driver.
- The caller specified an invalid semaphore ID.

Send Semaphore (_send)

Description: The send function either increases a semaphore's count if no processes are awaiting the semaphore, or places the highest-priority waiting process into the ready state.

Subroutine Call:

```
Return Code = _send (semaphore ID);
```

Calling Register Conventions:

GPR2 = Semaphore ID.

Return Codes: contained in GPR2

0 = Successful.

Termination: The following conditions cause the system to abend:

- The **_send** function was called by a process' exception handler or by a device driver.
- The caller specified an invalid semaphore ID.

Timer Management

The VRM contains several mechanisms to control the timer. The timer management runtime routines include:

- Cancel interval timer
- Control device timer
- Set device timer
- Set interval timer
- Wait for interval timer

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Programmer's Note

Timer interrupts **cannot** be used to break a 'spin loop' in a device driver or process. Due to adverse effects on system performance, you should **never** use spin loops. Depending on the length of time required to wait for an event, one of the following methods should be used to signal completion of the event:

- If the time to wait is approximately 1 millisecond or less, use a loop with a finite counter.
- If you require more wait time, set the interval timer. The device manager should then use the **_sleep** function to wait for the interval timer to expire, and device driver functions (subroutines) should return to their caller.

Cancel Interval Timer (_cnltmr)

Description: This function cancels an interval timer request. The timer ID parameter determines the timer request to cancel.

In a situation where a cancel request is made and the timer interval expires before the cancel is completed, the exception handler routine is called. After that routine completes (returns), processing of the cancel request continues. Because the timer ID will not be found, the cancel will fail.

Subroutine Call:

```
Return Code = _cnltmr (timer ID);
```

Calling Register Conventions:

GPR2 = Timer ID.

Return Codes: contained in GPR2

0 = Successful
12 = Matching request not found.

Control Device Timer (_ctldvt)

Description: The device timer is automatically started by the enqueue function and cleared by the dequeue function. If explicit control of the device timer is required, the control device timer function can be used. The three options are:

- Stop
- Restart
- Stop and restart.

The options can be specified together or separately. The restart option resets the timer interval to its original value.

Subroutine Call:

```
Return Code = _ctldvt (option, QID);
```

Calling Register Conventions:

GPR2 = Option (in binary)
  01 = restart
  10 = stop
  11 = stop and restart.
GPR3 = Queue ID.

Return Codes: contained in GPR2

0 = Successful.

Termination: The following conditions cause the system to abend:

- The caller specified an invalid queue ID.
- The device timer is not set for this queue.

Set Device Timer (_setdvt)

Description: The set device timer function detects timeout conditions in a device driver. Specifically, set device timer generates a call to an exception handler if a queue element is not processed within a specified interval.

The three parameters are the time interval, the signal mask to be used for a timer ID, and the queue that you want to time. This timer needs to be set only once. When set, the timer starts and stops automatically.

Note: The time interval granularity is different from the "Set Interval Timer (_settmr)" function. The interval is specified in units of 500 milliseconds.

The time intervals and mask values can be changed by calling this function again. The new input values override the existing values.

Subroutine Call:

```
Return Code = _setdvt (interval, mask, QID);
```

Calling Register Conventions:

GPR2 = Time interval
GPR3 = Timer ID (signal mask)
GPR4 = Queue ID.

Return Codes: contained in GPR2

-1 = Insufficient resources
0 = Successful.

Termination: The following conditions cause the system to abend:

- The set function was called by a process.
- The caller specified an invalid parameter:
  - Invalid queue ID
  - No exception handler specified.

Set Interval Timer (_settmr)

Description: The set timer function requests interruption of a process or device driver when a specified interval of time has expired. You can receive notification of timer expiration in one of three ways. They are:

- The requestor can have an ECB set with the **_post** function.
- The requestor can be sent a queue element when the interval expires.
- The requestor's exception handler entry point can be called when the interval expires.

If the requestor is a process and you choose the latter option, the notification works like an interrupt (see "Exception Handling" on page 3-16). If you request the latter option for a device driver, the driver's exception handler routine is called synchronously from the timer's SLIH.

If you request notification by ECB, the timer ID is an ECB mask with a bit set to identify the request to the caller. See "Post ECB (_post)" on page 5-37.

You can also specify a request for continuous notification, which automatically refreshes the request after the interval expires. A 32-bit value is specified for the timer interval, and the unit of timer granularity is 975.562 microseconds.

If you request signal notification, the timer ID parameter is a signal mask (see "Signal (_signal)" on page 5-14), which is passed to the exception handler when the interval expires. This mask is also used to identify a request being cancelled.

If you choose to be notified of timer expiration by way of a queue element, the requestor's queue (or, in the case of multiple queues, the primary queue) will receive a general purpose queue element. The first of the five user-defined data words will contain the timer ID, and the rest of the user-defined data words are set equal to zero.

The option flag parameter contains three bit flags. If the leftmost bit equals one (0x80), the request is continuous. If this bit equals zero, only one notification is generated. If the second bit equals one (0x40), the requestor will receive a queue element. If the third bit equals one (0x20), the caller will be posted. If the flag field equals 0x00 or 0x80, the requestor's exception handler routine will be called.

Subroutine Call:

```
Return Code = _settmr (interval, flag, timer ID);
```

Calling Register Conventions:

GPR2 = Timer interval
GPR3 = Flag
  0x20 = notify by ECB
  0x40 = notify by queue element
  0x80 = continuous notification
GPR4 = Timer ID (signal or ECB mask).

Return Codes: contained in GPR2

-1 = Insufficient resources
0 = Successful.

Wait for Interval Timer (_sleep)

Description: The wait function, unavailable to device drivers, places the process in the wait state until the timer interval expires.

Even if a queue element arrives in one of the process' queues while the process is waiting for the timer, the process will not re-enter the ready state. Note that if **_settmr** requested a queue element, you should use **_wait** or **_waitq** to wait for the timer to expire, not **_sleep**.

Subroutine Call:

```
Return Code = _sleep;
```

Return Codes:

0 = Successful.

Termination: The VRM abends the system when the wait function is called by a device driver.

Program Management

The VRM contains several mechanisms for controlling its programs. The program management runtime routines include:

- Allocate memory
- Bind module
- Copy module
- Free memory
- Query module ID.

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Allocate memory (_mallc)

Description: The allocate memory function is used to dynamically allocate storage from the VRM data heap.

This function cannot be called by device drivers.

Subroutine Call:

```
Return code = _mallc (length, alignment)
```

Calling Register Conventions:

GPR2 = Length in bytes of the requested storage
GPR3 = n, where 2^n is the alignment of the requested storage.

Return Codes: contained in GPR2

Upon successful completion of the allocate memory function, GPR2 contains the address of the storage area allocated.

-1 = Insufficient resources.

Bind Module (_bind)

Description: The bind module function resolves the external references to entry points or data.

The import symbols of the import module are compared to the export symbols in the export modules. If a match is found, the address of the symbol is updated in the import module. The symbols array provides an optional rename-like capability. Here the import symbol is resolved to the specified export symbol name which may or may not be the same. The symbols array returns the address of the symbol. The symbol will be searched for in all of the export modules if the index is -1, otherwise only the specified export module is searched.

You can query the address of a routine or data area in an export module without binding to the module by specifying 0 as the import module ID.

The bind function cannot be called by code executing on a hardware interrupt level.

Subroutine Call:

```
Return code =  _bind (import MID, export MIDs, # of array elements,
                      addr. of symbols list, length of symbols list);
```

Calling Register Conventions:

GPR2 = ID of the module to be bound
GPR3 = Address of the array containing module IDs. This is a 32-bit wide structure of variable length that contains the 32-bit module IDs.
GPR4 = Number of elements in module ID array
GPR5 = Address of the symbols list (shown in Figure 5-11). (If the value of this register is -1, then no symbol list exists.)
(0(R1)) = Length of the symbols list.

<table>
<tr><td colspan="3">Index</td></tr>
<tr><td colspan="3">Address</td></tr>
<tr><td>From length</td><td>To length</td><td>From name</td></tr>
<tr><td colspan="2">To name</td><td>. . . . .</td></tr>
</table>

0 31

Figure 5-11. Symbols List for _bind. Note that both the 'From name' and 'To name' can be from 4 to 16 characters in length. Subsequent entries in the symbols list follow immediately with no padding.

Note: An invalid symbol list may cause the system to abend or may produce unpredictable results.

Return Codes: contained in GPR2

0 = Successful completion
4 = Invalid module IDs specified
12 = Unresolved external symbol.

Copy Module (_copy)

Description: The copy module function serves two purposes. They are:

- To control the deletion of modules
- To ensure that each user has a unique copy of a serially-reusable module.

Each time you specify the add operation, the access count increases by one. Each time you specify the delete operation, the count is decreased by one. An access count of zero indicates that the module is not in use and can be deleted.

Program management does not automatically delete a module. A module is deleted only when the virtual machine issues an SVC and the access count is zero (no devices using the module). See "Define Code SVC" on page 4-49 for more information.

All users share a single copy of a re-entrant module. The module identifier returned by this routine is always the same as the copied module's identifier. The copy module service can be used to ensure that a virtual machine cannot delete a re-entrant module that is in use.

Each time a device manager creates a virtual device, the manager should issue this service to increase the access count associated with the module. The device manager should also issue this service to decrease the access count upon termination of the virtual device. The module cannot be deleted when in use since its access count is not zero.

The copy module service functions in a somewhat different manner when copying a serially-reusable module. The first time this service is called to copy a module, the module is simply protected from deletion. No physical copy of the module is produced. A subsequent call to **_copy** creates a physical copy of the module's read/write section. This ensures that each process has a private copy of the module.

A module can be deleted only when a virtual machine issues a **Define Code SVC** with the delete option and the module has not been protected by an initial call to this service.

Modules copied in this manner have unique module identifiers (MID) and an access count of one. The original module is always maintained, even if its access count is zero.

A device manager should issue **_copy** at least once to protect a module from deletion. The manager should also issue this service for each virtual device that it creates. The copy function increases the access count associated with the module and ensures that each virtual device has its own private data area. The copy function should also be used to decrease the access count upon termination of the virtual device. The module cannot be deleted when it is in use since its access count is not zero.

Note: The copy module service may not be called by code executing on a hardware interrupt level.

Subroutine Call:

```
Return Code= _copy (operation, MID, new MID)
```

Calling Register Conventions:

GPR2 = Operation
  0 = delete
  1 = add.
GPR3 = ID of the module to copy
GPR4 = Address of the returned new module ID. The new module ID is a word value that is word-aligned in memory. (If the value in this register is -1, no module ID is returned.)

Return Codes: contained in GPR2

-1 = Insufficient resources
0 = Successful completion
4 = Invalid module ID specified
32 = Not deleted, module in use.

Free Allocated Memory (_mfree)

Description: The free allocated memory function is used to free storage that was previously allocated from the VRM data heap with the **_mallc** function.

This function cannot be called by device drivers.

Subroutine Call:

```
_mfree (address, length)
```

Calling Register Conventions:

GPR2 = Address of the storage to be freed
GPR3 = Length in bytes of the storage to be freed.

Return Codes: none.

Query Module ID (_querym)

Description: The query module ID function is used to determine the module identifier (MID) associated with an IOCN. More than one module identifier may be associated with an IOCN. For example, each copy of a serially-reusable module has a common IOCN but a unique module identifier. In this case, the query service returns the module identifier of the original IOCN that was copied..

This service cannot be called by code executing on a hardware interrupt level.

The returned module ID is set to zero when the return code is nonzero.

Subroutine Call:

```
Return Code = _querym (IOCN, module ID);
```

Calling Register Conventions:

GPR2 = IOCN
GPR3 = Address of the returned module ID. The module ID is a word value that is word-aligned in memory.

Return Codes: contained in GPR2

0 = Successful completion
16 = Invalid IOCN specified.

Virtual Machine Control Procedures

The VRM provides two services for virtual machine control. They are:

- Query virtual machine
- Terminate virtual machine (Signal termination)

Each function is described in the following section.

Query Virtual Machine (_queryv)

Description: The query virtual machine function allows a process to determine:

- If a specific virtual machine exists in the system
- All existing virtual machines in the system.

_queryv returns this information in a structure. Before you call **queryv**, set the number-of-elements field to the maximum number of elements the structure can contain. See Figure 5-12.

Subroutine Call:

```
Return Code = _queryv (type, return info.,
                       length of info., query ID);
```

Calling Register Conventions:

GPR2 = Query type
  Bit 0 = 1 - Query by VMID
  Bit 1 = 1 - Query by name
  Bit 2 = 1 - Query all virtual machines.
GPR3 = Word-aligned address of the query structure
GPR4 = Length of the query structure
GPR5 = Virtual machine ID or virtual machine name.

The query structure has the following format.

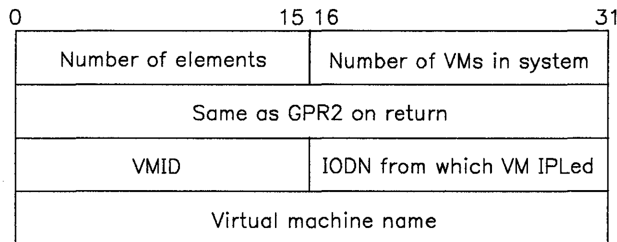


Figure 5-12. Structure Returned from _queryv

The third and fourth words in the structure are repeated for each element specified in bits 0-15 of the first word (number of elements).

Return codes: contained in GPR2

- 0 = Successful.
- 4 = No virtual machines satisfy this request.
- 16 = Invalid query option (GPR2 bits 0-7) or specified VMID is invalid.
- 24 = The supplied structure is too short to satisfy the request.

Comments: If the supplied structure is too short to contain all the virtual machines (return code = 24), the number-of-entries area in the structure tells you how many virtual machines actually exist. After you increase the structure size to allow for that number of virtual machines, you should re-issue the call.

Terminate Virtual Machine

Because a virtual machine is a process, a process terminates a virtual machine in the same way a process terminates other processes. A VRM process can terminate a virtual machine by signal termination. See "Signal (_signal)" on page 5-14 for a description of how signal termination works.

Input/Output Procedures

The VRM has several mechanisms to monitor and control input/output requests. The VRM input/output procedures include:

- Allocate a buffer from the buffer pool
- Free an allocated buffer
- Read block
- Read words
- Receive data from bus memory
- Ring queue create
- Ring queue delete
- Ring queue get word
- Ring queue put word
- Start direct memory access (DMA) transfer
- Write block
- Write words.

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Allocate a Buffer from the Buffer Pool (_bfget)

Description: This routine allocates a buffer from a block I/O device driver's buffer pool. This function accepts the 32-bit effective address of the buffer pool and returns the 32-bit effective address of a buffer from the pool. The segment and segment offset of the buffer pool are returned when the block I/O device is returned. For more information on the buffer pool and block I/O devices, see *VRM Device Support.*

Subroutine Call:

```
Return Code = _bfget (buffer pool effective address);
```

Calling Register Conventions:

GPR2 = Effective address of the buffer pool

Return Codes: contained in GPR2

-1 = Insufficient resources
¬ -1 = Address of the allocated buffer

Free Allocated Buffer (_bffree)

Description: This routine frees a buffer allocated with **_bfget**. For more information on buffer pools and block I/O devices, see *VRM Device Support.*

Subroutine Call:

```
Return Code = _bffree (effective address of buffer to free)
```

Calling Register Conventions:

GPR2 = Effective address of the buffer to free

Return Codes: contained in GPR2

0 = Successful completion

Read Block (_rdblk)

Description: This service reads 512 bytes of data from a 16-bit input/output device at the maximum possible data rate. The real address of the system memory buffer must be word-aligned, and the buffer must be contiguous in real memory. The I/O address is not increased between reads.

Subroutine Call:

```
_rdblk (I/O address, memory address);
```

Calling Register Conventions:

GPR2 = I/O register address
GPR3 = Real memory address.

Return Codes: none.

Read Words (_rdwds)

Description: This service reads words of data from a 16-bit input/output device at the maximum possible data rate. The real address of the system memory buffer must be word-aligned, and the buffer must be contiguous in real memory. The I/O address is not increased between reads.

The data length must be an integer number of words (the two least significant bits are set to zero).

Subroutine Call:

```
_rdwds (I/O address, memory address, data length);
```

Calling Register Conventions:

GPR2 = I/O register address
GPR3 = Real memory address.
GPR4 = Data length (in bytes).

Return Codes: none.

Receive Data from Bus Memory (_erecv)

Description: This routine receives data from the I/O bus into a buffer that is aligned on a halfword or fullword boundary.

Subroutine Call:

```
_erecv (target addr., target length, source addr., source length);
```

Calling Register Conventions:

GPR2 = Target address (must be halfword- or fullword-aligned)
GPR3 = Target length
GPR4 = Source address (must be fullword-aligned)
GPR5 = Source length

Return Codes: none.

Ring Queue Create (_rqc)

Description: This routine creates a ring queue from VRM heap storage space. The ring queue is aligned on a fullword boundary and each entry is a fullword. Figure 5-13 shows a VRM ring queue.

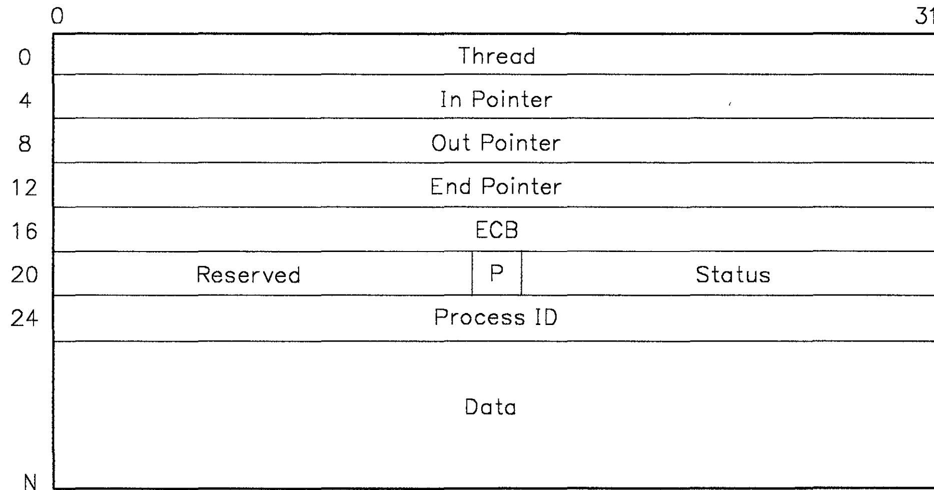


Figure 5-13. VRM Ring Queue

The fields in the preceding figure are defined as follows:

Thread	This field points to the next queue in the ring or back to the beginning of the ring.
In Pointer	This field points to the next address in the data area to put a fullword.
Out Pointer	This field points to the next address in the data area from which to get a fullword.
End Pointer	This field points to the first byte outside the data area. It is at this location that additional data is placed in the queue.
ECB	This field contains the ECB mask to be used with VRM processes or the virtual machine interrupt level (bits 21-23) and sublevel (bits 24-28) for virtual machine processes.
P	Bit 15 of byte offset 25 indicates whether the process that gets from the ring queue is a virtual machine process (bit set to 1) or a VRM process (bit set to 0).

Status — This field indicates the status of the queue on the ring. A value of 0x5555 indicates good status on the ring; a value of 0x0000 indicates the queue is no longer on the ring.

PID — This field contains the process ID of the process that gets data from the queue.

Data — This variable-length field contains data in fullword increments or pointers to blocks of data.

Subroutine Call:

```
_rqc (length, ecb, pid);
```

Calling Register Conventions:

GPR2 = Length. Indicates the number of fullwords for the queue.
GPR3 = ECB. Indicates the ECB mask for VRM processes or the interrupt level (bits 21-23) and sublevel (bits 24-28) for virtual machine processes.
GPR4 = PID. Contains the ID of the process that will get data from the queue.

Return Codes: contained in GPR2

-1 = Insufficient resources.

16 = Invalid level or sublevel specified.

A return code other than -1 or 16 is the address of the newly-created ring queue.

Ring Queue Delete (_rqd)

Description: This routine deletes a ring queue that was created with **_rqc**.

Subroutine Call:

```
_rqd (rqpointer);
```

Calling Register Conventions:

GPR2 = Contains a pointer to the ring queue to be deleted.

Return Codes: none.

Ring Queue Get Word (_rqgetw)

Description: This routine gets the next word of data from the specified queue.

Subroutine Call:

```
_rqgetw (rqpointer);
```

Calling Register Conventions:

GPR2 = Contains a pointer to the ring queue from which to obtain a fullword of data. The data pointed to by the out pointer field in the queue will be sent.

Return Codes: contained in GPR2

-1 = Ring queue full.

-2 = Ring queue pointer invalid.

A return code other than -1 or -2 is the word of data from the queue.

Ring Queue Put Word (_rqputw)

Description: This routine puts a word of data in the specified ring queue.

Subroutine Call:

```
_rqputw (rqpointer, data);
```

Calling Register Conventions:

GPR2 = Contains the address of the ring queue. The address contained in the pointer field of the queue is where the data is placed.
GPR3 = Contains the data to be placed in the queue. This value cannot be -1 or -2.

Return Codes: contained in GPR2

0 = Successful.

-1 = Ring queue full.

-2 = Ring queue pointer invalid.

Start Direct Memory Access Transfer (_stdma)

Description: The RT PC system recognizes two types of DMA devices, system and alternate. A system DMA device resides on the planar and an alternate DMA device resides on an adapter.

For alternate DMA devices, the Start DMA Transfer service initializes the channel's translation control words (TCW) for a DMA transfer. The starting bus address to be used for the transfer is returned if the function is successful. This starting bus address, not the input address, should be passed to the DMA controller.

The next bit in the type parameter is provided to support scatter/gather operations. The first call specifies next as zero and all subsequent calls for that transfer specify next as one.

Subsequent calls must supply the bus address as input to this function. The bus address must be set to the returned bus address plus the length of the previous transfer. Because the system controller does not support scatter/gather operations, the next bit should be set only for alternate DMA devices that do provide scatter/gather capability.

Alternate bus masters can arbitrate more than one device channel into a single bus DMA channel. When this happens, more than one window into system storage is required. The window number in the type parameter supports this function. Each **_stdma** command specifies the unique window number for the device. Because each device has its own window into storage, more than one device can be active at a time. Window size is determined by the number-of-windows parameter specified in the hardware characteristics section of the define device structure.

The number-of-windows and window size values are defined as follows:

$$\text{Number of windows} = 2^{\text{number of windows field of DDS}}$$

$$\text{Window size} = 2^{17 - \text{number-of-windows field of DDS}}$$

The maximum transfer size is less than or equal to the window size. The number of windows in the DDS can be 0, 1, 2, 3, or 4.

This window function is only useful for alternate DMA devices.

A system DMA device resides on the system planar. Two DMA controllers on the planar support system DMA transfers. The first controller supports 8-bit transfers with a 16-bit address for channels 0 through 3. The second controller supports 16-bit transfers with a 17-bit address for channels 5 through 7. For channels 0 through 3, the starting address and length do not have to be boundary-aligned and have a one-byte granularity. For channels 5 through 7, the starting address and length have to be even-byte aligned and have a two-byte granularity. The Start DMA Transfer

service initializes the channel's translation control words and controller for a DMA transfer.

A bus address in the address type field maps the transfer using the effective address space of the coprocessor, not the 32-bit main processor. This option is useful when performing alternate transfers for the coprocessor or when emulating a system device for the coprocessor.

Subroutine Call:

```
Return Code = _stdma
   (type, seg.ID, addr., length, bus addr.);
```

Calling Register Conventions:

GPR2 = Type of DMA operation

This word is defined in Figure 5-14.

GPR3 = Segment ID if address type is segment offset
GPR4 = Transfer address
GPR5 = Length (in bytes)
(0(R1)) = Address of the returned bus address. This address is a word value that is word-aligned in memory. (If this value is -1, no bus address is returned.)

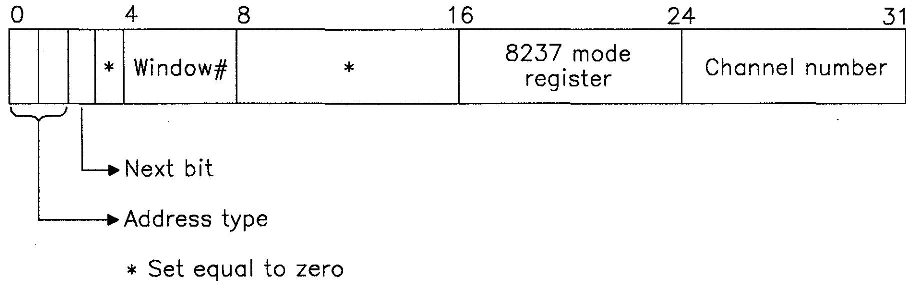


Figure 5-14. DMA Type Field

The fields in the preceding figure are defined on the following page.

- Address type — identifies the type of address in the transfer address (from GPR4). Possible values include:
  - 00 = segment offset, GPR3 contains the segment identifier
  - 01 = real address
  - 10 = coprocessor effective address
  - 11 = segment offset, GPR3 contains the segment identifier
- Next bit — indicates scatter/gather support.
- Window# — indicates the device's unique window into memory.
- 8237 mode register — this byte contains the value of the 8237 DMA controller mode register. See *IBM RT PC Hardware Technical Reference* for valid bit settings for this register.
- Channel number — indicates the DMA channel number.

Return Codes: contained in GPR2

-1 = Insufficient resources (TCWs)
0 = Successful
16 = Invalid address or length.

Comments: The VRM supplies the 8237 channel number in 8237 mode, since the 8237 channel number may be different from the DMA channel number. The 8237 does not support scatter/gather modes.

The VRM enables a device's DMA channel at attach time unless otherwise specified in the device's DDS (a bit in the DMA type field of the DDS may be set to indicate do not enable). If the bit is set and a DMA operation is directed to the device, the results are unpredictable because the device is not yet ready for DMA activity.

System DMA devices must support 24-bit addresses if they are to perform transfers to system memory.

This routine should be used only by DMA controllers running in page mode. TCWs for region-mode DMA controllers are initialized using the **_mapsys** function.

The maximum transfer length for any one operation is less than or equal to 128K bytes (64 TCWs per channel).

Termination:

This service abends the system if one or more of the following parameters specified are incorrect:

- Invalid channel number
- Transfer area not in storage.

Write Block (_wrblk)

Description: This service writes 512 bytes of data to a 16-bit input/output device at the maximum possible data rate. The real address of the system memory buffer must be word-aligned, and the buffer must be contiguous in real memory. The I/O address is not increased between reads.

Subroutine Call:

```
_wrblk (I/O address, memory address);
```

Calling Register Conventions:

GPR2 = I/O register address
GPR3 = Real memory address.

Return Codes: none.

Write Words (_wrwds)

Description: This service writes words of data to a 16-bit input/output device at the maximum possible data rate. The real address of the system memory buffer must be word-aligned, and the buffer must be contiguous in real memory. The I/O address is not increased between reads.

Subroutine Call:

```
_wrwds (I/O address, memory address, data length);
```

Calling Register Conventions:

GPR2 = I/O register address
GPR3 = Real memory address.
GPR4 = Data length (in bytes).

The data length must be specified as an integer number of words (the two least significant bits must be set to zero).

Return Codes: none.

Minidisk Management

The VRM contains several mechanisms to control minidisks. The minidisk management runtime routines include:

- Check for bad blocks
- Minidisk bad blocks
- Minidisk manager check.

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Check for Bad Blocks (_chkblk)

Description: The fixed disk device driver calls this routine at the start of I/O processing when bit 5 of the queue element's operation options field equals one. If this routine detects a bad block for an I/O transfer, then **_chkblk** sets bit 5 of the queue element's operation options field to one. This routine must be called until bit 5 equals zero (no more bad blocks detected). If this routine detects no bad blocks (bit 5 equals zero), the fixed disk device driver processes the entire I/O request.

The first time you use this routine, set the "initial call" flag. If **_chkblk** detects no bad blocks, the entire I/O request is processed. When the VRM detects bad blocks, this routine uses the queue element as a work area to update the queue element physical sector number (offset 24) after each call. Then **_chkblk** calculates the location of the next bad block and informs the caller of the number of blocks to transfer, starting at the queue element's physical sector number. This value, the number of bad blocks, is the intermediate transfer length. When the intermediate I/O transfer is complete, the fixed disk device driver must re-issue this call to determine the length of the next data transfer. This process continues until the bad blocks flag is reset, indicating no more blocks in the transfer.

Note: The **_chkblk** routine is part of the minidisk manager I/O routines. This routine **must** communicate with any fixed disk device driver you install.

Subroutine Call:

```
Return Code = _chkblk (QE, # of blocks, flag, disk IODN);
```

Calling Register Conventions

GPR2 = Word-aligned queue element address
GPR3 = Address of the returned number of blocks to transfer if a bad block is detected. This number of blocks to transfer is a word value that is word-aligned in memory.
GPR4 = Flags
- 0 = Initial call
- 1 = Subsequent call.

GPR5 = IODN of the fixed disk making the I/O request.

Return Codes: contained in GPR2

0 = Successful completion.
-40 = Unable to perform I/O to this sector. This error return code occurs in the following situation: When a read of a bad block is attempted, this service automatically attempts to remap the bad block. If no write operation is done to the remapped block before another read is attempted, the system is unable to perform the requested I/O and you get a -40 return code.

Minidisk Bad Blocks (_badblk)

Description: This minidisk manager function needs to be called if a bad block is detected by the fixed-disk device driver during an I/O operation after the driver has attempted the prescribed number of retries. The device driver must retry writing the block of data to the new physical location and then resume data transfer from the point at which the bad block caused the interruption. This routine updates the minidisk directory with the appropriate information and returns a relocated block number to the caller. The fixed-disk device driver depends on this routine to determine if the block is to be relocated. Depending on the minidisk characteristics, some write operations will be relocated and some will not. Thus, the device driver must check the return code to determine if the block should be retried at a new location upon completion of this routine.

Note: The **Minidisk Bad Blocks** routine is part of the minidisk manager I/O routines. This routine **must** communicate with any fixed-disk device driver you install.

Subroutine Call:

```
Return Code = _badblk
     (disk IODN, block number, I/O option, minidisk IODN);
```

Calling Register Conventions:

GPR2 = IODN of the fixed disk reporting the error
GPR3 = Bad block number
GPR4 = Input/output option.
    0 = read
    1 = write
GPR5 = IODN of the minidisk

Return Codes: contained in GPR2

All return codes other than 0 or -1 indicate the relocated sector number.

0 = Successful completion or no relocation required
-1 = Insufficient space to relocate the block number.

Minidisk Manager Check (_mdmchk)

Description: This minidisk manager service translates a queue element's logical minidisk block number into a physical disk sector number. This routine examines all I/O request queue elements for the fixed-disk device driver. If the request is for a minidisk, this routine translates the minidisk's logical block number to the fixed disk physical block number. The VRM updates this value in the queue element (byte offset 24). Therefore, the acknowledge queue element reflects the actual fixed disk sector number.

The VRM performs range checking for both minidisk and fixed disk I/O requests (read and write only) to ensure that the transfer length does not exceed the length of the minidisk or fixed disk (whichever is applicable). The transfer length is passed in byte offset 20 of the queue element. The IODN of the fixed-disk device driver is returned in byte offset 30 of the queue element.

This routine also looks ahead for bad blocks. If a bad block is detected for a particular I/O transfer, **_mdmchk** sets the bad block flag (bit 5 of the queue element options field). This bit should equal zero on input to this call.

For minidisk requests, the access rights of the caller are checked by this routine.

The **Minidisk Manager Check** routine should be called by the device driver's check parameter subroutine so the VRM can process the queue element before the device driver executes the request.

Note: The **Minidisk Manager Check** routine is part of the minidisk manager I/O routines. This routine **must** communicate with any fixed disk device driver you install.

Subroutine Call:

```
Return Code = _mdmchk (queue element type, address);
```

Calling Register Conventions:

GPR2 = Queue element type (22 or 23)
GPR3 = Contains the word-aligned address of the queue element.

Return Codes: contained in GPR2

0 = Successful completion
16 = Invalid IODN
22 = Invalid option
276 = Invalid block number
280 = Invalid access rights.

Device Management

The VRM contains several mechanisms to control devices. The device management runtime routines include:

- Allocate device-dependent data
- Assign a device to the coprocessor
- Change bus mask
- Define device
- Machine identification
- Map system memory
- Query device identifier
- Query device status
- Set up region mode.

Each function is described on the following pages. The description includes the format of the subroutine call, the calling register conventions, and the possible return codes.

Allocate Device-Dependent Data (_dalct)

Description: This routine allocates memory for device-dependent data. The specified device's DDS is copied into the start of the memory allocated for the device's data. The address of the data (or -1 for insufficient storage) is returned in GPR2. The allocate service can be called only by a devices driver's define device subroutine. It cannot be called by code executing on a hardware interrupt level.

Subroutine Call:

```
Return Code = _dalct (DDS addr., DDS length, data length);
```

Calling Register Conventions:

GPR2 = Word-aligned address of the device's DDS
GPR3 = Length in bytes of the device's DDS
GPR4 = Length in bytes of the device's data.

Return Codes: See 'Description' above.

Assign a Device to the Coprocessor (_assign)

Description: You can assign a device to the coprocessor only when the device is inactive and is specified as switchable. See "Define Device SVC" on page 4-51 to determine when a device is switchable. This service allocates a device and its associated resources to the coprocessor. The device is freed from coprocessor control when this service is issued with the release option specified.

This service cannot be called by code executing on a hardware interrupt level.

Subroutine Call:

```
Return Code = _assign (option, DID);
```

Calling Register Conventions:

GPR2 = Option (in binary)
11 = assign
00 = release.
GPR3 = 32-bit device ID

Return Codes: contained in GPR2

-1 = Device or its resources in use
-1 = Invalid hardware characteristics
-1 = Device is not a device driver
0 = Successful completion
4 = Invalid DID specified.

Comments: This service abends if you try to release a device not allocated to the coprocessor.

Change Bus Mask (_chgmsk)

Description: The change bus mask function allows components to enable or disable a bus interrupt level or a DMA channel. Valid bus interrupt levels are 0 through 15. Valid DMA channels are 1 through 3 and 5 through 8.

Subroutine Call:

```
_chgmsk (command, number);
```

Calling Register Conventions:

GPR2 = Command
- 0 = Disable bus interrupt
- 1 = Enable bus interrupt
- 2 = Disable DMA channel
- 3 = Enable DMA channel.

GPR3 = Interrupt level or DMA channel number.

Return Codes: None.

Comments: This service abends if you incorrectly specify one or more of the following parameters:

- Interrupts disabled on input
- Invalid command
- Invalid interrupt level
- Invalid channel number.

The change bus mask function keeps a count associated with each interrupt level and DMA channel. When you enable an interrupt or channel, you increase the count. When you disable an interrupt or channel, you decrease the count. A count that increases from 0 to 1 enables the interrupt or channel. A count that decreases from 1 to 0 disables the interrupt or channel.

Programmer's Note:

Note that the 'disable bus interrupt' option of this command may not work at all if the bus interrupt level you want to disable is shared by one or more devices.

If you need to disable an interrupt level for a relatively short period of time (such as 10 microseconds or less), it is more efficient to disable all interrupt levels (by setting bit 23 of the hardware ICS register) than to use the disable option of **_chgmsk**. Note that this technique disables all interrupts and should be used sparingly and for short time periods only.

Define Device (_defind)

Description: The define device service creates or deletes a device directory entry for the specified device. This device directory entry must exist before a virtual machine or internal process can query or activate a device. The IODN in the define device structure can be either input or returned. You must supply the IODN for a device driver, device manager, minidisk, or other static device. In the case of a virtual device or other dynamic device, the VRM returns an IODN. When you specify an IODN of zero in the DDS, the VRM assumes it is a virtual or dynamic device and replaces it with the IODN it assigns to the device.

Subroutine Call:

```
Return Code = _defind (option, DDS seg.ID, DDS eff. addr,
                       QID, MID, DID);
```

Calling Register Conventions:

GPR2 = Option
- 1 = add
- 0 = delete.

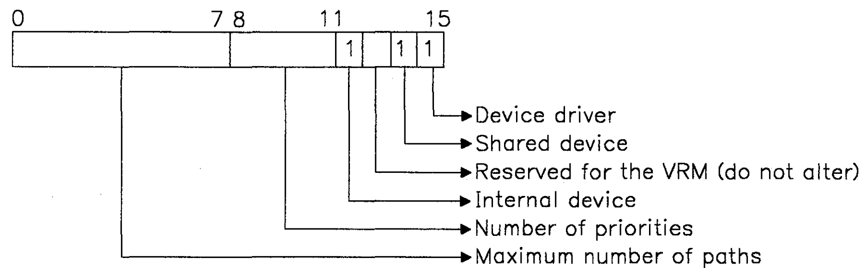
GPR3 = Segment ID of device's DDS
GPR4 = Word-aligned address of device's DDS
GPR5 = Queue ID of device
(0(R1)) = Module ID of device
(4(R1)) = Address of the returned device ID. The device ID is a word value that is word-aligned in memory. (If this value is -1, no device ID is returned.)

Return Codes: contained in GPR2

- 0 = Successful completion
- 4 = Invalid IODN specified
- 12 = Invalid DDS specified
- 16 = Invalid MID or QID specified
- 20 = Not deleted, device is in use.

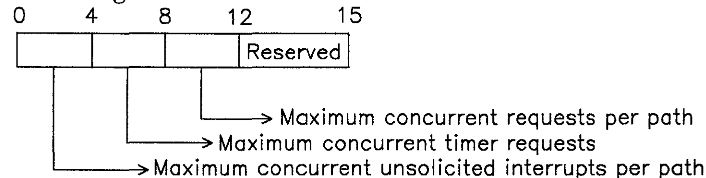
Comments: Device drivers must be specified with a queue ID of 0. All other devices must specify a valid queue ID.

The device type halfword in the define device structure is extended with the following fields:



A virtual machine cannot attach to a device that is specified as internal. In the preceding figure, the number of priorities field and the maximum number of paths field are used to create the queue for the device driver.

The define options halfword in the define device structure is extended with the following fields:



The default values are zero for the maximum number of concurrent unsolicited interrupts per path, the maximum number of concurrent timer requests, and the maximum concurrent requests per path.

This service does not verify that the IOCN or device options in the DDS are consistent with the MID or option parameters.

The define device service may not be called by code executing on a hardware interrupt level.

Also, this service alters the contents of segment register 13.

Machine Identification (_uname)

Description: The machine identification routine returns the unique serial number and model type of an RT PC.

Subroutine Call:

```
Return Code = _uname (uname_area, area_length);
```

Calling Register Conventions:

GPR2 = Address of a word-aligned data area at least 8 bytes long. Bytes 0-4 of the area contain the unique machine ID. Bytes 5-8 of the area indicate the model type. A zero in bytes 5-8 indicates a model 20 or 25; a one indicates a model 10.

GPR3 = Length of the area. The length of the area must be at least 8 bytes; if the area is more than 8 bytes, only 8 bytes are used.

Return Codes: contained in GPR2

0 = Successful.
4 = Successful, but unique ID may have been tampered with.
16 = Data area not word-aligned or less than 8 bytes in length.

Map System Memory (_mapsys)

Description: The map system memory function serves two functions. It allocates a portion of system memory for use by the coprocessor (the coprocessor by default has access to all bus memory) and can reserve TCWs for region-mode DMA transfers.

The **_mapsys** routine is not used to pin pages or reserve system memory.

When system memory is allocated for use by the coprocessor, pages in the mapped space are removed from the paging subsystem's pool of real pages, thus reducing the processor's working set. Use of system memory by the coprocessor reduces main processor performance because real memory space is reduced and contention for the remaining resources is increased.

Memory is allocated in 2K units and mapped in 32K units. A returned parameter provides you with the offset into system memory for the memory allocated to the coprocessor.

When you choose the "free" option, you can return to the main processor memory once allocated to the coprocessor and can unmap the TCWs.

Subroutine Call:

```
Return Code = _mapsys (type, MID, bus addr., length, eff.addr);
```

Calling Register Conventions:

GPR2 = Option
- 0 = free
- 1 = free and unmap
- 2 = allocate
- 3 = allocate and map.

GPR3 = Coprocessor module ID (or ignored)
GPR4 = Bus address
GPR5 = Length in bytes
(0(R1)) = Address of the returned offset into **_sysmem**. The returned offset is a word value that is word-aligned in memory. (If this value is -1, no offset is returned.)

Return Codes: contained in GPR2

- -1 = Insufficient real memory
- 0 = Successful.

Comments: System memory can be allocated to the coprocessor only if the coprocessor MID is specified correctly in the **_mapsys** call. If the MID is not correct, options 0 (free) and 2 (allocate) are no-operation routines. For options 1 and 3, the mapping or unmapping of region-mode TCWs will occur if the coprocessor MID is incorrectly specified, but

the free and allocate portions of those options must have the MID specified correctly to be successful.

This routine may be issued more than once to map discontinuous areas of memory. The map system memory service may not be called by code executing on a hardware interrupt level.

Termination:

This service abends the system if one or more of the following parameters are incorrect:

- Invalid operation type
- Invalid bus address or length.

Query Device Identifier (_queryd)

Description: You can determine the VRM device identifier associated with a specified IODN with the query device ID routine.

This service cannot be called by code executing on a hardware interrupt level.

Subroutine Call:

```
Return Code = _queryd (IODN, device ID);
```

Calling Register Conventions:

GPR2 = IODN
GPR3 = Address of the returned device ID. The device ID is a word value that is word-aligned in memory.

Return Codes: contained in GPR2

0 = Successful completion
16 = Invalid IODN specified.

Comments: If this operation is unsuccessful (return code does not equal zero), the VRM sets the value for the returned device ID to zero.

Query Device Status (_qryds)

Description: The query device status function is used to obtain information about a specified device or I/O operations associated with that device.

The query device status service may not be called by code executing on a hardware interrupt level. Also, this service alters the contents of segment register 13.

For more information on query services, see "Query Device SVC" on page 4-58.

Subroutine Call:

```
Return Code = _qryds (device ID, query information);
```

Calling Register Conventions:

GPR2 = Device ID
GPR3 = Word-aligned address of the query device structure
GPR4 = Length in bytes of the query device structure.

Note that GPR4, rather than the length field of the QDS, indicates the length of the QDS.

The fields within the query device structure are defined as follows:

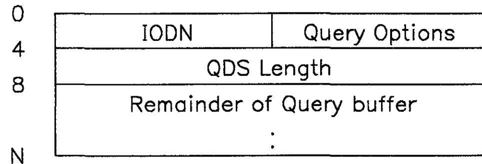


Figure 5-15. Query Device Structure

Return Codes: contained in GPR2

0 = Successful completion
12 = QDS not on a full word boundary or invalid address
16 = Invalid DID/IODN specified
20 = QDS not long enough.

Set up region mode (_dmamov)

Description: The **_dmamov** function is used by region-mode DMA devices to move data between a user buffer and an address in segment register 14 (the range 0xE0000000 through 0xE0FFFFFF).

The VRM reserves segment register 14 for translating region-mode memory references from the I/O bus. All effective addresses generated by the planar are in the range 0xE0000000 through 0xE0FFFFFF. The memory management unit uses segment register 14 to translate this effective address first into a virtual address and then into a real address.

At system initialization, the VRM initializes all region-mode TCWs to disallow system memory access. The planar will not respond to any region-mode address generated on the bus. The VRM then creates a segment and places the segment ID into segment register 14. This segment is unique in that no memory management SVC can operate on it, and page faults within it result in exceptions. No page frames are assigned to this segment until the **_mapsys** or **_dmamov** functions have been called.

The **_mapsys** function is used by the coprocessor to take page frames from the main processor and assign them to this segment. It is also used to map TCWs for region-mode DMA use.

The **_dmamov** function is used by alternate DMA controllers (other than the coprocessor) to move page frames into segment 14.

The data moved into segment register 14 is pinned. The data is unpinned when it is moved out of segment register 14. If the source address is in segment register 14 and the segment ID is set to -1, the **_dmamov** function simply unpins and frees the pages in the range.

Subroutine Call:

```
Return Code = _dmamov (seg_ID, source, target, length);
```

Calling Register Conventions:

GPR2 = Segment ID of user buffer
GPR3 = Address of source data
GPR4 = Address of target
GPR5 = Length in bytes of data to move.

Return Codes: contained in GPR2

-1 = Insufficient real memory
0 = Successful completion
8 = Invalid segment ID
12 = Invalid page in the page range

Comments: When moving data from segment register 14, any pages in the target range are freed. If an error occurs during a move from segment register 14, all remaining data in the range of the request will be removed from segment register 14 and the pages will be lost.

The caller should have already allocated region-mode TCWs with the **_mapsys** function before using **_dmamov**.

This service cannot be used successfully by code executing on a hardware interrupt level.

VRM Trace and Error Process Interfaces

The virtual machine uses the **Send Command SVC** and queue elements to communicate with the VRM error and trace processes.

A virtual machine must first establish a path to the VRM error and trace processes with **Attach Device SVC**. When this path is established, two asynchronous operations may be used to control the trace and error information interface. They are:

- **errrecvr** on or off

  This command specifies the receiver of VRM error entries. VRM error entries are sent to the virtual machine (after a path is set up) as virtual interrupts one at a time, as they occur.

- **trace** on or off

  This command determines whether the VRM trace facilities are active. With **traceon**, VRM trace entries are placed in a buffer. When the buffer becomes full, a pointer to the buffer and the buffer size are sent to the virtual machine as virtual interrupts.

Error Process (errrecvr on/off)

Two operations control whether error entries are sent to the virtual machine. One of the operations establishes a path from the VRM error process to the virtual machine, and the other operation erases the path. The contents of the queue elements associated with this device option are defined on the following pages.

The error process can also receive a **_post** call from another VRM component such as a device driver. The **_post** tells the error process that an event was recorded and that the virtual machine needs to be informed. When the error process receives a **_post**, it sends an unsolicited interrupt to the AIX Operating System device driver **/dev/error**.

Error Process Input Queue Element

The **Send Command SVC** queue element sent to the error process is defined in Figure 5-16 on page 5-120.

Offset			
0	Reserved		
4	Path ID		
8	Type	Priority	Options
12	IODN		Command Extension Segment ID
16	Node Name (from GPR3)		
20	Node Name (from GPR4)		
24	Buffer Address (from GPR5)		
28	Buffer Length (from GPR6)		
32			

Figure 5-16. Send Command Queue Element for errrecvr

The significant fields are defined below:

Type = 1 (Send Command queue element)

IODN = 0x50

Options = This halfword includes both operation and device options.

The low-order 5 bits of this field are the operation options. The binary value of this field is 10010. The high-order three bits are reserved.

The second byte contains the device options. Valid values for this field are:

- 0x08 — Establish path
- 0x11 — Erase established path.

Node name = This is a two-word field obtained from GPR3 and GPR4 that is only used with device option 8.

Buffer address =
This field indicates the memory location of the buffer and is only used with device option 8.

Buffer length =
This field contains the length of the buffer in bytes and is used only with device option 8.

Error Process Acknowledgement Queue Element

Figure 5-17 on page 5-121 shows the acknowledgement queue element resulting from a **Send Command SVC** call to the error process.

Offset				
0	Reserved			
4	Path ID			
8	Type	Reserved	Flags	Overrun
12	Operation Results		IODN	
16	Options		Command Extension Segment ID	
20	Command Extension Address			
24	IPL Data			
28	Reserved			
32				

Figure 5-17. Acknowledgement Queue Element for errrecvr

The significant fields are defined below:

Type = 0 (acknowledgement queue element)

Status flags = The binary value of this field equals 00010100, indicating a solicited interrupt.

Overrun count = 0

Operation results = Any of the following values:

0 = Successful
4 = Failed — path already established
8 = Failed — buffer length too small
12 = Unrecognized request
16 = Unable to pin buffer.

IPL data = This 32-bit field contains the following information for the establish error path request:

- Byte 1 — state
- Byte 2 — machine type
- Byte 3 — IPL device address
- Byte 4 — NVRAM validity.

The remaining fields are filled in by the VRM **_deque** routine from the input queue element.

Error Entry Acknowledgement Queue Element

Each time the error process receives an error entry notification, an acknowledgement queue element is sent to the AIX Operating System device driver **/dev/error** as an unsolicited interrupt. This queue element is shown in Figure 5-18 on page 5-122.

Offset				
0	Reserved			
4	Path ID			
8	Type	Operation Option	Flags	Overrun
12	Operation Results		IODN	
16	Options		Command Extension Segment ID	
20	Command Extension Address			
24	Device Dependent			
28	Reserved			
32				

Figure 5-18. Error Entry Acknowledgement Queue Element for errrecvr

The following fields are filled in by the error process:

Path ID = Indicates the path from the operating system file **/dev/error** to the VRM error process

Type = 0 (acknowledgement queue element)

Operation Option = The first five bits of this byte indicate the operation option. The binary value of these bits is 10010.

Flags = Set in the following bit pattern: 00010000 (unsolicited interrupt)

Overrun Count = 0

Operation Results = 0

IODN = 0x50

Options = Includes both operation and device options. The first five bits of this field (operation option) are set to a binary 10010.

The second byte of this field (device option) is set to 0x10.

Command Extension Segment ID = This field contains the ID of the segment in which to find the address of the VRM error entry.

Command Extension Address = This 32-bit value provides a segment ID and segment displacement. The first four bits indicate the segment ID and the next 28 bits specify a displacement into the segment where the VRM error entry may be found.

Device Dependent = Length of the VRM error entry.

Trace Process

The virtual machine uses the **Send Command SVC** with two device options to either turn tracing on or off in the VRM. The contents of the queue elements associated with these operations are defined on the following pages.

The trace process can also receive a **_post** from another VRM component, such as a device driver. The **_post** tells the trace process that a trace buffer is full and that the virtual machine needs to be informed. When the trace process receives one of these **_post** calls, it sends an unsolicited interrupt to the AIX Operating System device driver **/dev/trace**.

Several external variables contain information about trace process status. See "External Variables" on page D-21 for a definition of these variables.

Trace Process Input Queue Element

The **Send Command SVC** queue element sent to the trace process to turn VRM trace on is defined in Figure 5-19.

Offset	Fields
0	Reserved
4	Path ID
8	Type | Priority | Options
12	IODN | Command Extension Segment ID
16	Channels to Enable (GPR3)
20	Entry Size | VM ID | P | Channel Index
24	Buffer Address (from GPR5)
28	Buffer Length (from GPR6)
32	

Figure 5-19. Send Command Queue Element for trace

The significant fields are defined below:

IODN = 0x51

Options = Includes both operation and device options. The high-order 5 bits of the first byte (operation options) are set to a binary 10010.

The second byte (device option) is set to 0x06.

Channels to Enable = Indicates groups of hookids to be active

Entry Size = Total number of bytes in each trace entry

VM ID = Contains the ID of the virtual machine to which trace buffer data is to be sent.

P = The P bit, when set, indicates that the trace process does not notify the virtual machine when the trace buffer is full. Instead, the process sends the last buffer to the virtual machine when trace is stopped.

Channel Index = a 7-bit value used to keep track of trace sessions.

Buffer Address = Contains the address of the full trace buffer for access by the virtual machine.

Buffer Length = Contains the length of the trace buffer in bytes. The minimum buffer length is 3K bytes.

Trace Process Acknowledgement Queue Element

Figure 5-20 shows the acknowledgement queue element resulting from a **Send Command SVC** call to the trace process.

Offset				
0	Reserved			
4	Path ID			
8	Type	Reserved	Flags	Overrun
12	Operation Results		IODN	
16	Options		Command Extension Segment ID	
20	Command Extension Address			
24	Reserved		Channel Index	
28	Reserved			
32				

Figure 5-20. Acknowledgement Queue Element for trace

The significant fields are defined below:

Type = 0 (acknowledgement queue element)

Status Flags = Set in the following bit pattern: 00010100 (unsolicited interrupt)

Overrun Count = 0

Operation Result = Can be one of the following values:

- 0 = Successful
- 4 = Failed — trace already on
- 12 = Unrecognized request

- 16 = Unable to pin buffer from virtual machine.

Channel Index = A 16-bit value that identifies the trace session for which the operation was performed.

The remaining fields are filled in by the VRM **_deque** call from the input queue element.

Trace Process Input Queue Element

The **Send Command SVC** queue element sent to the trace process to turn VRM trace off is defined in Figure 5-21.

Offset				
0	Reserved			
4	Path ID			
8	Type	Priority	Options	
12	IODN		Command Extension Segment ID	
16	Channels to Disable			
20	Not Used (from GPR4)			Channel Index
24	Not Used (from GPR5)			
28	Not Used (from GPR6)			
32				

Figure 5-21. Send Command Queue Element for trace

The significant fields are defined below:

IODN = 0x51

Options = Includes both the operation and device options. The first five bits of the first byte (operation options) are set to a binary 10000. The second byte of the options halfword (device option) is set to 0x07.

Channels to Disable = Indicates the channels to disable.

Channel Index = A 7-bit value that indicates the trace session to turn off.

Trace Process Acknowledgement Queue Element

Figure 5-22 on page 5-126 shows the acknowledgement queue element resulting from a **Send Command SVC** to the trace process.

<table>
<tr><td>0</td><td colspan="4">Reserved</td></tr>
<tr><td>4</td><td colspan="4">Path ID</td></tr>
<tr><td>8</td><td>Type</td><td>Reserved</td><td>Flags</td><td>Overrun</td></tr>
<tr><td>12</td><td colspan="2">Operation Results</td><td colspan="2">IODN</td></tr>
<tr><td>16</td><td colspan="2">Options</td><td colspan="2">Command Extension Segment ID</td></tr>
<tr><td>20</td><td colspan="4">Command Extension Address</td></tr>
<tr><td>24</td><td colspan="2">Reserved</td><td colspan="2">Channel Index</td></tr>
<tr><td>28</td><td colspan="4">Reserved</td></tr>
<tr><td>32</td><td colspan="4"></td></tr>
</table>

Figure 5-22. Acknowledgement Queue Element for trace

The following fields are filled in by the trace process:

Type = 0 (acknowledgement queue element)

Flags = The binary value of this field is 00010100 (solicited interrupt).

Operation Results = Can be one of the following values:

- 0 = Successful
- 8 = Failed — trace is already off, or the requestor trying to turn trace off is not the one that turned trace on.
- 12 = Unrecognized request
- 16 = Unable to pin buffer.

The remaining fields are filled in by the VRM **_deque** routine from the input queue element.

Trace Buffer Acknowledgement Queue Element

Figure 5-23 on page 5-127 shows the acknowledgement queue element sent to the AIX Operating System device driver by the trace process when it receives a full buffer notification. It sends the queue element as an unsolicited interrupt to the AIX Operating System **/dev/trace** device driver.

Offset				
0	Reserved			
4	Path ID			
8	Type	Operation Option	Flags	Overrun
12	Operation Results		IODN	
16	Options		Command Extension Segment ID	
20	Command Extension Address			
24	Trace Buffer Length		Generic Channel Index	
28	Reserved			
32				

Figure 5-23. Acknowledgement Queue Element for trace

The following fields are filled in by the trace process:

Path ID = Contains the ID of the path from the virtual machine to the trace process.

Type = 0 (acknowledgement queue element).

Operation Option = Set in the following bit pattern: 10010.

Flags = The binary value of this field is 00010000 (unsolicited interrupt).

Overrun Count = 0.

Operation Result = 0.

IODN = Contains the IODN of the trace process.

Operation Options = Set in the following bit pattern: 10010.

Command Extension Segment ID = Contains the ID of the segment in which the full VRM trace buffer is found.

Command Extension Address = This 32-bit value is comprised of a 4-bit segment register ID and a 28-bit displacement into that segment where the full trace buffer may be found.

Trace Buffer Length = A 16-bit value indicating the length of the VRM trace buffer.

Generic Channel Index = A 16-bit value indicating the generic channel index.

Event Monitoring

The VRM contains five subroutines that allow you to monitor or examine events in the system. They are:

- Generate Error Entry (**_errvrm**)
- Generate Generic Trace Entry (**_trcgen**)
- Generate Trace Entry (**_trcvrm**).
- Internal VRM Trace (**_vrmtrc**)
- Save/Get Dump Table Entry (**_dmptbl**).

Each subroutine is described on the following pages. The description includes the format of the subroutine call and the possible return codes, if any.

For additional information about related event-monitoring components, see *AIX Operating System Programming Tools and Interfaces.*

Generate Error Entry (_errvrm)

Description: This subroutine generates an error entry for a specific VRM device or component. Error entries can be generated for two reasons:

- An error occurred in the VRM that must be made known to the error log problem determination routine.
- An event occurred in the VRM that is significant enough to warrant an entry in the error log.

When a device driver or other component performs an **_errvrm** call, the following actions take place:

1. The VRM error collector handles the call and posts the VRM error process.
2. The VRM error process places the entry into a memory buffer and passes a pointer to the virtual machine error log device driver.
3. The virtual machine error log device driver takes the information in the memory buffer and transfers it to a buffer maintained by the error daemon.
4. The error daemon pipes the entry to the error log problem determination routine.
5. The error log problem determination routine processes the entry and pipes the resultant information about the entry back to the error daemon.
6. The error daemon appends the resultant information to the entry, adjusts the length field of the entry to account for the added information, and writes the entry out to the error log.

Once an entry is in the error log, it can be formatted using the AIX Operating System **errpt** command. The programmer is responsible for providing a suitable format template for user-defined error entries.

The information required to generate an error entry is obtained from a DDS defined for each device. Components that do not use a DDS must use a similar structure in order for this subroutine to properly create the entry.

Note: This subroutine description uses the acronym 'DDS' to refer to an actual DDS as used by device drivers or a DDS-like structure as used by non-device driver components within the VRM.

The VRM maintains two different types of error entries. For hardware errors, the error entry is structured as shown in Figure 5-24 on page 5-130.

All non-hardware error entries are structured as shown in Figure 5-25 on page 5-131.

The value in the class field determines the type of error entry generated by this subroutine. If it is set to 0x01, a hardware entry is generated. If it is set to 0x02 through 0x06, a non-hardware entry is generated.

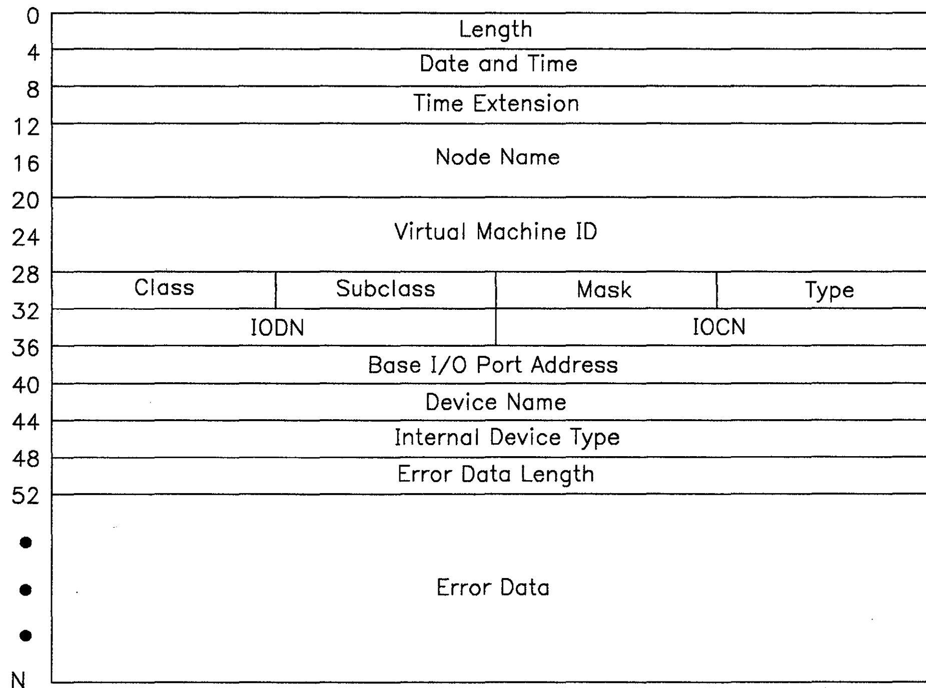


Figure 5-24. Hardware Error Entry Structure

The fields of an error entry are defined below. The fields that are specific to hardware error entries are obtained from the DDS header and hardware characteristics and are defined in "Define Device SVC" on page 4-51.

Length
: This is the length, in words, of the error entry.

Date and Time
: This value is obtained from the **_toy** variable.

Time Extension
: This value is obtained from the **_toyx** variable.

Node Name
: This value is passed to the VRM error process when it is activated by the virtual machine.

Virtual Machine ID
: This value is always blank for VRM error entries.

Class
: The class of the error that occurred. For user-defined error entries, this value is always 0x01 for hardware error entries, or 0x06 for non-hardware error entries.

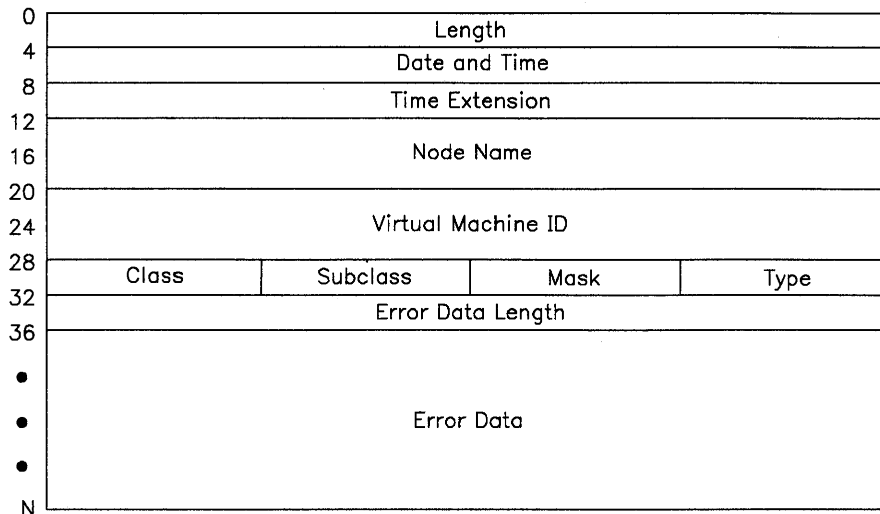


Figure 5-25. Non-Hardware Error Entry Structure

Subclass — The subclass of the error that occurred. For user-defined hardware error entries, the subclass must be 0x0F. For user-defined non-hardware error entries, the subclass can range from 0x01 to 0x0F.

Mask — The mask of the error that occurred. This can range from 0x01 to 0x0F.

Type — The type of the error entry. For user-defined error entries, this can be one of the following values:

- 0x80 — Permanent Error. The device driver could not complete the operation.
- 0x40 — Temporary Error. The device driver completed the operation successfully, but the device driver had to perform some error recovery operations.
- 0x20 — Information Entry. No error occurred, but some event occurred that was significant enough to warrant an entry in the error log.
- 0x10 — Counter Error. The number of errors exceeded the Error Ratio Threshold.

Error Data Length — The number of words used for error data in this error entry. This field is included when calculating the length.

Error Data — The error data for this error entry.

The error data is obtained from the error log section of the DDS used by the component generating the error entry. The **_errvrm** subroutine knows where to find the necessary information once it knows the location of the DDS, so all you need to provide is the address of the DDS.

Note: The error log in the DDS may contain more data than is indicated by the Error Data Length field. This additional data does not become part of the error entry. This allows the device driver or component to use part of the error log section of the DDS for purposes other than error entries.

Subroutine Call:

```
_errvrm(dds addr)
```

Calling Register Conventions:

GPR2 = The address of the DDS. Using information in the header of the DDS, the subroutine locates the DDS error log section and copies the number of words indicated by the 'Error Data Length' field into the error entry.

Return Codes: This routine does not generate return codes. Unsuccessful completion causes an abend. The possible causes for an abend are listed below:

- The return code from **_post** is non-zero.
- The length of the area passed by the user is not a multiple of full words.
- The length of the entire error entry is greater than 476 bytes.

Generate Generic Trace Entry (_trcgen)

Description: This subroutine generates generic trace entries for a specific device or component in the VRM. When a trace entry is generated, the following actions take place:

1. The VRM generic trace collector handles the call and places the trace entry into the generic trace buffer. If the entry causes the trace buffer to reach a pre-defined threshold, the VRM trace collector posts the VRM generic trace process.
2. The VRM trace process passes a pointer to the full buffer to the virtual machine trace log device driver.
3. The virtual machine trace log device driver takes the information in the trace buffer and transfers it to a buffer maintained by the trace application.
4. The trace application writes the buffer out to the generic trace log files maintained in the system.

Once a trace entry is in the trace log file, it can be formatted using the AIX Operating System **trcrpt** command. The programmer is responsible for providing a suitable format template for user-defined entries.

The length of a trace entry is specified in the AIX **trc_start** call. The default entry length is 40 bytes. The first 20 bytes constitute the header and the remaining 20 bytes are used for component or device data. Figure 5-26 shows the structure of a trace entry.

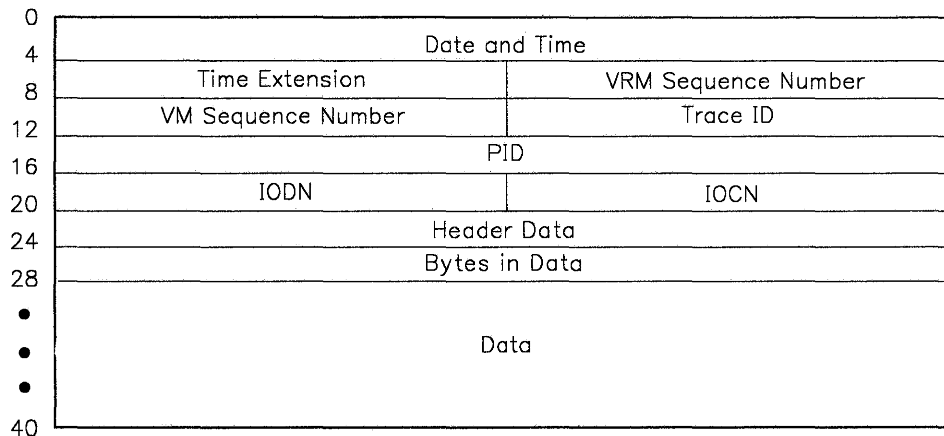


Figure 5-26. Generic Trace Entry Structure

The **_trcgen** subroutine fills in the following fields required by the trace entry header:

Date and Time:
: This value is obtained from the **_toy** variable.

Time Extension:
: This value is obtained from the **_toyx** variable.

VRM Sequence Number:
: The VRM sequence number is the value of a counter incremented by the VRM trace collector each time it receives a trace entry. This value is used to sort the sequence of VRM trace events within a time interval.

VM Sequence Number:
: The virtual machine sequence number is the value of a counter incremented by the virtual machine trace collector each time it receives a trace entry. This value is used to sort the sequence of virtual machine trace events within a time interval.

Process ID: This value is obtained from the **_curid** variable.

The rest of the information required for the trace entry is obtained from the subroutine call and from the component that performs the subroutine call.

Subroutine Call:

```
Return Code = _trcgen(buf_adr, trc_id, trc_data_hdr, trc_data)
```

Calling Register Conventions:

GPR2 = The address of the generic trace buffer.
GPR3 = The two-byte trace ID of the component generating the trace entry subroutine call, right-justified.
GPR4 = The address of a structure that contains an IODN, an IOCN, four bytes of user header data, and a four-byte value that indicates the number of data bytes in the trace data buffer.
GPR4 = The address of the trace data buffer.

Return Codes: contained in GPR2

0 = Successful completion
4 = Channel number is not on.

Generate Trace Entry (_trcvrm)

Description: This subroutine generates trace entries for a specific device or component in the VRM. When a trace entry is generated, the following actions take place:

1. The VRM trace collector handles the call and places the trace entry into the trace buffer. If the entry causes the trace buffer to reach a pre-defined threshold, the VRM trace collector posts the VRM trace process.
2. The VRM trace process passes a pointer to the full buffer to the virtual machine trace log device driver.
3. The virtual machine trace log device driver takes the information in the trace buffer and transfers it to a buffer maintained by the trace application.
4. The trace application writes the buffer out to the trace log files maintained in the system.

Once a trace entry is in the trace log file, it can be formatted using the AIX Operating System **trcrpt** command. The programmer is responsible for providing a suitable format template for user-defined entries.

Trace entries have a fixed length of 40 bytes. The first 20 bytes constitute the header and the remaining 20 bytes are used for component or device data. Figure 5-27 shows the structure of a trace entry.

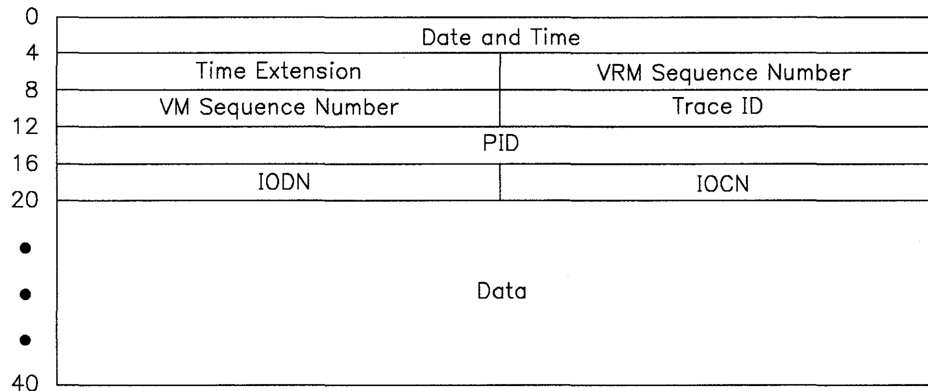


Figure 5-27. Trace Entry Structure

The **_trcvrm** subroutine fills in the following fields required by the trace entry header:

Date and Time
: This value is obtained from the **_toy** variable.

Time Extension
: This value is obtained from the **_toyx** variable.

VRM Sequence Number
: The VRM sequence number is the value of a counter incremented by the VRM trace collector each time it receives a trace entry. This value is used to sort the sequence of VRM trace events within a time interval.

VM Sequence Number
: The virtual machine sequence number is the value of a counter incremented by the virtual machine trace collector each time it receives a trace entry. This value is used to sort the sequence of virtual machine trace events within a time interval.

Process ID
: This value is obtained from the **_curid** variable.

The rest of the information required for the trace entry is obtained from the subroutine call and from the DDS of the device driver or component that performs the subroutine call.

If the trace data passed to the subroutine is in memory, then that data is placed in the trace entry. If the trace data is not in memory, the data bytes in the trace entry are set to 0xFFFF.

Subroutine Call:

```
Return Code = _trcvrm(trace ID, trace data, trace length)
```

Calling Register Conventions:

GPR2 = The two-byte trace ID of the component generating the trace entry subroutine call. The first five bits contain the channel number and the remaining eleven bits contain the hook ID number. For user-defined trace entries, the channel number is always set to 31 and the hook ID is set to a number ranging from 300 to 399.
GPR3 = Data required for the trace entry.
GPR4 = Length of the trace entry in bytes. This can range from 0 to 24.

Return Codes: contained in GPR2

0 = Successful completion
4 = Channel number is not on
8 = Entry is not a full number of words.

Internal VRM Trace (_vrmtrc)

Description: This subroutine formats trace data for predefined and user-defined trace points in the VRM. VRM trace points have already been defined for the following VRM elements:

- All first-level interrupt handlers (FLIHs)
- The VRM process dispatcher
- The system abend routine.

The VRM supports an internal trace buffer of 2K bytes that starts at real memory address 0x1000. The following locations provide more information about this buffer:

- 0x410 – start of the buffer
- 0x414 – end + 1 of the buffer
- 0x418 – current pointer into the buffer.

Note that the buffer starts on a power of 2 boundary and is of a power of 2 length.

In order to utilize this internal VRM trace, a component must run below the VMI (for example, a VRM device driver, VRM device manager, or VRM process).

Note that, unlike other tracing components, the trace data kept by **_vrmtrc** is not buffered, but remains in memory. This formatted trace data also wraps around from the end of the 32-byte trace area to the beginning.

Subroutine Call:

```
_vrmtrc(trace entry)
```

Calling Register Conventions:

GPR2 = Points to a word-aligned, 32-byte trace entry

Return Codes: none.

Comments: Figure 5-28 shows the layout of the currently-defined VRM trace points. You can define 32-byte trace point entries for installed code. The only restrictions associated with defining a trace point entry is that the first two bytes provide the trace point ID and that this ID is >0x100 (IDs ≤0x100 are reserved for the VRM). Trace point IDs not reserved by the VRM will be formatted as a 32-byte hexadecimal display.

Byte Offset	Length	Description
0	2	Trace point ID
2	2	Reserved
4	8	Program status word
12	4	Current ID
16	4	Data word 1
20	4	Data word 2
24	4	Pointer to process block address
28	4	Data word 3

Figure 5-28. Trace Entry Format

The fields in the preceding table are defined as follows:

- Trace point ID

  Identifies the trace point that is being queried. The VRM reserves values 0x000 to 0x100 for its own trace points. The following trace points are defined:

0x00 - 0x06	Used by first-level interrupt handler levels 0 through 6, respectively
0x07	Used by SVC trace points
0x0C	Used by the process dispatcher for virtual machine processes
0x0D	Used by the process dispatcher for VRM processes
0x0E	Used by machine check level
0x0F	Used by program check level
0x10	System abend

- Program status word

  This program status word contains information on the 32-bit processor's instruction address register, interrupt control status register, and condition status registers.

- Current ID

  This field is the 32-bit ID of the process, virtual machine, or interrupt handler that was running when **_vrmtrc** was called.

- Data word 1

  For a machine communications check or program status check, the low-order halfword contains the machine check status or program check status, respectively.

  For a virtual machine dispatch or SVC, this word contains the value of the virtual machine interrupt control status register.

  For a system abend, bits 8 through 15 of this word contain the component ID, bits 16 through 23 contain the machine check status, and bits 24 through 31 contain the program check status.

- Data word 2

  For an SVC, this word contains the SVC code. For a virtual machine dispatch, this word contains the real address of the virtual machine's page 0.

  For a system abend, this word contains the abend code.

- Pointer to process block address

  This word contains a pointer to the control block of the process that was most recently on execution level 7.

- Data word 3

  This word contains the virtual machine interrupt level for SVCs or virtual machine dispatches.

  For a system abend, this word contains the abend type. Abend type is defined as follows:

  1 = VRM hardware
  2 = VRM software
  3 = Virtual machine.

For more information on how the debugger handles this trace formatting, see "TRace — Display formatted trace table entries" on page 8-82.

Save/Get Dump Table Entry (_dmptbl)

Description: This subroutine manages dump table entries in the master dump table. Each entry in the master dump table identifies a single component dump table whose entries list the name, length, and location of data structures that should be included in a dump. See Figure 5-29.

Master Dump Table

CID	Len	Addr
126	44	addr
130	84	addr
120	0	addr
115	24	addr
150	84	addr

Component Dump Table

Len	Name	Len		
44	Header	12	addr	rsvd
	Counter	20	addr	rsvd

Memory

Data Structure 1

Data Structure 2

Figure 5-29. Tables Used to Identify Data Structures to be Dumped

The VRM dump program always dumps a pre-defined set of VRM data, but there may be other data that should be included in the dump. Each VRM component that

wishes to dump specific data structures must create and maintain its own component dump table. The method used to create the component dump table is not important as long as the byte structure of the table matches the byte structure expected by the dump program.

The VRM dump program always dumps the pre-defined VRM data first. It then uses the master dump table to locate the component dump tables and dump additional data structures. The VRM dump program will continue to process the component dump tables as long as there is sufficient storage space on the dump diskette.

Once the data identified by the component dump tables has been written to diskette, it can be formatted into a readable dump report. See the definition of the **dumpfmt** command in *AIX Operating System Commands Reference* for guide information. There is also a discussion of the dump components in *AIX Operating System Programming Tools and Interfaces.*

The main use of the **_dmptbl** subroutine is to create or update a master dump table entry; however, this subroutine can also perform two other operations:

- Copy the dump data associated with a master dump table entry into a specified memory buffer.
- Copy the starting address of the master dump table into a specified memory buffer.

These other functions are performed by changing the parameters used in the subroutine call. Before using the subroutine to create or update a master dump table entry, the component must build a component dump table. A component dump table is structured as a list of records, each of which is 20 bytes long. A dump table can contain a maximum of 50 entries. Attached to the beginning of the list is a single 4-byte field that specifies the overall length of the component dump table. The length can be computed using the following formula:

```
Length = 4 + (20 * Number of Entries in Component Dump Table)
```

Each entry in a component dump table has the following structure:

- Name — 8 bytes identifying the name of the data structure identified by this entry. This name will appear in the dump report.
- Length — 4 bytes identifying the length of the data structure.
- Address — 4 bytes identifying the starting address of the data structure.
- Reserved — 4 bytes reserved for the dump program.

Once a component has created a component dump table, it must place an entry in the master dump table that tells where the component dump table is located. A master dump table entry has the following structure:

- Component ID — 4 bytes identifying the component associated with the dump table entry. For user-defined component dump tables, this can be any number greater than 100. The following component IDs are reserved for the components listed below:
  - 0 — Reserved
  - 1 — VRM Nucleus
  - 2 — VRM Virtual Memory Manager
  - 3 — VRM Dump facility
  - 4 — VRM Trace facility
  - 5-19 — Reserved
  - 30 — Virtual Terminal Screen Manager
  - 31-62 — Virtual Terminal Dump Table Entries
  - 64 — Virtual Terminal Locator Device Driver
  - 65 — Virtual Terminal Keyboard Device Driver
  - 66-69 — Virtual Terminal Physical Displays
  - 70 — Virtual Terminal Resource Controller
  - 71 — Multiprotocol Device Driver
  - 75 — Block I/O Device Manager
  - 76 — Baseband Device Driver
  - 77 — 5080 Peripheral Adapter
  - 80 — Asynchronous
  - 81 — Parallel
  - 82 — Base PC Network Services
  - 83 — Streaming Tape
  - 84 — Fixed Disk
  - 85 — Diskette
  - 86 — CD ROM
  - 87 — ESDI Fixed Disk
  - 88 — SCSI Fixed Disk
  - 89 — Token ring device driver
  - 90 — PC Support of SDLC
  - 91 — DCA, 3278 Emulation
  - 92 — Distributed Function Terminal Device Driver
  - 100 — Personal Computer AT® Coprocessor Option.
- Length — 4 bytes identifying the length of the component dump table associated with this dump table entry.
- Address — 4 bytes identifying the starting address of the component dump table associated with this dump table entry.

Subroutine Call:

```
Return Code = _dmptbl(command, comp. id, length, address)
```

Calling Register Conventions: To add a dump table entry to the master dump table:

GPR2 = **save**. This has a #define value of 1.
GPR3 = The component ID. If the 'comp id' parameter matches an existing component ID in the master dump table, that entry is overlaid with the new entry data; otherwise, the entry is added to the master dump table.
GPR4 = A pointer to a 4-byte word-aligned area that contains the length of the dump table.
GPR5 = The starting address of the component dump table.
GPR6 = The length of the component dump table.

Calling Register Conventions: To retrieve data from a particular master dump table entry:

GPR2 = **getm**. This has a #define value of 2.
GPR3 = The component ID of the table entry.
GPR4 = A pointer to a 4-byte word-aligned area that contains the length of the returned information.
GPR5 = The address of the buffer to receive the table entry data.
GPR6 = The length the component dump table.

Calling Register Conventions: To retrieve the starting address of the master dump table:

GPR2 = **getm**. This has a #define value of 2.
GPR3 = Set equal to zero
GPR4 = Not used
GPR5 = The address of the buffer to receive the master dump table address.

Return Codes: contained in GPR2

0 = Successful completion
4 = Dump table entry not found
6 = Dump table is full
10 = Buffer too small for dump data.

Chapter 6. Managing Minidisks

CONTENTS

About This Chapter 6-3
Minidisk Manager Operations 6-4
  Add a Fixed Disk 6-5
  Create a Minidisk 6-7
  Open a Minidisk for Access 6-9
  Close a Minidisk for Access 6-11
  Delete a Minidisk 6-13
  Create a Minidisk by Sector Number 6-15
  Define Minidisk Characteristics 6-17
  Query a Minidisk 6-19
  Query a Fixed Disk for Minidisks 6-22
Data Access Operations 6-25
Query Operations 6-27

About This Chapter

This chapter explains the functions used when dealing with the minidisk manager. The functions discussed in this chapter may be used to:

- Create or delete a minidisk
- Add a fixed disk for minidisk operations
- Access minidisk data
- Query the minidisk.

Minidisk Manager Operations

The minidisk manager is part of the standard VRM and is automatically defined and initialized when the workstation is IPLed. The virtual machine must attach to it before issuing any commands.

Commands for managing fixed disks and minidisks use the **Send Command SVC** for communication with the minidisk manager. (See "Send Command SVC" on page 4-63.) The IODN specified with this SVC is the minidisk manager's IODN (0x0200).

The **Send Command SVC** supports the following device options to the minidisk manager:

0 = Add a fixed disk
1 = Create a minidisk
2 = Open a minidisk
3 = Close a minidisk
4 = Delete a minidisk
5 = Create a minidisk by sector number
6 = Define minidisk characteristics.
7 = Query a minidisk
8 = Query a fixed disk for minidisks

Valid minidisk IODNs fall in the range from 16,384 to 32,767. However, IODNs from 32,760 to 32,767 are reserved for specific functions.

Unless otherwise specified, all of these operations can be performed synchronously or asynchronously. Because these operations may result in significant I/O delays, the operating system should not specify synchronous operation. Instead, it should wait for completion notification by way of a virtual interrupt. Notification is received on the level and sublevel specified when the operating system attached to the minidisk manager.

The descriptions of these operations include only the parameters and return codes unique to the minidisk manager.

See "Send Command SVC" on page 4-63 for a complete description of the input parameters, return codes, and other common conventions used with the SVC.

Add a Fixed Disk

Description: The Add a Fixed Disk service informs the minidisk manager of a fixed disk that may be used for minidisk operations. The fixed disk is reserved for the exclusive use of the minidisk manager until the delete form of this command is used. This service also updates or creates the minidisk directory and bad block directory on the fixed disk being added to the system.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager by way of the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0 and the device option field must contain a value of 0.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. The disk must have already been defined by the virtual machine prior to issuing this service.

GPR4 = Configuration Options

Two bits in this register define configuration options. They are:

Bit 31 = Set to add the fixed disk to the minidisk configuration.
Bit 30 = Set to delete the fixed disk from the minidisk configuration.

Return Codes: contained in GPR2

-1 = Insufficient resources.
-2 = Fixed disk will not support minidisks.
-4 = Invalid IODN specified.
-36 = Not deleted, one or more minidisks open.
-64 = Disk I/O error.
0 = Successful.
260 = Invalid operation option.
264 = Invalid configuration option.

Comments: The size of the fixed disk can be queried with the **Query Device SVC** after this operation successfully completes. You can use this command to determine the RT PC disk size.

Return code -2 indicates that the fixed disk could not be initialized, either because the minidisk contains bad data or because cylinder 0 of the disk could not be read or written.

As many as 64 fixed-disk IODNs can be added to the system with this command.

The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (<0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation and device options for the command. Bits 16-31 = 0.
Data Word 2 =	0
Data Word 3 =	0.

Create a Minidisk

Description: This service creates a minidisk.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0, and the device option field must contain a value of 1. This command supports the following bit-encoded modifiers to the operation options:

Bits 22-23 These bits determine where on the fixed disk the new minidisk is located. Possible values are:

00 = Default to the next available space.
01 = Low end (first 1/3) of the fixed-disk sectors.
10 = Middle 1/3 of fixed-disk sectors.
11 = High end (last 1/3) of fixed-disk sectors.

Bit 24 Write verify

Setting this bit enables the write verify function provided by the fixed-disk device driver. Note that you should expect a degradation of system performance if you specify this function for a minidisk.

Bit 25 Do not relocate bad blocks.

Bit 26 Paging minidisk.

Note that a paging space minidisk must have a block size of 512 bytes and must use bad block relocation (bit 25 = 0).

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. The minidisk is created on this disk unless a value of 0 is specified, in which case the VRM selects a fixed disk to create the minidisk on.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31. The virtual machine must specify a unique IODN for the minidisk.

GPR4 = Block Size

The logical block size is specified in multiples of 512 bytes. The values in this field may range from 1 to 16 inclusive. A value of zero in this field specifies the default which is 2,048 bytes.

GPR5 = Number Of Blocks

This value represents the number of logical blocks to be allocated for this minidisk.

GPR6 = Minidisk name

This field can contain up to four ASCII characters to represent a minidisk name.

Return Codes: contained in GPR2

-4 =	Fixed disk is not defined.
-16 =	Invalid minidisk number.
-28 =	Insufficient free space on fixed disk.
-64 =	Disk I/O error.
0 =	Successful.
260 =	Invalid operation option.
268 =	Invalid block size.

Comments: The number of logical blocks allocated to a minidisk does not change. Minidisks are assumed to be static objects and, therefore, virtual disks are not supported.

The operation completion information provides some helpful information for recovery from the insufficient free space errors. The number of blocks indicates the maximum continuous free space available on the fixed disk specified or on the VRM-selected fixed disk when zero is specified. In both cases, the corresponding disk number is also returned.

The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (<0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation options for the command. Bits 16-31 = Block size.
Data Word 2	Bits 0-15 = Fixed-Disk Number. This is the IODN of the disk that the minidisk is on. Bits 16-31 = Minidisk Number. This is the IODN of the new minidisk.
Data Word 3	Number Of Blocks — This is the number of logical blocks allocated for this minidisk.

Open a Minidisk for Access

Description: This service opens a minidisk for data access operations.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0, and the device option field must contain a value of 2.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk, in bits 0 through 15, that contains the minidisk. All of the defined fixed disks are searched for the minidisk when 0 is specified.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31.

GPR4 = Access Rights

The minidisk access rights are specified in bits 16 through 19. The following values are supported (write access implies read access):

0 = Read access requested, others allowed read access.
1 = Read access requested, others allowed write access.
2 = Write access requested, others not allowed access.
3 = Write access requested, others allowed read access.
4 = Write access requested, others allowed write access.

Level — This notify interrupt level is specified in bits 20 through 23. Only values between 0 and 6 are allowed.

Sublevel — This notify interrupt sublevel is specified in bits 24 through 31. Only values between 0 and 255 are allowed.

Return Codes: contained in GPR2

-4 =	Fixed disk is not defined.
-16 =	Invalid minidisk number.
-20 =	Minidisk is already opened by a virtual machine.
-24 =	Conflicting access rights requested.
-64 =	Disk I/O error.
0 =	Successful.
260 =	Invalid operation option.
272 =	Invalid level/sublevel specified.

Comments: Minidisk number 0x7FFF is a special number that does not correspond to an actual minidisk. It is specified when the install/update program wants to modify the IPL record on the specified fixed disk. Only one IPL record can be open for access by at most one virtual machine at any instance in time. Otherwise, an error results (return code = -24).

A path identifier is passed back to the requestor in the third data word of the PSB. This path is then used as input for the **Start I/O SVC** in GPR3.

The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (<0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation options for the command. Bits 16-31 = Block size (bits 16 through 31).
Data Word 2	Bits 0-15 = Fixed-Disk Number. This is the IODN of the disk that the minidisk is on. Bits 16-31 = Minidisk Number. This is the IODN of the minidisk.
Data Word 3	Path identifier — Use this as input to the **Start I/O SVC**. The path is the link between the requestor and the minidisk.

Close a Minidisk for Access

Description: This service is used to terminate access to a minidisk for data access operations.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0 and the device option field must contain a value of 3.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk, in bits 0 through 15, that contains the minidisk. All of the defined fixed disks are searched for the minidisk when 0 is specified.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31.

Return Codes: contained in GPR2

-4 = Fixed disk not defined.
-16 = Invalid minidisk number.
-20 = Minidisk not opened by virtual machine.
-64 = Disk I/O error.
0 = Successful.
260 = Invalid operation option.

Comments: This service also cancels all uninitiated I/O to the specified minidisks from a virtual machine.

The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (< 0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation options for the command. Bits 16-31 = 0.
Data Word 2	Bits 0-15 = Fixed-Disk Number. This is the IODN of the fixed disk that the minidisk is on. Bits 16-31 = Minidisk Number. This is the IODN of the minidisk.
Data Word 3	0.

Delete a Minidisk

Description: This service is used to delete a minidisk from a directory. The space used by the deleted minidisk is then marked as free space.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0 and the device option field must contain a value of 4.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. All of the defined fixed disks are searched for the minidisk when 0 is specified.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31.

Return Codes: contained in GPR2

-4 = Fixed disk not defined.
-16 = Invalid minidisk number.
-20 = Minidisk is open.
-64 = Disk I/O error.
0 = Successful.
260 = Invalid operation option.

Comments: The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (<0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation options for the command. Bits 16-31 = 0.
Data Word 2	Bits 0-15 = Fixed-Disk Number. This is the IODN of the fixed disk that the minidisk is on. Bits 16-31 = Minidisk Number. This is the IODN of the minidisk.
Data Word 3	0.

Create a Minidisk by Sector Number

Description: This routine allows you to create a minidisk and specify the minidisk's starting sector number on the fixed disk. This routine allows for a more precise placement of the minidisk than "Create a Minidisk" on page 6-7, which can also be used to create a minidisk.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0, and the device option field must contain a value of 5. This command supports the following bit-encoded modifiers to the operation options:

Bits 22-23 Reserved

Bit 24 Write verify

Setting this bit enables the write verify function provided by the fixed-disk device driver for the new minidisk. Write verify means, for all I/O write operations, that the fixed disk performs the write, then verifies that the CRC written is valid. If the CRC is invalid, the fixed disk relocates the data or returns an I/O error.

Bit 25 Do not relocate bad blocks.

Bit 26 Paging minidisk.

Note that a paging space minidisk must have a block size of 512 bytes and must use bad block relocation (bit 25 = 0). You may specify the write verify option for paging minidisks, but expect a degradation of system performance if you do so.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. You must specify an IODN in this register, as no default disk is provided with this service. If you want to use a system default disk, you do not need the precision of this service and should use "Create a Minidisk" on page 6-7.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31. The virtual machine must specify a unique IODN for the minidisk.

GPR4 = Sector number

This register contains the physical sector number at which to locate the new minidisk. This value must be greater than 4 times the number of sectors per

track on the fixed disk, as sectors below this value are reserved for the POST control block, configuration record, minidisk directory, and so on.

GPR5 = Block Size

The logical block size is specified in GPR5 bits 0-4 as multiples of 512 bytes. The values in this field may range from 1 to 16 inclusive. A value of zero in this field specifies the default which is 2,048 bytes.

Number of Blocks — Contains the number of logical blocks to allocate for this minidisk in bits 5-31.

GPR6 = Minidisk name

This field can contain up to 4 ASCII characters to represent a minidisk name. This is an option; no checks are made on the name.

Return Codes: contained in GPR2

-4 = Fixed disk is not defined.
-8 = Invalid sector number.
-16 = Invalid minidisk number.
-28 = Insufficient free space on fixed disk.
-64 = Disk I/O error.
0 = Successful.
260 = Invalid operation option.
268 = Invalid block size.

Comments: The number of logical blocks allocated to a minidisk does not change. Minidisks are assumed to be static objects and, therefore, virtual disks are not supported.

The operation completion information provides some helpful information for recovery from the insufficient free space errors. The number of blocks indicates the maximum contiguous free space available on the fixed disk specified. In this case, the corresponding fixed-disk number is also returned.

The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (< 0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation options for the command. Bits 16-31 = Block size.
Data Word 2	Bits 0-15 = Fixed-Disk Number. This is the IODN of the disk that the minidisk is on. Bits 16-31 = Minidisk Number. This is the IODN of the new minidisk.
Data Word 3	Number Of Blocks — This is the number of logical blocks allocated for this minidisk.

Define Minidisk Characteristics

Description: This service is used to specify the characteristics of a minidisk.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 0 and the device option field must contain a value of 6.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. All of the defined fixed disks will be searched for the minidisk when 0 is specified.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31.

GPR4 = Characteristics

This register contains the minidisk's characteristics. They are defined as follows:

Bit 24 = This indicates that the write verify feature is in effect for the minidisk.
Bit 25 = Reserved
Bit 26 = Reserved
Bit 27 = This specifies the AIX Operating System file system minidisk.
Bit 28 = This specifies the AIX Operating System minidisk.
Bit 29 = This specifies the coprocessor minidisk.
Bit 30 = This specifies the VRM minidisk.
Bit 31 = This specifies that the operating system is automatically IPLed when the VRM is IPLed.

Return Codes: contained in GPR2

-4 = Fixed disk not defined.
-16 = Invalid minidisk number.
-64 = Disk I/O error.
0 = Successful.
260 = Invalid operation option.

Comments: The operation completion information is as follows:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (< 0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation options for the command. Bits 16-31 = 0.
Data Word 2	Bits 0-15 = Fixed-Disk Number. This is the IODN of the fixed disk. Bits 16-31 = Minidisk Number. This is the IODN of the minidisk.
Data Word 3	0.

Query a Minidisk

Description: This service is used to query a minidisk. Information about the minidisk is passed back by way of a buffer.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 1 and the device option field must contain a value of 7.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. All defined fixed disks will be searched for the minidisk when 0 is specified.

Minidisk Number — Contains the IODN of the minidisk in bits 16 through 31.

GPR4 = Reserved

GPR5 = Buffer address

This register contains the address of the minidisk buffer.

GPR6 = Buffer length

This register contains 24, which is the length of the minidisk buffer.

The fields within the query minidisk structure upon completion of the SVC request are defined in Figure 6-1 on page 6-20.

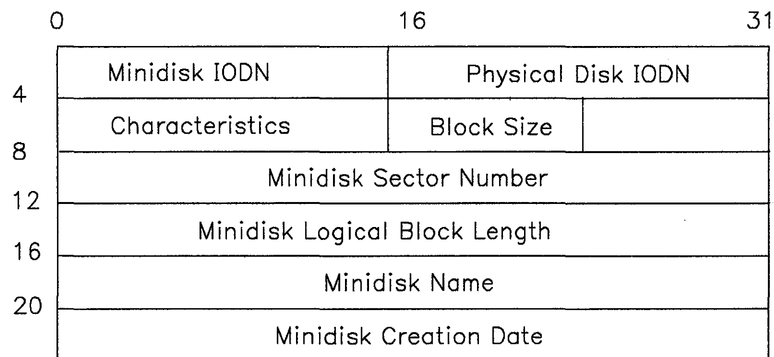


Figure 6-1. 'Query Minidisk' Structure

The fields within the minidisk structure are defined as follows:

Field	Description
Characteristics	Minidisk characteristics (if bit is set): Bit 5 = Write verify Bit 6 = Low end disk position Bit 7 = Middle of disk position Bit 8 = High end disk position Bit 9 = No bad block management Bit 10 = Paging space Bit 11 = AIX Operating System file Bit 12 = AIX Operating System Bit 13 = Coprocessor Bit 14 = VRM Bit 15 = Automatic IPL.
Block Size	Contains the size of a logical block in multiples of sectors (1-16).
Sector Number	The sector number of the fixed disk where the minidisk resides.
Logical Block Length	Minidisk length in logical blocks.
Name	Minidisk name, specified when created.
Creation date	Time in seconds (since Jan. 1, 1970) that the minidisk was created.

Return Codes: contained in GPR2

-4 = Fixed disk not defined.
-16 = Invalid minidisk number.
-64 = Disk input/output error.
0 = Successful.

12 =	Buffer not word-aligned.
260 =	Invalid operation option.

Comments: The length of the minidisk is found by multiplying the logical block size by the logical block length.

The operation completion information has the following structure:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (<0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation and device options for the command. Bits 16-31 = Segment ID of the buffer.
Data Word 2	Buffer address
Data Word 3	0.

Query a Fixed Disk for Minidisks

Description: This service maps the minidisks that are located on a particular fixed disk. Information about each minidisk is passed back by way of a buffer.

Format: See "Send Command SVC" on page 4-63.

Calling Register Conventions:

GPR2 = IODN

This register contains the IODN of the minidisk manager (0x0200) in bits 0 through 15. The operating system must have previously attached to the minidisk manager with the **Attach Device SVC**.

Operation Options — Contains the operation options in bits 16 through 31. The command extension bit must be 1 and the device option field must contain a value of 8.

GPR3 = Fixed-Disk Number

This register contains the IODN of the fixed disk in bits 0 through 15. Specifying zero in this register results in querying the minidisks contained on the fixed disk from which the VRM was IPLed.

GPR4 = Reserved

GPR5 = Buffer address

This register contains the address of the minidisk buffer.

GPR6 = Length of buffer

This register contains the length of the buffer, in bytes.

Figure 6-2 on page 6-23 shows the structure returned from a Query All Minidisks request.

0	16	31
0	Physical Disk IODN	Number of Minidisks Returned
4	Length of Physical Disk	
8	Total number of minidisks on this real disk	
12	Query Minidisk Structure	
	Query Minidisk Structure	
8 + (24*n)		

Figure 6-2. 'Query All Minidisks' Structure

The fields within the structure are defined as follows:

Field	**Description**
Physical Disk IODN	The IODN of the physical disk.
Minidisks Returned	The number of minidisks that could fit into the buffer allocated.
Total # of Minidisks	All minidisks on the fixed disk
Length	Total number of sectors on the fixed disk
Query Minidisk Structure	A substructure for each minidisk located on the fixed disk.

Return Codes: contained in GPR2

-4 = Fixed disk not defined.
-30 = Insufficient buffer space.
-64 = Disk input/output error.
0 = Successful.
12 = Buffer not word-aligned.
260 = Invalid operation option.

Comments: If the buffer is not large enough, the minidisk manager will update as much of it as possible and pass back the number of minidisks it contains. The PSB will contain the total number of minidisks on the fixed disk at this time.

The returned structure includes any free areas on the disk. Free areas are also considered to be minidisks. Minidisks that consist of free space have IODNs of 0. In fact, all fields except for the minidisk sector number, minidisk length, and logical block size will be 0. Free space minidisks have a logical block size of 1 sector. The

characteristics field of free minidisks contains indicators as to where they reside on the disk (low, middle, or high).

The operation completion information has the following structure:

Status Flags	00010100
Overrun Count	0
Status Word	Bits 0-15 = Return codes (<0) as specified above. Bits 16-31 = IODN of the minidisk manager.
Data Word 1	Bits 0-15 = Operation and device options for the command. Bits 16-31 = Segment ID of the buffer.
Data Word 2	Buffer address.
Data Word 3	The total number of minidisks on the fixed disk.

Data Access Operations

Description: The **Start I/O SVC** can communicate data access operations to the minidisk manager. In order to start an I/O operation to a minidisk, the device-dependent parameters are inserted in the command header of the CCB as follows:

Offset		
0	Reserved	Options
4	Reserved	Minidisk number
8	Block Number	
12	Reserved	
24		

Figure 6-3. Minidisk Command Header Format

Format: See "Send Command SVC" on page 4-63.

The option section of the minidisk CCB header shown above can have the following values:

Read: A device option of 0 specifies the read option. One or more command elements follow which contain the length and memory address for each chain link.

Write: A device option of 1 specifies the write option. One or more command elements follow which contain the length and memory address for each chain link. This command writes data in 512 byte blocks. If the sum of all command element lengths is not a multiple of 512, the remainder of the data in the last sector is unpredictable.

Position: A device option of 2 specifies the position option. No command elements follow. This function is available to virtual machines supporting disk arm scheduling.

This is an advisory command and may be ignored by the VRM, depending on other system activity.

Return Codes: contained in GPR2

<0 =	Disk I/O error.
0 =	Successful.
12 =	CCB or buffer alignment error.
13 =	Length specification error.
16 =	Invalid IODN.
260 =	Invalid operation option.
276 =	Invalid block number.
280 =	Invalid access rights.

Comments: Only one word is required for device-dependent parameters. This word specifies the block number for all options. If an invalid block number is specified, a value of 276 is returned. If the device option is equal to 2 (position), the positioning of the cylinder number is computed by the IOS.

For all minidisk data transfers, the memory address must be on a word boundary. If it is not, a return code of 12 is returned. The length specification must define an exact number of words. If it does not, a return code of 13 is returned.

The status flags in the PSB, byte offset 18, indicate whether the interrupt was solicited or unsolicited. With the current disk hardware, the interrupt is always solicited.

The minidisk manager controls minidisk I/O operations with three VRM runtime routines called by the fixed-disk device driver. All installed device drivers must have a defined interface to these routines. See "Minidisk Manager Check (_mdmchk)" on page 5-105, "Check for Bad Blocks (_chkblk)" on page 5-103, and "Minidisk Bad Blocks (_badblk)" on page 5-104 for more information.

Query Operations

Query Device SVC provides information about the minidisk manager and minidisks.

Information about the minidisk manager and its last completed management operation can be obtained by specifying the minidisk manager's IODN (0x200) in the QDS. The minidisk manager does not provide any information in the device-dependent information field of the QDS. See each of the minidisk management operations for a description of its operation completion information.

Chapter 7. Floating-Point Services

CONTENTS

About This Chapter 7-3
VRM Floating-Point Support 7-4
  Floating-Point Hardware Descriptions 7-4
  Floating-Point Accelerator Register Set Allocation 7-7
  Current Register Set Control 7-8
  MC68881 Registers 7-8
  MC68881 Hardware/Software Assist Modes 7-10
  Saving and Restoring Floating-Point Registers 7-12
  How the Emulation Code Works 7-15
  FPA and AFPA Status 7-15
  Not-a-Numbers (NaN) 7-17
  Interrupts 7-18
  Floating-Point Operations without Emulation Code 7-22

About This Chapter

This chapter describes the floating-point computation services available for the RT Personal Computer. Floating-point support is provided by way of several hardware configurations. This chapter describes the differences between those hardware configurations, as well as register set allocation, error and exception handling, and a VRM software emulation feature that supports some of the floating-point hardware.

VRM Floating-Point Support

The RT Personal Computer provides floating-point computation with several different hardware features. Each hardware configuration achieves a different level of conformance with the Institute of Electrical and Electronic Engineers (IEEE) standard as described in "IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std. 754-1985)." For the hardware that does not provide complete IEEE conformance, the virtual machine and/or VRM must provide a software assist to more closely implement the IEEE standard. See "How the Emulation Code Works" on page 7-15.

The VRM provides many services to the virtual machine in support of floating-point operations. These include:

- Determination of floating-point hardware configuration
- Floating-point register set allocation
- Control of the current register set
- Emulation code to ensure IEEE conformance for the various floating-point hardware
- Interrupts (trap-enabled exceptions) to the virtual machine.

These services allow virtual machines to detect and control the floating-point hardware. For the Floating-Point Accelerator (FPA) and Advanced Floating-Point Accelerator (AFPA), which both offer multiple register sets, these services allow a single virtual machine to control multiple sets of floating-point registers for use by multiple users or processes.

Floating-Point Hardware Descriptions

The floating-point hardware configurations include:

- IBM RT PC Floating-Point Accelerator (FPA)

  The FPA option plugs into the feature slot on the planar and provides up to 32 sets of floating-point registers. Each register set consists of sixteen 32-bit registers. Of the 16 registers in a set, the FPA reserves two registers, the instruction exception register (IER) and the floating-point status register (FPSR). The IER saves the actual opcode of a command that results in an exception. The FPSR is described in "FPA and AFPA Status" on page 7-15.

  The FPA supports double-precision operations by combining even-odd register pairs. As many as seven 64-bit registers can be defined this way. Double-precision registers are specified by the address of the even-numbered member, and the 32 high-order bits of the value are placed in the even-numbered register.

  The FPA does not fully implement the IEEE standard without the VRM emulation code and some function provided by the virtual machine (such as handling remainders, round to integer operations, square roots, and some decimal conversion operations).

If the virtual machine needs more than 32 register sets for use by its processes, the register sets can be shared. Before swapping a register set to another process, the virtual machine must save and restore the state of the FPA, including:

- The 14 general-purpose floating-point registers (registers 0-13)
- The status register (register 14)
- The instruction exception register (register 15)
- Any pending exception.

The virtual machine can save and restore all the registers in a set, but cannot save and restore a pending exception due to hardware limitations. To circumvent this restriction, make sure that the register set selected for the swap has no pending exceptions. See "Saving and Restoring Floating-Point Registers" on page 7-12 for details.

- IBM RT PC Advanced Floating-Point Accelerator (AFPA)

  The AFPA option also plugs into the feature slot on the planar, so the FPA and AFPA are mutually exclusive on the same RT PC. The AFPA provides up to 32 sets of floating-point registers. Each register set consists of sixty-four 32-bit registers. In addition, each set provides two additional registers, an instruction exception register (IER) and a floating-point status register (FPSR). The IER saves the actual opcode of a command that results in an exception. The FPSR is described in "FPA and AFPA Status" on page 7-15.

  Note that the AFPA status registers are separate from the main body of general purpose registers (0-63) in each AFPA register set, as opposed to the FPA, which allocates 2 status registers from its 16-register set. As such, the AFPA status registers have unique **read** and **write** commands. All the floating-point commands are defined in *IBM RT PC Hardware Technical Reference.*

  The AFPA supports double-precision operations by combining even-odd register pairs. As many as thirty-two 64-bit registers can be defined this way. Double-precision registers are specified by the address of the even-numbered member, and the 32 high-order bits of the value are placed in the even-numbered register.

  The AFPA can be used in conjunction with the IBM RT PC Advanced Processor Card to provide additional floating-point capabilities, such as the use of Direct-Memory Access for **read** and **write** floating-point operations. A PIO or DMA instruction is identified by the setting of bits 0-7 of the floating-point instruction. Bits 0-7 are 0xFE for a DMA instruction and 0xFF for a PIO instruction.

  Note that, due to overlapped instruction execution on the APC, you may have to synchronize certain DMA **read** and **write** operations because the processor can alter the memory location of a DMA transfer before the transfer is complete. For example, suppose a DMA **write** instruction to move the value from address 0x420 to the AFPA is followed immediately by an instruction that changes the value at 0x420. A race condition would be created and the location may be changed before the DMA operation completes. The best way to synchronize DMA operations is to issue a **read** instruction to the DMA length register, although a **clear status bit on the permanent zero bit** command can also be used.

When the AFPA is used with the APC, the APC provides a third control register to specify the length of DMA transfers that are greater than two words. The length value is specified in bits 26-31 of the DMA length register. Bits 0-25 are set to zero.

Note that eight of the 32 register sets are reserved for system use, leaving 24 sets for use by processes. If the virtual machine needs more than 24 register sets for use by its processes, the register sets can be shared. Before swapping a register set to another process, the virtual machine must save and restore the state of the AFPA, including:

- The 64 general-purpose floating-point registers
- The status register
- The instruction exception register
- The DMA length register
- Any pending exception.

The virtual machine can save and restore all the registers in a set. Saving and restoring a pending exception is more difficult, however, and requires some assistance from the VRM. See "Saving and Restoring Floating-Point Registers" on page 7-12 for details.

- Support provided with the IBM RT PC Advanced Processor Card (APC)

The APC has an onboard floating-point coprocessor (MC68881) that provides one floating-point register set. The MC68881 provides a register set consisting of eight general-purpose data registers (0-7), a control register, and a status register. For a complete description of programming interfaces, instruction set, timing, and so on for the MC68881, see *MC68881 Floating-Point Coprocessor User's Manual*, First Edition, 1985, available from Motorola, Inc.

Two additional registers are provided by the APC, a DMA length register for DMA transfers of more than two words and a last opcode register that contains the MC68881 opcode of a floating-point instruction that resulted in an exception. Note that IBM bit-numbering conventions are used when defining MC68881 registers.

The eight MC68881 general-purpose registers are each three words in length, providing single, double, and extended precision values. Although the VMI supports only single and double precision operations, the MC68881 can be used in extended precision mode, except when underflow, overflow, or inexact exceptions are enabled. When these exceptions are enabled, the designated result must be reported to the virtual machine in the correct precision. Thus, the result must always be able to be converted to single or double precision without causing any further exceptions. Because of this restriction, the MC68881 must be run in non-extended precision in order to guarantee a result that can be converted.

Before swapping the single register set to another process, the virtual machine must save and restore the state of the MC68881, including:

- The eight general-purpose registers
- The status register
- The control register
- The DMA length register
- The last opcode register
- Any pending exception.

The virtual machine can save and restore all the items above except the last opcode register. The VRM provides the facility to save and restore the last opcode register. See "Saving and Restoring Floating-Point Registers" on page 7-12 for details.

The MC68881 provides a full implementation of the IEEE standard, including a full set of trigonometric and transcendental functions. The only VRM support required is to report interrupts to the virtual machine for trap-enabled exceptions (see "MC68881 Registers" on page 7-8) and to assist in saving and restoring the last opcode register.

If either the FPA or AFPA is present in a system that also has an APC, the FPA or AFPA floating-point support is used rather than that provided by the MC68881.

As noted, a single RT PC can have two floating-point hardware configurations. At IPL time, the VRM determines the hardware configuration and selects the floating-point support. The selection is made in the following order:

- The AFPA is always selected if it is present and functioning in the system.
- The FPA is selected next if it is present and functional in the system.
- The APC floating-point coprocessor is selected only if the APC is present in the system and the AFPA or FPA are not present in the system.

When the floating-point hardware configuration is determined, the VRM places a value at byte 0xCC of the virtual machine's page 0. Possible values for this byte are defined as follows:

- 0 = no floating-point hardware is present
- 1 = FPA is present
- 2 = APC floating-point coprocessor is present
- 4 = AFPA is present
- C = AFPA and APC present (DMA operations supported).

For more information on the specifics of the floating-point hardware options and the entire floating-point command set, see *IBM RT PC Hardware Technical Reference.*

Floating-Point Accelerator Register Set Allocation

The VRM allows virtual machines to allocate and deallocate the register sets of the various floating-point hardware features with the **Allocate Floating-Point Register Sets SVC** and the **Free Floating-Point Register Sets SVC**. A virtual machine can also query which register sets it owns with the **Query Floating-Point Register Sets SVC**.

Note: Some of the floating-point register sets may be reserved for use by the VRM.

Before a virtual machine can use any floating-point hardware, it must allocate at least one register set. Note that the APC floating-point coprocessor (MC68881) provides only one register set. Therefore, only one process can use this floating-point hardware at a time. If a virtual machine has more than one process that wants to use the MC68881, more than 32 processes that want to use the FPA, or more than 24 processes that want to use the AFPA, the virtual machine (with VRM

assistance) must perform the context switch for the floating-point hardware. See "Saving and Restoring Floating-Point Registers" on page 7-12 for more details.

Current Register Set Control

The VRM controls the setting of the current register set in the FPA or AFPA by changing the register-set value when a virtual machine is dispatched. This is done to allow a virtual machine to handle multiple register sets. This function requires that the virtual machine tell the VRM the register set that should be made current.

When a virtual machine is first IPLed, it does not own any floating-point register sets until it issues the **Allocate Floating-Point Register Sets SVC**. When a set is successfully allocated, the virtual machine indicates the active register set by placing a value (0-31) in the byte at location 0xD7 (see Figure 2-1 on page 2-5 for a complete description of this memory location). The virtual machine must then update memory-mapped control blocks with an Execution Control SVC, such as **No Operation SVC**. If the value in 0xD7 is valid (in-range and owned by the virtual machine), the VRM sets that value as the current register set for the floating-point hardware. If the value is not valid, the VRM replaces the value with 0xFF and locks the Floating-Point Accelerator.

To lock the floating-point hardware, the virtual machine must set location 0xD7 to 0xFF and issue an Execution Control SVC. The register set last specified remains the current register set for the virtual machine until the byte at 0xD7 is reset.

MC68881 Registers

In addition to the eight general-purpose floating-point registers, the MC68881 provides one 32-bit status register and one 32-bit control register.

The status register contains a condition code byte, an exception status byte, quotient bits, and an accrued exception byte. This register is defined as follows:

- Bits 0-7 (condition code)

  The bits in the condition code byte are used in comparisons of data to determine specific order between two operands. Use the following equations with the condition bits:

  - Unordered = NaN
  - Less than = N and ¬ (NaN) and ¬Z
  - Equal = Z and ¬(NaN)
  - Greater than = ¬(N) and ¬(Z) and ¬(NaN).

The bits in the condition code byte are defined as follows:

Bits 0 through 3 are reserved.
Bit 4 indicates negative and is used as a comparison bit for negative.
Bit 5 indicates zero and is used as a comparison bit for zero.
Bit 6 indicates infinity and is used as a comparison bit for infinity.
Bit 7 indicates not-a-number and is used as a comparison bit for unordered compares.

- Bits 8-15 (quotient)

  The quotient byte contains the seven least-significant bits of the quotient and the sign of the entire quotient of a modulo or remainder instruction.

- Bits 16-23 (exception status)

  This byte contains a bit for each floating-point exception that occurred from the last arithmetic or move instruction. Multiple exceptions on one instruction are possible and should be processed accordingly. Exception status bits are organized in decreasing priority from left to right. The bits in the exception status byte are defined as follows:

  Bit 16 indicates a branch or set on unordered exception occurred.
  Bit 17 indicates an invalid operand exception occurred as a result of a signalling NaN.
  Bit 18 indicates an invalid exception occurred.
  Bit 19 indicates an overflow exception occurred.
  Bit 20 indicates an underflow exception occurred.
  Bit 21 indicates a divide-by-zero exception occurred.
  Bit 22 indicates an inexact exception occurred.
  Bit 23 indicates an inexact decimal exception occurred.

- Bits 24-31 (accrued exceptions)

  The accrued exceptions byte contains a bit for each of the five exception conditions defined by the IEEE standard. These bits are derived from the exception status bits according to the following equations:

  - Invalid = BSUN, SNaN, or OPERR
  - Overflow = OVFL
  - Underflow = UNFL and INEX
  - Divide-by-zero = Divide / zero
  - Inexact = OVFL, INEX, or INEX Dec.

  The bits in the accrued exception are not cleared unless specifically written by the user. The bits in the accrued exception byte are defined as follows:

  Bit 24 indicates an invalid operand exception occurred.
  Bit 25 indicates an overflow exception occurred.
  Bit 26 indicates an underflow exception occurred.
  Bit 27 indicates a divide-by-zero exception occurred.
  Bit 28 indicates an inexact exception occurred.
  Bit 29 through 31 are reserved.

The second MC68881-provided register, the control register, contains the exception enable byte and the mode control byte. The exception enable byte has one bit defined for each exception class, and the mode control byte determines the rounding precision and rounding mode to use. The control register is defined as follows:

- Bits 0 through 15 are reserved.
- Bit 16, when set, causes an exception to be generated for branch or set on unordered operations.
- Bit 17, when set, causes an exception to be generated for signalling NaN operations.
- Bit 18, when set, causes an exception to be generated for invalid operations.
- Bit 19, when set, causes an exception to be generated for overflow operations.
- Bit 20, when set, causes an exception to be generated for underflow operations.
- Bit 21, when set, causes an exception to be generated for divide-by-zero operations.
- Bit 22, when set, causes an exception to be generated for inexact operations.
- Bit 23, when set, causes an exception to be generated for inexact decimal operations.
- Bits 24 and 25 indicate the rounding precision to use. These bits are defined as follows (binary):
  - 00 = extended
  - 01 = single
  - 10 = double
  - 11 = reserved.
- Bits 26 and 27 indicate the rounding mode to use. These bits are defined as follows (binary):
  - 00 = round to nearest
  - 01 = round toward zero
  - 10 = round toward -infinity
  - 11 = round toward +infinity.
- Bits 28 through 31 are reserved.

MC68881 Hardware/Software Assist Modes

The APC provides the logic to implement the MC68881 protocol. This logic is manifested in two modes provided by the APC, a hardware assist mode and a software assist mode.

Hardware Assist Mode

Hardware assist mode is initiated by issuing the processor **store** instruction that contains the MC68881 command, specification of any operand transfer, and the 32-bit DMA address. Software executes a single **store** instruction. The APC initiates the command to the MC68881 and carries out the necessary protocol to finish the instruction, including instructions that involve transfers of operands. These operands are read from or written to the system address that is specified in the data packet of the **store** instruction.

A hardware assist mode instruction to the MC68881 can be identified by a 0xFC in bits 0-7 of the instruction.

A typical APC to MC68881 protocol sequence for a memory-to-register **add** operation is performed in the following steps:

1. Write the 68881 **add** command to the command coprocessor interface register (CIR).

   Note that MC68881 protocols that do not begin by writing to the command CIR are not supported in hardware mode by the APC logic. Instructions such as **Fsave** and **Frestore**, branch instructions, and control instructions such as **Abort** do not use this protocol and must use software assist mode to perform these instructions.
2. Poll the response CIR until the 68881 indicates it is ready for the operand.
3. Write the data (which was obtained from memory) to the 68881 operand CIR for the required number of words.
4. Poll the response CIR until the 68881 indicates it requires no further service.

The hardware assist mode data format is a 32-bit value defined as follows:

Bits 0-7 = Address extension for hardware assist mode. Set to 0xFC.
Bits 8-9 = Reserved
Bits 10-13 = Operation direction field (in binary)
- 0000 = memory to 68881
- 1111 = 68881 to memory
- 0111 = to or from DMA length register

Bits 14-16 = Op class, such as register to register, memory to register, move multiple, and so on.
Bits 17-19 = RX field. This field contains either the source register, data format, or register list depending on the op class.
Bits 20-22 = RY field. This field contains either the destination register (for register-to-register and memory-to-register operations), the source register (for register-to-memory operations), or is unused.
Bits 23-29 = Extension field. This field defines the type of arithmetic instruction (such as add, sine, and so on) for register-to-register and memory-to-register commands.
Bits 30-31 = Transfer size (in binary)
- 00 = no data transfer
- 01 = 1 word
- 10 = 2 words
- 11 = *n* words transfer defined by the DMA length register.

Hardware assist mode has another restriction when used with instructions that may take a relatively long time to complete, such as remainder, sine, modulo, and so on. If a remainder command is followed by another hardware assist mode command, the hardware is occupied until the first command completes. This can result in interrupt latency problems resulting in lost interrupts. Also, protocol violations can occur when a decimal move in instruction is followed too closely by another command. To avoid these problems, the response CIR should be polled with a software assist mode command after each long command. The polling should continue until the response comes back as 0x08020000, which indicates that all commands have been completed.

Software Assist Mode

Software assist mode is intended to be used only for commands that cannot be implemented by hardware assist mode or for protocol sequences that do not begin by writing to the command CIR. The primary function of software assist mode is to provide access to the CIRs on the MC68881.

A software assist mode instruction to the MC68881 can be identified by a 0xFD in bits 0-7 of the instruction.

Access to the CIRs is gained by ORing the CIR offset with the value 0xFD000000 to read and write the registers. The offsets to the various CIRs are defined as follows:

CIR	Offset
Response	0x00
Control	0x02
Save	0x04
Restore	0x06
Command	0x0A
Condition	0x0E
Operand	0x10
Register select	0x14

The 68881 transfers all 16-bit registers on the upper 16 bits, so only fullword accesses should be used. For write operations, the lower 16 bits should be zeroes; for read operations, the lower 16 bits are undefined.

For more information on the CIRs, see *MC68881 Floating-Point Coprocessor User's Manual.*

Saving and Restoring Floating-Point Registers

When a virtual machine wants to provide floating-point services for more processes than it has register sets allocated, the virtual machine must save the state of the floating-point hardware (FPA, AFPA, or MC68881) for the first process and restore the state of the second process. Each floating-point hardware configuration requires differing amounts of VRM assistance to perform this context switch. The requirements for each configuration are defined in the following section.

Register Swap for the FPA

The virtual machine can save and restore the general-purpose, status, and instruction exception registers for the FPA, but cannot save and restore any pending exceptions. Therefore, the register set you select for the swap must have no pending exceptions. This is done by setting the byte at virtual machine page 0 location 0xD7 to 0xFB and issuing an execution control SVC (such as a **No Operation SVC**). The VRM polls all register sets on the FPA until a set with no pending exceptions is found or all sets have been polled. If a set with no exceptions is found, the VRM activates that register set and places the register set number in 0xD7 of the virtual machine's page 0. If all sets have pending exceptions, the VRM will lock the FPA and place 0xFF in location 0xD7. In this case, the virtual machine should either terminate the process requesting a new register set or place the

process in a wait state until a clean register set becomes available. The operations required for a register swap for the FPA should be done in the following order:

1. Set virtual machine page 0 location 0xD7 to 0xFB to request a clean register set.
2. Issue a **No Operation SVC**.
3. The VRM places the clean register set number (if one exists) in location 0xD7 and activates that register set.
4. The virtual machine must save the contents of the 14 general-purpose register sets, the status register, and the instruction exception register for the first process.
5. The virtual machine must then restore the contents of the 14 general-purpose register sets, the status register, and the instruction exception register with the values from the second process.

Register Swap for the AFPA

The virtual machine can save and restore the general-purpose, status, instruction exception, and DMA length registers (if APC is configured) for the AFPA, but cannot save and restore a pending exception without VRM assistance.

The first step in register swapping for the AFPA is to choose the register set to swap and save any pending exceptions associated with that register set. To do this, OR the selected register set number with 0x80, place this value at location 0xD7 of the virtual machine's page 0, and issue a **No Operation SVC**. For example, register set 0x11 ORed with 0x80 yields 0x91. The 0x91 value would be placed in location 0xD7.

The VRM then issues a command to the AFPA to activate the register set, read the current status, and clear any exceptions. The value of the AFPA status register, which contains information regarding the existence and type of any exception, is placed at location 0xB4 of the virtual machine's page 0. When this procedure completes, the virtual machine can proceed with saving the state of the AFPA registers.

For the restore portion of the swap procedure, the virtual machine should first restore the general-purpose registers, instruction exception register, DMA length register (if APC is configured), and status register. If bit 19 of the status register is set, an exception is pending and the virtual machine must restore the pending exception. See "FPA and AFPA Status" on page 7-15 for details. To do this, OR the selected register set number with 0xC0, place this value at location 0xD7 of the virtual machine's page 0, and issue a **No Operation SVC**. The VRM then issues a command to the AFPA that causes pending exceptions indicated by the restored status register to be made pending again. This procedure should be performed only when bit 19 of the saved status register indicates a pending exception.

The operations required for a register swap for the AFPA should be done in the following order:

1. Select a register set to be swapped (for example, register set 0x0F).
2. Set location 0xD7 to the value yielded from the OR of the register set number and 0x80 (0x0F ORed with 0x80 yields 0x8F).
3. Issue a **No Operation SVC**. Location 0xB4 now contains the status register value that must be saved as part of the process state.
4. Save the DMA length register if APC is configured.

5. Save the contents of the 64 general-purpose registers and the instruction exception register for the first process.
6. Restore the contents of the 64 general-purpose register sets, the instruction exception register, and the DMA length register (if APC is configured) for the second process.
7. Restore the status register. If the restored status register indicates a pending exception, do the following:
   a. Set location 0xD7 to the value yielded from the OR of the register set number and 0xC0 (0x0F ORed with 0xC0 yields 0xCF).
   b. Issue a **No Operation SVC**.

Register Swap for the MC68881

The virtual machine can save and restore the general-purpose, status, control, and DMA length registers. In addition, pending exceptions can also be saved and restored by the MC68881. However, because the last opcode register cannot be accessed in problem state, VRM assistance is required for saving and restoring this register.

Before reading the MC68881 register, a virtual machine should own the register set as a result of issuing an **Allocate Floating-Point Register Set SVC**. To read the register, place the value 0xFC at location 0xD7 of the virtual machine's page 0 and issue a **No Operation SVC**. The VRM unlocks the 68881 if it was locked and reads the last opcode register. This 16-bit value is returned to location 0xC8 of the virtual machine's page 0.

If the virtual machine is the owner of (has allocated) the MC68881 register set, it can save and restore the last opcode register in a single call to the VRM. To perform this function, the virtual machine places in location 0xC8 the value it wants to write to the last opcode register. Now place the value 0xFD in virtual machine page 0 location 0xD7 and issue a **No Operation SVC**. The VRM then unlocks the 68881 if it was locked and reads the last opcode register. The value placed at location 0xC8 will then be written to the last opcode register and the opcode that was read from the register earlier is returned in location 0xC8.

The operations required for a register swap for the MC68881 should be done in the following order:

1. Save the DMA length register.
2. Issue the **Fsave** command using software assist mode commands to save the internal state of the MC68881.
3. If bit 4 of the bus interface unit (BIU) flags of the **Fsave** state frame indicate that an exception is pending, do the following:
   a. Set location 0xD7 to 0xFC.
   b. Issue the **No Operation SVC**. Location 0xC8 will then contain the value from the last opcode register.

   See *MC68881 Floating-Point Coprocessor User's Manual* for details on the BIU flags.
4. Save the eight general-purpose registers.
5. Save the status and control registers.
6. Restore the status and control registers with the values from the second process.
7. Restore the general-purpose registers with the values from the second process.

8. Issue the **Frestore** command using software assist mode commands to restore the internal state of the MC68881.
9. If the BIU flags of the **Frestore** state frame indicate that an exception is pending, do the following:
   a. Set location 0xC8 to the opcode register value to be restored.
   b. Set location 0xD7 to 0xFD.
   c. Issue the **No Operation SVC**.

How the Emulation Code Works

The VRM provides emulation code for floating-point operations that the floating-point hardware cannot handle. For example, the FPA cannot handle a divide-by-zero operation. When this situation occurs, the FPA causes an exception to the VRM. The VRM emulation code determines that a divide-by-zero was requested by the FPA. The VRM then calculates the correct default result according to the IEEE standard and writes the result to the destination register (if traps are disabled for divide-by-zero operations). If traps are enabled, the VRM emulation code issues a machine communications interrupt to the virtual machine as described in "Floating-Point Exceptions" on page 7-18.

The emulation code simply provides the correct answer for an operations if one exists. When a simple correct answer exists, the emulation code is transparent (except for possibly a slight decrease in performance) to the virtual machine.

Floating-point exceptions include:

- Divide-by-zero
- Overflow
- Underflow
- Inexact results
- Other invalid operations.

For more information, see "Floating-Point Exceptions" on page 7-18.

FPA and AFPA Status

Each floating-point register set has a floating-point status register (FPSR) dedicated to monitoring status. For the FPA, the FPSR is one of the 16 general-purpose registers; for the AFPA, the FPSR provided in addition to the 64 general-purpose registers. The bits in a floating-point status register have different definitions from a hardware or software point of view. The bit definitions provided here are from the software point of view. See *IBM RT PC Hardware Technical Reference* for the hardware definitions of these status bits.

The status register bits have the following software meanings when they are set (are equal to one). Note that certain status conditions are reflected by pairs of bits with specific meanings. For example, bits 2 and 3, 4 and 5, 6 and 7, 8 and 9, and 25 and 26 each have a bit that indicates whether a

particular condition occurred and another bit that determines whether (if bit 0 is set) to generate an interrupt for the condition.

The FPSR bits for the FPA and AFPA are defined as follows:

Bit	Meaning
0	Generates an interrupt to the virtual machine if an exception occurs.
1	Is the OR of bits 2, 4, 6, 8, and 25.
2	Indicates an invalid operation occurred.
3	Enables an exception for an invalid operation.
4	Indicates a divide-by-zero operation was attempted.
5	Enables an exception for divide-by-zero operations.
6	Indicates an overflow condition occurred.
7	Enables an exception for overflow conditions.
8	Indicates an underflow occurred.
9	Enables an exception for underflow conditions.
10-17	Reserved.
18	Indicates that the FPA had a bus parity exception.
19	Indicates an exception is pending.
20	Reserved.
21-22	Compare result These bits are defined as follows: 00 = less than 01 = equal 10 = greater than 11 = unordered.
23-24	Rounding mode These bits are defined as follows: 00 = round to nearest 01 = round toward zero 10 = round toward +infinity 11 = round toward -infinity.
25	Indicates an inexact result occurred.
26	Enables an exception for inexact results.

27	Must be set to zero.
28	Must be set to one.
29-31	Last operation status.

These bits are defined as follows:

000 = no results
001 = underflow
010 = overflow
011 = divide-by-zero
100 = DMA parity error (AFPA only)
101 = invalid operation
110 = inexact result
111 = illegal command.

Bit 0 is the master control bit for floating-point status. If bit 0 equals zero, no machine communications interrupts will be generated as the result of floating-point operations.

If bit 0 equals one, the individual control bits (3, 5, 7, 9, and 26) determine whether a given condition causes a machine communications interrupt. When both bit 0 and an individual control bit are set, an interrupt is sent to the virtual machine that made the floating-point request. Bit 0 of the FPSR is then set to zero.

For example, if bit 0 equals one, bit 5 equals zero, and bit 7 equals one, an overflow condition causes a machine communications interrupt (because bit 0 and 7 are both equal to one), but a divide-by-zero does not (because bit 5 is not equal to one). If a divide-by-zero occurs when bit 5 equals zero, status register bits 1 and 4 are set to one. In addition, the result register will contain the divide-by-zero result, a properly signed infinity.

The virtual machine accessing the hardware floating-point status register must use care in reading from or writing to this register. The following conventions must be honored:

- The status register should be read only with the **Read Status Register** command.
- The status register should be updated only with the **Write Status Register** command.
- Reserved fields in the floating-point status register are not protected. When writing to the reserved fields, those values must not be changed. Because you must write the entire register at the same time, you should determine the current register value with the **Read Status Register** command. The current value of certain fields may then be modified, but the reserved values should be restored to their original value.

Not-a-Numbers (NaN)

The IEEE draft defines any value with a maximum exponent and a non-zero fraction as not-a-number (NaN). The VRM emulation code uses the high-order fraction bit to distinguish quiet from signalling NaNs. When generated, all NaNs are quiet (high-order bit set to one). To change a NaN to a signalling NaN, set the high-order bit to one.

Interrupts

The VRM reports virtual interrupts to the virtual machine for the following reasons:

- Floating-point errors
- Floating-point exceptions.

These interrupt types are described in the following sections.

Floating-Point Errors

Floating-point errors occur when the hardware cannot perform the operation requested. This can happen in the following situations:

- When the Floating-Point Accelerator is not present or is locked
- When the operation is a privileged Floating-Point Accelerator command
- When the access type is illegal (not a **load** or **store** instruction)
- When the VRM emulation code is not present (see "Floating-Point Operations without Emulation Code" on page 7-22).

When a floating-point error occurs, the virtual machine receives a program check interrupt with bit 22 of the program check PSB status word set. Data word 1 of the program check PSB contains the Floating-Point Accelerator status. The program check type in the Floating-Point Accelerator status determines whether or not the old IAR is the address of the instruction causing the error.

The FPA, AFPA, and MC68881 use the address space from 0xFC000000 through 0xFFFFFFFF for its operations. If neither the FPA, AFPA, nor MC68881 is present, attempts to access this address space with **load** or **store** instructions will fail, like any other reference to an invalid address. The virtual machine will receive a program check error with bits 24 and 30 set in the status word of the program check PSB.

Floating-Point Exceptions

The five types of floating-point exceptions defined by the IEEE standard are:

- Divide-by-zero
- Overflow
- Underflow
- Inexact results
- Invalid operations.

The IEEE standard also defines two classes of exceptions that indicate the level of exception execution required by the virtual machine.

For trap-disabled exceptions, the VRM emulation code writes the IEEE default result to the destination register. The virtual machine is not interrupted when the exception occurs, and can determine that an exception has occurred only by reading the FPSR's bits 1, 2, 4, 6, 8, and 25.

Trap-enabled exceptions allow the virtual machine to override the IEEE values for the specific exceptions. For trap-enabled exceptions, the virtual machine is notified of the exception by way of a machine communication interrupt, with bit 22 of the machine communication PSB status word set.

When the MC interrupt is received by the virtual machine, the IAR does not point to the instruction that caused the exception, but to the next instruction to the floating-point hardware. The command to the floating-point hardware that caused the exception is available in data word 1 of the returned PSB.

Although machine communication interrupts may be masked, masking these interrupts while the FPA or AFPA is in use will result in a program check to the virtual machine if the VRM cannot issue a machine communication interrupt to the virtual machine.

When the VRM issues a machine communication interrupt to the virtual machine, the status and data words of the returned PSB (shown in Figure 2-4 on page 2-17) are defined as follows:

- Status word

  The status word will have bit 22 set equal to one and bit 24 set equal to zero to reflect the floating-point exception.
- Data word 1

  Data word 1 (shown in Figure 7-1) contains information on the operation that was being performed when the exception occurred, source and destination register information, and information on the type of exception that occurred.

0	8	10	16	17	18	24	29 31
Operation	*	Source Register	DL	DI	Dest. Register	Exception Flags	Trapped Exception

* Reserved

Figure 7-1. PSB Data Word 1 for Floating-Point Exceptions

The fields in data word 1 of a PSB returned due to a floating-point exception are defined as follows:

- Bits 0 - 7

  Bits 0 through 7 contain a value that indicates the operation being performed at the time of the exception. Where two values are shown, the first value indicates register-to-register operations and the second value indicates immediate-to-register operations. Possible values are:

Value	Operation
2	Conversion of an integer source to a single-precision destination
3	Conversion of an integer source to a double-precision destination
8,9	Conversion of a single-precision source to a double-precision destination
10,11	Conversion of a double-precision source to a single-precision destination
12,13	Negation of a single-precision source to a single-precision destination
14,15	Negation of a double-precision source to a double-precision destination

16,17	Absolute value of a single-precision source to a single-precision destination
18,19	Absolute value of a double-precision source to a double-precision destination
20,21	Extraction of the integer part from a single-precision source to a single-precision destination
22,23	Extraction of the integer part from a double-precision source to a double-precision destination
24,25	Round of a single-precision source to an integer destination
26,27	Round of a double-precision source to an integer destination
28,29	Truncation of a single-precision source to an integer destination
30,31	Truncation of a double-precision source to an integer destination
32,33	Floor of a single-precision source to an integer destination
34,35	Floor of a double-precision source to an integer destination
36,37	Comparison of the single-precision values in source and destination
38,39	Comparison of the double-precision values in source and destination
40,41	Single-precision addition of a source and a destination to a single-precision destination
42,43	Double-precision addition of a source and a destination to a double-precision destination
44,45	Single-precision subtraction of a source and a destination to a single-precision destination
46,47	Double-precision subtraction of a source and a destination to a double-precision destination
48,49	Single-precision multiplication of a source and a destination to a single-precision destination
50,51	Double-precision multiplication of a source and a destination to a double-precision destination
52,53	Single-precision division of a source and a destination to a single-precision destination
54,55	Double-precision division of a source and a destination to a double-precision destination
56,57	Single-precision IEEE remainder of a source and a destination to a single-precision destination
58,59	Double-precision IEEE remainder of a source and a destination to a double-precision destination
60,61	Square root of a single-precision source to a single-precision destination
62,63	Square root of a double-precision source to a double-precision destination
87,88	Sine of a double-precision source to a double-precision destination
89,90	Cosine of a double-precision source to a double-precision destination
91,92	Tangent of a double-precision source to a double-precision destination
93,94	e^x from a double-precision source to a double-precision destination
95,96	Arctangent of a double-precision source to a double-precision destination
97,98	Arccosine of a double-precision source to a double-precision destination
99,100	Arcsine of a double-precision source to a double-precision destination
101,102	Natural log of a double-precision source to a double-precision destination
103,104	Log (base 10) of a double-precision source to a double-precision destination
105,106	IEEE logb of a double-precision source to a double-precision destination

107,108	Double-precision arctangent of a source and a destination to a double-precision destination
109	Scale the exponent of a double-precision source to a double-precision destination
110,111	Single-precision modulo of a source and a destination to a single-precision destination
112,113	Double-precision modulo of a source and destination to a double-precision destination

- Bits 8 and 9 are reserved.
- Bits 10 - 15 indicate the operand 1 register.
- Bit 16 – Destination location.

  This bit indicates (when set) that data word 3 does not contain the low-order word of the designated result. Instead, data word 3 contains the storage location where the designated result may be placed by the virtual machine's trap handler. This bit can be set only when the exception occurred on a move from the MC68881 to memory.
- Bit 17 – Destination invalid.

  This bit indicates (when set) that the destination field in data word 1 contains the upper three bits of the low-order word of the designated result (left justified). The remaining bits of the low-order word will always be zeroes. Data word 2 contains the high-order word as usual. This bit can be set only when the exception occurred on a move from the MC68881 to memory for single-precision values.
- Bits 18 - 23 indicate the operand 2 register.
- Bits 24 - 28 – Exception flags.

  These bits indicate all exceptions that occurred on the operation. The following exceptions are indicated when the associated bit is set to one:

  Bit 24 indicates an inexact occurred.
  Bit 25 indicates a divide-by-zero occurred.
  Bit 26 indicates an underflow occurred.
  Bit 27 indicates an overflow occurred.
  Bit 28 indicates an invalid operation occurred.
- Bits 29 -31 indicate the particular exception being trapped. The following binary values are defined for this field:

  000 = invalid
  001 = overflow
  010 = underflow
  011 = divide-by-zero
  100 = inexact
  111 = illegal command

- Data words 2 and 3

  Typically, data word 2 contains the designated result of single-precision operations, if any. For double-precision operations, data word 2 contains the 32 most-significant bits of the result, if any. Data word 3 typically contains the 32 least-significant bits of a double-precision result, if any.

  Exceptions that occur on floating-point commands (except commands that involve conversion, such as conversion of an integer source to a single-precision destination) will place the IEEE-designated result of floating-point operations in data words 2 and 3. For underflow and overflow exceptions, results will be scaled. For inexact exceptions, the designated result is rounded to the destination precision. For invalid and divide-by-zero exceptions, for which no IEEE-designated result is defined, data words 2 and 3 will be undefined.

  Exceptions that occur on conversion-type floating-point commands define data words 2 and 3 as follows:

  - Single- or double-precision conversion to integer

    When bit 16 of data word 1 is set, data word 3 contains the address in memory of the destination. For inexact exceptions, data word 2 contains the designated result in integer format.

  - Double-precision to single-precision conversion in memory

    For inexact exceptions, data word 2 contains the correctly-rounded designated result and data word 3 contains the destination address in memory. For overflow or underflow exceptions, data word 2 contains the high-order word of the designated result. The low-order word, rounded for single-precision, has only three significant bits. These bits are placed, left-justified, in the destination register field of data word 1. Data word 3 contains the destination address in memory.

  - Integer to single-precision conversion

    Data word 2 contains the single-precision designated result of this conversion.

  - Double-precision to single-precision conversion

    For overflow and underflow exceptions, data words 2 and 3 contain the double-precision designated result. For inexact exceptions, data word 2 contains the single-precision result. For invalid and divide-by-zero exceptions, data words 2 and 3 are not defined.

Floating-Point Operations without Emulation Code

The emulation code is a file in the VRM file system. When this file cannot be loaded at VRM initialization time because it has been deleted or has had its permission bits changed, the Floating-Point Accelerator is still usable (if it is present). The virtual machine interface, however, is different.

In the absence of emulation code, all Floating-Point Accelerator exceptions are passed directly to the virtual machine as floating-point errors.

To determine whether the emulation code is present, the virtual machine can perform an operation which would normally be handled by the emulation code (such as addition with a denormal number) and determine whether or not a floating-point error interrupt results. If a floating-point error occurs, the virtual machine must determine whether or not to use the Floating-Point Accelerator.

This is generally not a problem because the emulation code is loaded by default if the VRM file system has not been modified.

Chapter 8. Virtual Resource Manager Debugger

CONTENTS

About this Chapter 8-5
Debugging Code in the VRM 8-6
Debugger Programming Interfaces 8-6
  Loading and Starting the Debugger 8-7
  Setting Breakpoints 8-8
Rules for Entering Commands 8-9
  Entering Numeric Values and Strings 8-9
  Specifying Expressions 8-10
  Debugger Variables 8-10
  Pointer References 8-12
Debugger Commands 8-13
  Alter — Alter memory 8-15
  AScii — Show ASCII representation of memory 8-16
  Back — Decrease the instruction address register (IAR) 8-17
  BReak — Set a breakpoint 8-18
  BREAKS — List the current breakpoints 8-19
  Clear — Clear one or all breakpoints 8-20
  CTldsp — Display Formatted VRM Control Block 8-21
  Display — Display a specified amount of memory 8-47
  DItto — Re-execute the last command 8-48
  Ebcdic — Display the EBCDIC representation of memory 8-49
  Find — Search storage 8-50
  Go — Start executing the program under test 8-53
  Ior — Perform an IOR instruction 8-54
  IOW — Perform an IOW instruction 8-55
  Loop — Start the debugger from this point 8-56
  Map — Display a module map 8-57
  Next — Increase the IAR 8-59
  Origin — Set the address origin 8-60
  Quit — Terminate the debugger session 8-61
  Reset — Clear a user-defined variable 8-62
  RESTore — Set the restore mode on or off 8-63
  Screen — Display a Screen of Data 8-64
  SEt — Initialize a variable 8-66
  SHow — Show the screen that was up (PC monochrome only) 8-67
  SRegs — Display segment registers 8-68
  ST — Store a fullword into memory 8-71
  STArt — Start executing the program under test 8-72
  STC — Store one byte into memory 8-73
  STEp — Execute instructions single-step 8-74
  STH — Store a halfword into memory 8-75
  STOp — Set a breakpoint 8-76
  STOPS — List the current breakpoints 8-77
  SWap — Switch to or from an RS-232 port 8-78

Tlb — Display the translate lookaside buffer 8-79
TOuch — Page-in the referenced memory 8-81
TRace — Display formatted trace table entries 8-82
Vars — Display a list of the user-defined variables 8-84
Xlate — Display the real address 8-85

About this Chapter

This chapter describes the VRM debugger, which is used primarily to locate and diagnose errors in VRM code. The description includes techniques for loading and invoking the debugger, as well as the commands used in the debugging process. Refer to *IBM RT PC Messages Reference* for an explanation of messages you may encounter while using the debugger.

Debugging Code in the VRM

The debugger helps to determine errors in code running in the VRM. The primary application of this component is debugging device drivers installed in the VRM with the **Define Code SVC**.

The debugger can run in any configuration that includes the VRM nucleus, a keyboard, and most IBM-supported RT PC displays or asynchronous terminals connected to a serial/parallel adapter. The VRM debugger does not support any 5080 displays or displays attached to the IBM PC Enhanced Graphics Adapter. For processing keystrokes, the debugger uses the keyboard translation table found in IOCN 0xC2.

When the debugger is started, it uses the entire display screen with no virtual terminal functions.

Debugger function includes commands to:

- Display and change processor memory and registers
- Display and change memory manager segment registers
- Display the memory manager translate lookaside buffer (TLB)
- Display I/O memory
- Display real address
- Display a system map
- Set breakpoints
- Execute instructions single-step
- Perform looping
- Page-in memory (if the machine state allows)
- Interpret data in ASCII or EBCDIC in addition to hexadecimal
- Search for a string (hexadecimal or ASCII)
- Display formatted trace table
- Set and use debug variables.

Debugger Programming Interfaces

This section explains the following:

- How to load and start the debugger
  - The **Debug SVC**
  - The **Cntl-Alt-Pad4** key sequence
- How to set breakpoints (debugger breakpoints or static debugger traps).

Loading and Starting the Debugger

The debugger must be loaded before it can be started. When started, the debugger accepts the commands described in "Debugger Commands" on page 8-13.

Because the debugger requires at least 80K bytes of pinned memory, it is loaded only when you request it. To load the debugger, enter the **Cntl-Alt-Pad4** key sequence or execute the **Debug SVC**.

The debugger checks for a supported display the first time it is invoked. The displays listed below are the only supported displays. They are searched for in the following order:

- Advanced Monochrome Graphics Display (APA-8)
- Advanced Color Graphics Display (APA-8 Color)
- Extended Monochrome Graphics Display (APA-16)
- IBM Personal Computer Display (PC monochrome)
- An asynchronous terminal (such as an IBM 3161 ASCII Display Station).

If a supported display is not found, the debugger returns and normal processing continues. Note that static debugger traps encountered when the debugger is not loaded cause program checks.

For asynchronous terminals, the debugger sets request to send (RTS) and data terminal ready (DTR) as part of its initialization process. The terminal must respond with dataset ready (DSR) within 500 milliseconds or the debugger will return. For more information on asynchronous terminals, see "SWap — Switch to or from an RS-232 port" on page 8-78.

You may want to load and initialize the debugger before the VRM dispatches any virtual machines. Use the **mvmd** command to change the permission bits of the debugger file **/vrm/ldlist/vrmbase/vdebug.0020.01** to a value of 440 (octal).

See *AIX Operating System Commands Reference* for the definition of the **mvmd** command.

After changing the permissions, the debugger will be loaded at the next system IPL.

When the debugger is brought up in this way, the debugger's internal keyboard translation table is used instead of the table at IOCN 0xC2. The debugger's internal table supports only the United States keyboard.

The **Debug SVC** and the **Cntl-Alt-Pad4** key sequence both load and start the debugger. Once loaded, you can check for errors in code with debugger commands, static debugger traps, or debugger breakpoints.

Also, if the debugger is loaded and the VRM should happen to abend, the debugger is automatically started. Message 032-024 displays with an abend code indicating the cause of the failure. See *IBM RT PC Messages Reference* for a listing and description of all messages.

If the VRM abends and the debugger is not loaded, VRM error logging facilities such as VRM dump handle the abend.

"Debugger Commands" on page 8-13 defines valid debugger commands. "Setting Breakpoints" on page 8-8 defines static debugger traps and debugger breakpoints.

The following discussion describes the **Debug SVC** and the **Cntl-Alt-Pad4** key sequence. Both of these methods load and invoke the debugger.

The Debug SVC

A virtual machine can issue the **Debug SVC** to give the debugger control when it, or any other virtual machine, is next dispatched.

The **Debug SVC** has an SVC code of 0xFFB6 and is defined as follows:

Calling register conventions:

GPR2 — contains the VMID of the virtual machine to be debugged.

Return codes: contained in GPR2

0 = Successful completion
16 = The virtual machine does not exist.

Programming Notes

- The debugger gets control when the virtual machine to be debugged is next dispatched. Therefore, a virtual machine that has issued a **VM Wait SVC** cannot start the debugger until it gets a virtual interrupt.
- To start the debugger at virtual machine IPL time, the machine should execute a **Debug SVC**. The **Debug SVC** may be used only if the virtual machine has been IPLed.
- If the debugger is not already loaded, execution of the **Debug SVC** loads it. The virtual machine issuing the SVC waits until the load is complete.

The Cntl-Alt-Pad4 Key Sequence

When you simultaneously press the control, alternate (the left alternate key), and number 4 numeric pad keys, you load and start the debugger.

This keystroke combination sends a non-maskable interrupt to the processor. The VRM, which fields such key combinations, immediately starts the debugger.

Setting Breakpoints

You can use two methods to explicitly start a loaded debugger. One method places static debugger traps (SDT) in predefined points in a program. The other method uses the debugger commands **BReak** or **STOp** to specify an address at which to start the debugger.

You can place SDTs in a program, such as a device driver, to start the debugger.

An SDT consists of the processor instruction **tgte 1,1**, which has a runtime value of 0xBD11. When the VRM encounters an SDT, a program check interrupt occurs. The VRM recognizes the SDT and

gives control to the debugger. If the debugger is not loaded when the VRM sees an SDT, the SDT is treated as any other trap instruction.

After the debugger is started, SDTs are treated the same as other processor instructions. The **STEp** command can be used to step over SDTs. The **Go** or **Loop** commands can be used to resume execution at the instruction following the SDT.

A debugger breakpoint is a spot in memory that causes control to pass to the debugger.

Use the **BReak** or the **STOp** command to set a breakpoint. (See "Debugger Commands" on page 8-13.)

Breakpoints do not show up in code that the debugger displays. The instruction overlaid by a breakpoint executes when you issue a **STEp** or a **Go** command. You can set breakpoints only in code that resides in real memory.

Rules for Entering Commands

The debugger must be loaded and started before it can accept commands.

The debugger recognizes commands of the form:

```
command parm1 parm2 ...
```

The debugger handles commands alphabetically, so only those letters necessary to make each command unique are required. For example, the **Back** command is entered as **B**. The **BReak** command must have a separate designation, in this case **BR**. Note that throughout this section, the minimum number of letters necessary to differentiate between commands are capitalized.

Commands and parameters can be entered in either uppercase or lowercase, or any combination of the two. Case is significant only when searching storage for a value specified by a quoted string.

Entering Numeric Values and Strings

Numbers can be decimal or hexadecimal, depending on the command. Decimal numbers must either be decimal constants (0-9), variables, or expressions involving the two.

Hexadecimal numbers may also include the letters A-F.

In some cases, only numeric constants are allowed. Where true, this restriction is clearly identified.

A string, on the other hand, is either a hexadecimal constant or a character constant of the form 'cccc...'. Single quotes separate the string from other data. To specify a single quote in a string, use two of them together. For example, to enter the string 'cccc'ccc', enter 'cccc''ccc'.

Specifying Expressions

The debugger does not allow full expression processing. Expressions may contain the following terms:

- Decimal or hexadecimal constants
- Variables
- Pointer references (see "Pointer References" on page 8-12).

Valid operators include:

- + (addition)
- − (subtraction)
- * (multiplication)
- / (division).
- : (segment ID)

  This operator indicates that a value is to be taken as a segment ID and adjusted so it will be a valid value for a segment register. The original value is shifted left two bit positions and 0x10000 is added to it. The following example indicates how to set segment register 5 to reference memory in segment 0x840:

  ```
  SEt s5 840:
  ```

  In this case, 840 is shifted left two bit positions and becomes 0x2100. The value 0x10000 is then added to 0x2100 to make 0x12100. This value is placed in segment register 5. This operation could also have been performed with `840*4+10000`.

Expressions are processed from left to right only. The type of data specified, decimal or hexadecimal, must be the same for all terms in the expression.

Debugger Variables

Variables may be up to eight alphanumeric characters (not including symbols and numeric constants) in length. They usually represent locations or values which are used repeatedly. Be careful when defining variables, since some character combinations represent reserved variables. A variable must not represent a valid number.

The reserved variables are shown in the following list.

COU Timer Counter
COUS Timer Counter Source
CS Condition Status
ECR Exception control register
IAR Instruction Address Register (program counter)
ICS Interrupt Control Status
IRB Interrupt Request Buffer
MCS Machine Check Status

MQ Multiply Quotient Register
PCS Program Check Status
Rn General purpose registers 0 through 15
Sn Segment registers 0 through 15
TS Timer Status.

These variables may be referenced and changed, but they represent specific hardware registers that govern program execution. Do not use them as work variables.

The following variables are defined to the debugger. However, because they have a special meaning to the debugger, their use may yield unpredictable results.

FX A variable set by the **Find** command
R Means real when used with an address
V Means virtual when used with an address
ORG A variable set by the **Origin** command

The **SEt** command can both define and initialize a variable. Variables may contain from 1 to 4 bytes of numeric data, or up to 32 characters of string data. For example:

```
SEt VAR1 12FE
SEt r2 VAR1 + 8
SEt S1 5*4+10000
```

Note that r2 and S1 are initialized each time you start the debugger, as are all reserved variables except FX.

If you change a segment register or general purpose register when using the debugger, the change remains in effect when the program resumes control. Keep this in mind and be careful when changing registers, especially segment registers.

The value placed in the segment register must conform to the following conventions:

Bits 0-14: Unused

Bit 15: Segment present bit

1 = Segment present
0 = Segment not present

Bit 16: Segment access to I/O

1 = Segment inaccessible to input/output operations
0 = Segment accessible to input/output operations

Bit 17: Segment access to the 32-bit processor

1 = Segment inaccessible to the 32-bit processor
0 = Segment accessible to the 32-bit processor.

Bits 18-29: 12-bit segment identifier

Bit 30: Special bit

Bit 31: Key bit.

For example, if you wish to put segment ID 12 into segment register 1, you enter:

```
Set S1 10048
Set S1 12*4+10000
```

You can release a variable with the **Reset** command. **Reset** cannot be used with reserved variables.

Pointer References

A pointer reference indirectly refers to a memory location. For example, assume location 0xC50 contains the address of a control block. A pointer reference can be used to access the control block.

Pointer references are specified as:

a > l

where:

a = A hexadecimal constant, variable, R0 through R15, or an expression involving the above terms.
> = The > is used to indicate that this is a pointer reference.
l = The length or number of bytes referenced. The default is 4.

If used in an expression, a pointer reference consists of all terms to that point in the expression. In other words, the **a** term consists of everything to the left of the **>**, including other pointer references. Consider the following example:

The variable ADDR1 contains 0xC50, and the variable DISP1 contains 8.

The expression 0x250 + DISP1 + ADDR1 > 2 + 9 is evaluated as follows:

1. 0x250 is added to 8 yielding 0x258.
2. 0x258 is added to ADDR1 yielding 0xEA8.
3. The two bytes at address 0xEA8 are added to 9.

Note that all the terms in a pointer reference are evaluated in hexadecimal.

The following example shows how to use a pointer reference on an Alter statement. (See "Alter — Alter memory" on page 8-15.)

1. Alter 0x124 > 0582
2. Alter ADDR1 > +8 D96E

In the first case, the data is put into the memory location pointed to by the word at address 124. The second case places 0xD96E into the address computed by adding 8 to the 4 bytes pointed to by ADDR1. Note that here a length could have been used; the default (4) was used.

Debugger Commands

The VRM debugger recognizes only a limited set of commands. The valid debugger commands are defined on the following pages. Each definition includes a general description of the command, the command syntax, and, when applicable, the execution sequence and results.

Several command descriptions include sample figures of data or memory that will be displayed when you enter certain parameters. Please note that these figures are provided for your information only and may not match data or memory on your system. Also, several figures have explanatory notes indicated by superscripted numerals beside the field. These numerals (which do not appear on screens you will see) refer you to a 'Figure Notes' section following the figure. This section defines the field on the screen.

If an error is encountered when attempting to execute a command, the command is rejected. If an invalid command is entered, error message 032-001 is returned.

In addition to the debugger commands, you have two services available for use with the debugger. The print service allows you to print a debugger screen (commands or memory) as it appears on the display device. The printer in use must be a PC printer.

To use this service, press the **Print** key. The contents of the currently-displayed screen will print for non-ASCII terminals if you have a PC-type printer. The print service does not work with ASCII terminals.

The other service allows you to display a list of the valid debugger commands. This list includes a brief description of each command.

To display this list of commands, enter **help** or a question mark (?) after the > prompt. Figure 8-1 on page 8-14 shows the resulting screen.

? or Help - Display the list of valid commands.
' ' (or nothing) - Repeat the last command.
Alter- Alter memory.
AScii - Interpret displayed memory in ASCII (default).
Back - Decrement the IAR.
BReak - Set a breakpoint.
BREAKS or STOPS- List currently set breakpoints.
Clear - Clear breakpoint(s).
CTldsp - Formatted VRM control block display.
Display - Display a specified amount of memory.
Ebcdic - Interpret displayed memory in EBCDIC.
Find - Find a string in memory.
Go - Start executing the program.
Ior - Execute an IOR instruction.
IOW - Execute an IOW instruction.
Loop - Execute until control returns to this point.
Map - Display a system module map.
Next - Increment the IAR.
Origin - Set the origin.
Quit - End a debugger session.
Reset - Release a user-defined variable.
RESTore - Restore or do not restore the screen.
Screen - Display a screen containing registers and memory.
SEt - Define and/or set a variable.
SHow - Show the screen if a PC monochrome display is in use.
SRegs - Display segment registers.
ST - Store a fullword into memory.
STArt - See Go.
STC - Store one byte into memory.
STEp - Perform an instruction single-step.
STH - Store a halfword into memory.
STOp - See BReak.
SWap - Switch from the current display/keyboard to an RS-232 port.
Tlb - Display the translate lookaside buffer (TLB).
TOuch - Touch, page-in, the referenced memory.
TRace - Display formatted trace information.
Vars - Display a listing of the user-defined variables.
Xlate - Display the real address of a memory location.

Figure 8-1. Screen produced by the Help command

Alter — Alter memory

Description: The **Alter** command changes a specified memory location to the hexadecimal value entered. **Alter** can be used to change one or several bytes of memory.

Syntax: **Alter** is specified as follows:

```
Alter address data
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. You can omit leading zeroes.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

Data is defined as follows:

Data must be specified as a hexadecimal constant. You cannot use variables or expressions. If you want to modify storage to the value of a variable or expression, use the ST, STC, or STH commands. You can break the constant into a maximum of 5 sections with imbedded blanks. Each section can be up to 32 digits (16 bytes) in length. The number of bytes modified depends on the number of bytes entered. If you enter an odd number of bytes, only the first four bits of the last byte are changed.

Errors: See messages 032-002, 032-003, and 032-004.

AScii – Show ASCII representation of memory

Description: The **AScii** command determines whether memory is displayed in ASCII. **AScii** ON displays the ASCII representation of the memory. See Figure 8-29 on page 8-65. Memory appears in hexadecimal regardless of the **AScii** setting.

Syntax: **AScii** is specified as follows:

```
AScii On or Off
```

The **On** parameter turns the ASCII display on. This is the default setting.

The **Off** parameter turns the ASCII display off.

Execution sequence: The new setting takes effect the next time you display memory.

Errors: See message 032-002.

Back — Decrease the instruction address register (IAR)

Description: The **Back** command decreases the IAR value by the specified value.

Syntax: The **Back** command is defined as follows:

```
Back n
```

The **n** parameter requires a valid, hexadecimal number. If you omit the **n** parameter, the default is 4.

Errors: See message 032-002.

BReak — Set a breakpoint

Description: The **BReak** command sets a breakpoint in a program. A breakpoint starts the loaded debugger when the instruction at the specified address is to be executed. You can also use **BReak** to set a trace point. Trace points place an entry into the trace table.

When a breakpoint is encountered, the debugger displays the data shown in Figure 8-29 on page 8-65 (unless you specify **Screen Off.**). Message 032-020 or 032-021 also displays, depending on the source of the breakpoint.

Syntax: The **BReak** command is specified as follows:

```
BReak address T
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeroes can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

T is an optional value that specifies a trace point.

Execution sequence: The debugger is started by a breakpoint trap at the point where the instruction at that address would normally be executed. If the breakpoint is a trace point, the trace point is entered in the VRM trace table, and execution continues without operator intervention.

If the breakpoint is not a trace point, you control the debugger through the keyboard. A **STEp**, **Go**, or **Loop** command will execute the instruction and, except for **STEp**, continue.

Execution results: A breakpoint trap, 0xBD00, is placed at the address specified and is added to the debugger's table of breakpoints.

Errors: See messages 032-002, 032-004, and 032-006.

BREAKS – List the current breakpoints

Description: The **BREAKS** command displays a list of the breakpoints currently active. See Figure 8-2 for an example of this display.

Syntax: The **BREAKS** command is specified as follows:

```
BREAKS
```

The **BREAKS** command has no parameters.

Errors: None.

The following figure is displayed when you request the **BREAKS** command.

```
xxxxxxxx B    xxxxxxxx T    xxxxxxxx B    xxxxxxxx B
```

Figure 8-2. Breakpoint Display

Figure notes:

xxxxxxxx is the breakpoint address.

B or T indicates a break or trace point, respectively.

Clear — Clear one or all breakpoints

Description: The **Clear** command allows you to clear one or all of the breakpoints.

Syntax: **Clear** is specified as follows:

```
Clear
Clear address
Clear *
Clear *  T or B
```

If you specify no parameters (**Clear**), the breakpoint pointed to by the IAR is cleared.

Address is defined as follows:

xxxxxxxx x must be a valid hexadecimal digit. Leading zeroes can be omitted.

R xxxxxxxx R implies a real address and is the default if mode = Real.

V xxxxxxxx V implies a virtual address and is the default if mode = Virtual.

+xxxxxxxx + means that the value is a displacement relative to the setting of the Origin.

Clear * means clear all trace points and breakpoints.

Clear * followed by **T** or **B** means clear all trace points or breakpoints, respectively.

Execution results: The debugger is no longer started at the specified trace or breakpoints.

Errors: See messages 032-002, and 032-008.

CTldsp — Display Formatted VRM Control Block

Description: The **CTldsp** command produces formatted VRM control block displays. The control block displays provide information on VRM components such as processes, SLIHs, queues, devices, and so on. This command reflects VRM status and has little to do with hardware. Therefore, you are expected to have a relatively thorough understanding of the concepts described in the rest of *IBM RT PC Virtual Resource Manager Technical Reference* before executing this command.

Note: Illustrations of control blocks are provided as an aid to debugging. These illustrations are not intended to represent the exact format of any particular structure.

Syntax: **CTldsp** is specified as follows:

```
CTldsp
```

This command has no parameters, but is menu-driven.

Errors: None.

The Selection Menu

The **CTldsp** command is driven from a menu that is displayed when you enter the command. The **CTldsp** menu is shown in Figure 8-3 on page 8-22. Each menu option is explained in the following sections. Most of the selections allow you to display all control blocks of a given type.

When you see the prompt

```
Press ENTER or X to exit
```

you can return to the main selection menu by entering X. If no control blocks exist for the menu option you choose, the message None are present is displayed.

The following symbols have special meanings to this command and are defined below.

#	A hexadecimal number.
#d	A decimal number.
@@@@@@@@	An address.
$$$$	A name (4 ASCII characters).
ttxx####	This field is a 32-bit ID of the following format: tt = type xx = generation number #### = block index.

The following figure shows the screen that appears when you request the **CTldsp** command.

```
Current ID: ttxx#### Name: $$$$ IAR: @@@@@@@@
Control block display, select type:
 1 = Semaphores
 2 = Timer requests
 3 = Processes
 4 = SLIHs
 5 = Modules
 6 = Devices
 7 = Select block by index
 8 = Utilization information
 9 = Queues
10 = Paths
11 = Timer information
12 = Virtual machines
13 = Segment information
14 = Virtual page information
 X = End
Enter your selection.
>
```

Figure 8-3. CTldsp Command Selection Menu

Each option is defined on the following pages.

Option 1 — Semaphore Control Blocks

If option 1 is selected and one or more semaphores exist in the system, the screen shown in Figure 8-4 is displayed.

```
Semaphore:
ID:      ttxx####
Address: @@@@@@@@
Name:    $$$$
Count:   #d                              (1)
Owner ID/Name:  ttxx####/$$$$            (1)
Nest Count:  #                           (1)
Nobody's waiting on this one             (2)
Waiting processes:                       (2)
ttxx####
ttxx####
 . . .
Press ENTER or X to exit.
```

Figure 8-4. Semaphore Control Block Display

Figure notes:

1. If the count is zero, one or both of the lines following the count line may be displayed.
2. If another component is waiting for a semaphore, the waiting IDs are displayed.

You may press **Enter** to see the next semaphore or return to the main selection menu if this was the last semaphore.

Option 2 — Timer Request Blocks

If you select option 2 for data on timer request blocks, the screen shown in Figure 8-5 is displayed.

If a timer request block (TRB) exists in the system, the TRB can be active or completed. If no TRBs of a given type exist, one or both of the following messages may be displayed:

```
No completed TRBs
No active TRBs
```

The following figure appears when you select **CTldsp** option 2.

```
Completed TRB:                                    (1)
Active TRB:                                       (1)
Requestor's ID/Name:  ttxx####/$$$$
Address:              @@@@@@@@
Mask:                 #
Current count:        #d
Original value:       #d
Press ENTER or X to exit.
```

Figure 8-5. Timer Request Block Display

Figure note:

1. A timer request block may be either completed or active.

Option 3 — Process Control Blocks

If you select option 3 for data on process control blocks, Figure 8-6 on page 8-26 is displayed.

```
Process Control Block:
ID:      ttxx####
Address: @@@@@@@@
PB @:    @@@@@@@@
Name:    $$$$
Flags:
  Cancel wait                                            (1)
  Timer signal received                                  (1)
  is a VM                                                (1)
  Excp Handler running                                   (1)
  has been signaled                                      (1)
  is in TIMER wait                                       (1)
  is in SEMAPHORE wait                                   (1)
  is in EVENT wait                                       (1)
  is in page wait                                        (1)
  Process is READY                                       (1)
  Executing VM instructions                              (1)
  VM WAIT state                                          (1)
  Running under PLB                                      (1)
  Terminating/Terminated                                 (1)
  Virtual stack space                                    (1)
  Debug mode                                             (1)
No exception handler.                                    (2)
Exception handler addr:  @@@@@@@@                        (2)
ECBs posted:
ECB Wait mask:
PB Chain:                @@@@@@@@
PB Priority chain:       @@@@@@@@
Priority:                #
VMCB address:            @@@@@@@@                        (3)
PLB address:             @@@@@@@@                        (3)
No queues are active.                                    (4)
Number of queues:  #d                                    (4)
Do you want to see them?                                 (4)
Do you want to see the segment registers?                (5)
Do you want to see the GPRs?                             (6)
Press ENTER or X to exit.
```

Figure 8-6. Process Control Block Display

Figure notes:

1. One or more of these lines may be displayed depending upon the setting of the flags.
2. If the process has an exception handler, the exception handler address is displayed.
3. These lines are displayed only if the process is a virtual machine.
4. If the process owns any queues, you may look at their control blocks by responding with a **y**. Figure 8-19 on page 8-39 shows the queue control block display.
5. When you request to see the segment register display, fields similar to those at the top part of Figure 8-30 on page 8-69 are displayed.
6. When you request to see the GPR display, fields similar to those at the top part of Figure 8-29 on page 8-65 appear. The values shown for the GPRs, ICS, CS, MQ, ECR (if APC is present), and IAR are significant. If the APC is present and restart data exists for an exception, you can optionally view a screen that indicates the location of the restart data, and the length, address, and data of the operation performed. Other values, such as that for the IRB, should be ignored.

Option 4 — SLIH Control Block Display

If you select option 4 for data on second-level interrupt handler (SLIH) control blocks, Figure 8-7 is displayed.

```
SLIH control block:
ID:           ttxx####
Address:      @@@@@@@@
DDS pointer: @@@@@@@@
SLIH Chain:  #
Name:         $$$$
No exception handler.                          (1)
Exception handler addr:  @@@@@@@@              (1)
No queues are active.                          (2)
Number of queues: #d                           (2)
Do you want to see them?                       (2)
>
Do you want to see the SLIH chain?             (3)
>
Press ENTER or X to exit.
>
```

Figure 8-7. SLIH Control Block Display

Figure notes:

1. If an exception handler exists, its address is displayed as shown.
2. Figure 8-19 on page 8-39 shows a queue display.
3. If you request to see the SLIH chain, Figure 8-8 on page 8-29 is displayed.

The following figure appears when you request to see the SLIH chain from Figure 8-7 on page 8-28.

```
SLIH entry chain display:
Address:         @@@@@@@@
Flags:           #
Poll length:     #
Priority:        #
Sublevel:        #
Int. Index:      #
Alias chain:     @@@@@@@@
Poll address:    @@@@@@@@
Off level addr:  @@@@@@@@
Off level TOC:   @@@@@@@@
SLIH address:    @@@@@@@@
TOC address:     @@@@@@@@
Press ENTER
```

Figure 8-8. SLIH Chain Display

Figure note:

- Upon pressing **Enter**, you see either the next entry on the chain or the last prompt in Figure 8-7 on page 8-28.

Option 5 — Module Control Block Display

If you select option 5 for data on module control blocks, Figure 8-9 is displayed.

```
Module Table entry:
Address:          @@@@@@@@
IOCN:             #
Module ID:        ttxx####
Use count:        #d
No. of devices:   #d
Module Type:
* resident                      (1)
* one-use                       (1)
* reuseable                     (1)
* copy                          (1)
* Duplicate                     (1)
Main entry point:  @@@@@@@@
TOC address:       @@@@@@@@
Start address:     @@@@@@@@
Module length:     #
Copy of      ttxx####           (2)
Copy Count:  #d                 (2)
Press ENTER or X to exit
```

Figure 8-9. Module Table Entry Display

Figure notes:

1. These fields represent module type flags.
2. Only one of these fields is displayed.

When you press **Enter**, you see the next entry or return to the main menu.

Option 6 — Device Control Blocks

If you select option 6 for data on device control blocks, Figure 8-10 is displayed. When you press **Enter** after viewing an entry, you continue through the table or return to the main menu.

```
Device Table entry:
Address:            @@@@@@@@
IODN:               #
Device ID:          ttxx####
Number of users:    #
Device type:
* driver                        (1)
* Internal                      (1)
* Shared                        (1)
DDS segment ID:     #
DDS offset:         #
Queue ID:           ttxx####
Module ID:          ttxx####
Adapter ID:         ttxx####
Press ENTER or X to exit.
```

Figure 8-10. Device Table Entry

Figure note:

1. These fields represent the type flags.

Option 7 — Select Block by Index

If you select option 7, you are prompted to enter an index value to display a specific control block. The index value must be in the range 0x0000 to 0xFFFF. The format of the control block may differ, depending on the type of control block you request. If you request an invalid type, the following figure is displayed.

```
Control block at index: #
Address:  @@@@@@@@
Data: #                (1)
Press ENTER to continue
```

Figure 8-11. Generic Control Block Display

Figure note:

1. This line is displayed eight times where # is a word of data.

When you select option 7, the following prompt displays:

```
Enter control block index in hex (or X to exit)
```

Type in the index, press **Enter**, and the specified control block is displayed. After viewing the control block, press **Enter** again to return to the above prompt. You may then enter another index or enter an **X** to return to the main menu. If you just press **ENTER**, the next sequential block is displayed.

If you specify an index for a queue element, the screen in Figure 8-12 on page 8-33 is displayed. This screen refers you to another screen, depending on the type of queue element you request.

```
Queue Element at index:   #
Address:    @@@@@@@@
Next ID:    #
Path ID:    ttxx####
Flags:      #
 *  *  *
Press ENTER
```

Figure 8-12. Prompt for Queue Element Selection

Figure note:

Depending on the type of queue element you want to see, Figure 8-13 on page 8-34, Figure 8-14 on page 8-35, Figure 8-15 on page 8-36, Figure 8-16 on page 8-37, or Figure 8-17 on page 8-37 may be displayed.

The following screen appears when you request to see a response queue element from Figure 8-12 on page 8-33.

```
Acknowledgement queue element:
Options:   #
Flags:     #
* Solicited response
* Unsolicited response
Overrun:   #
Results:   #
IODN:      #
PSB Data:  #
PSB Data:  #
PSB Data:  #
PSB Data:  #
```

Figure 8-13. Response Queue Element

Figure note:

Responses (*) can be either solicited or unsolicited.

The following screen appears when you request to see a send command queue element from Figure 8-12 on page 8-33.

```
Send Command Request:
Priority:    #
Options:     #
* Unconditional ack
* Ack on error
* Synchronous
* Chained CBs
* Control
IODN:        #
SegID:       #
GPR3 data:   #
GPR4 data:   #
GPR5 data:   #
GPR6 data:   #
```

Figure 8-14. Send Command Queue Element

Figure note:

The options field is broken down, and one or more of these lines (*) may be displayed.

The following screen appears when you request to see a start I/O queue element from Figure 8-12 on page 8-33.

```
Start I/O Request:
Priority:     #
Options:      #
* Unconditional ack
* Ack on error
* Synchronous
* Chained CBs
* Control
IODN:         #
SegID:        #
CCB address: @@@@@@@@
CCB length:   #
```

Figure 8-15. Start I/O Queue Element

Figure note:

The options field is broken down, and one or more of these lines (*) may be displayed.

The following screen appears when you request to see a general purpose queue element from Figure 8-12 on page 8-33.

```
General Purpose Request:
Priority:     #
Options:      #
* Unconditional ack
* Ack on error
* Synchronous
* Chained CBs
* Control
Data:         #
Data:         #
Data:         #
Data:         #
Data:         #
```

Figure 8-16. General Purpose Queue Element

Figure note:

The options field is broken down, and one or more of these lines (*) may be displayed.

The following screen appears when you request to see a detach queue element from Figure 8-12 on page 8-33.

```
Detach Request
```

Figure 8-17. Detach Queue Element

Option 8 — Utilization information

If you select option 8, Figure 8-18 provides information about system utilization of control blocks. This command can tell you the number of free control blocks, the number of queues in the system, and so on.

```
Utilization Information:
Unused Control Blocks:   #d
Used Control Blocks:     #d
Devices:                 #d
Processes:               #d
Queues:                  #d
Modules:                 #d
Semaphores:              #d
Paths:                   #d
Interrupt handlers:      #d
Segments:                #d
Pinned Pages:            #d
Timer Requests:          #d
Mini-disks:              #d
Virtual Machines:        #d
Queue Elements:          #d
Queue Extensions:        #d
Queue I/O Extensions:    #d
Path Extensions:         #d
SQA in use:              #
Free SQA:                #
Press ENTER
```

Figure 8-18. Utilization Information Display

Figure note:

The counts represent the number of control blocks which, in some cases, includes control block extensions.

Option 9 — Queue Control Blocks

If you select option 9 for data on queue control blocks, Figure 8-19 is displayed.

```
Queue Control Block:
ID:          ttxx####
Address:     @@@@@@@@
Maximum priority       #
ECB Mask:              #
Server's ID:           ttxx####
Unlimited no. of paths:                               (1)
Maximum no. of paths:  #d                             (1)
No Active Elements:   #                               (2)
Active Element ID:    #                               (2)
Level ## - Empty                                      (3)
Level ## - 1 element, index: #                        (3)
Level ## - #d elements, first index: #d               (3)
This isn't an I/O queue                               (4)
This is an I/O queue:                                 (4)
  I/O Initiation EP:  @@@@@@@@
  I/O Initiation TOC: @@@@@@@@
No Check Parameters routine                           (5)
Check Parameters EP:  @@@@@@@@                        (5)
Check Parameters TOC: @@@@@@@@                        (5)
Timer is not in use                                   (6)
Timer is active                                       (6)
  Timeout Mask:       #d
  Timeout Value:      #d
  Current Value:      #d
There are no paths                                    (7)
Do you want to display Path data?                     (7)
>
Press ENTER or X to exit.
```

Figure 8-19. Queue Control Block Display

Figure notes:

1. The path limit, if any, is displayed.
2. If the queue has an active element, its index is displayed. You may use option 7, and this index, to display the element.
3. For each priority level (##), the number of elements (or empty) and the index of the first element is displayed.
4. If the queue is an I/O queue, the entry points are displayed as shown.
5. If the control block has a check parameters routine, the entry points are displayed as shown.
6. If the timer is in use, timer values are displayed as shown.
7. If paths are present, you may respond with a Y to display path data, as shown in Figure 8-20 on page 8-41.

Option 10 — Path Control Blocks

If you select option 10 for data on path control blocks, Figure 8-20 is displayed.

```
Path Information:
Path ID:             ttxx####
Address:             @@@@@@@@
From ID:          ttxx####
To ID:            ttxx####
To Queue ID:      ttxx####
To Server:        ttxx####
From Server:      ttxx####
Path type: No Ack.                             (1)
Path type: Short Ack.                          (1)
Path type: Long Ack.                           (1)
Path type: Interrupt Ack.                      (1)
Path type: Invalid                             (1)
Ack. ECB:         ########                     (2)
Ack. Queue:       ttxx####                     (2)
Int. Level:       #                            (2)
  Sublevel:       #                            (2)
No device extension.                           (3)
Do you want to see the device extension?       (3)
>
Press ENTER or X to exit.
```

Figure 8-20. Path Information Display

Figure notes:

1. A path may be set up for one of the types shown.
2. The return mechanism for the acknowledgement is displayed here.
3. Figure 8-21 on page 8-42 describes the device extension display.

The following figure appears when you request to see the device extension from Figure 8-20.

```
Device Extension:
Flags:          #
Results:        #
IODN:           #
Data 1:         #
Data 2:         #
Data 3:         #
Press ENTER
```

Figure 8-21. Device Extension for Path Information Display

Figure note:

After you press **Enter**, you see the last prompt shown in Figure 8-20 on page 8-41.

Option 11 — Timer Information

If you select option 11 for timer information, the screen shown in Figure 8-22 gives timer status from when the debugger was started.

```
Timer Information:
Time of year:             #d
Time of IPL:              #d
Elapsed time from IPL:    #d
60 hz Counter:            #d
mm/dd/yy hh:mm:ss          (1)
Press ENTER
```

Figure 8-22. Timer Information Display

Figure note:

1. mm = month, dd = day, yy = year, hh = hour, mm = minute, ss = second

Option 12 — Virtual Machine Information

If you select option 12 for virtual machine information, the screen shown in Figure 8-23 is displayed.

If no virtual machines exist in the system when you make this request, you will receive a message indicating this situation.

```
VM Control Block:
Index:  #
Paths to system servicer:
* MCP                :ttxx####
* Timer SLIH         :ttxx####
Address of page 0    :#            (1)
Interrupt queue      :#
Timer source         :#d
VM Process ID        :ttxx####
VM Name              :$$$$
VM Number            :#d
IODN used for IPL    :#d
Press ENTER
>
```

Figure 8-23. Virtual Machine Display

Figure note:

1. The address of page 0 is a real address.

Option 13 — Segment Information

If you request option 13 to display segment information, Figure 8-24 is displayed.

Note that you can, from the main selection menu, enter option 13 with a specific segment ID. If you do so, information on only that segment will be displayed.

When you press **Enter**, you will see the next segment entry or return to the main selection menu.

```
Segment Information:
Segment Entry Addr: #
Segment Identifier: #
Segment Size:       #
Default Protection: #
Creating VMID:      #       (1)
Main List Start     #
Attach Count:       #       (1)
Inverted Segment            (1)
VMM SVCs not allowed        (1)
Page Faults not allowed     (1)
Press ENTER or X to exit.
```

Figure 8-24. Segment Information Display

Figure notes:

1. One or more of these lines may be displayed.

Option 14 — Virtual Page Information

If you select option 14 to display virtual page information, Figure 8-25 is displayed. Note that you can, from the main command selection menu, enter option 14 with a particular segment identifier. When you use this feature, the virtual page information starts with the first page in the segment you specified. You can also specify a virtual page number after the segment ID. If you do so, information on only that page in the specified segment is displayed.

```
Virtual Page Information:
Segment Identifier:   #
Address in the XPT:   @@@@@@@@   (1)
Shared XPT Address:   @@@@@@@@   (1)
Virtual Page Number:  #
Storage Protection:   #          (1,2)
Mapped to Minidisk:   #          (1,2)
Minidisk LBN:         #          (1,2)
Address in 1st IPT:   @@@@@@@@   (1)
Address in 2nd IPT:   @@@@@@@@   (1)
Page is all zeros                (1,2)
Page is mapped copy on write     (1,2)
Page is mapped to file system    (1,2,3)
Page is mapped to paging space   (1,2,3)
Page is System pinned            (1)
Page is a VM's page 0            (1)
VM pin count:         #          (1)
Press ENTER or X to exit
```

Figure 8-25. Virtual Page Information Display

Figure notes:

1. One or more of these lines may be displayed.
2. This line is displayed only if the control block is in memory.
3. Only one of these lines will be displayed.

Display — Display a specified amount of memory

Description: This command displays storage starting at the specified address. You indicate the number of bytes to display, and this figure is rounded up to the next multiple of 16. The display formats like the memory display for the **Screen** command (see Figure 8-29 on page 8-65), except that the address shown to the left is the exact address specified. If you specify a length of 1, 2, or 4 bytes, however, the **Display** command shows the exact amount of storage requested. When you specify a length of 1, 2, or 4 bytes, the debugger uses the processor machine instructions load character, load half, and load, respectively. These instructions are preferred when you want to display I/O address space (addresses of the form 0xFnnnnnnn).

Syntax: **Display** is specified as follows:

```
Display address length
```

Address is defined as follows:

xxxxxxxx x must be a valid hexadecimal digit. Leading zeroes can be omitted.

R xxxxxxxx R implies a real address and is the default if mode = Real.

V xxxxxxxx V implies a virtual address and is the default if mode = Virtual.

+xxxxxxxx + means that the value is a displacement relative to the setting of the Origin.

Length is a hexadecimal number. If the length attribute is omitted, 0x10 bytes are displayed.

Errors: See message 032-002.

DItto – Re-execute the last command

Description: The **DItto** command causes the most recently entered command to be re-executed. This technique is especially useful when stepping through a program or scrolling through memory.

Syntax: Press **Enter**, or type a string of blanks and press **Enter**.

Execution sequence: If **DItto** is the first command entered, the Help screen appears.

Note: Be careful when using the **DItto** command. If this command is entered on return to the debugger after executing a command that caused the debugger to give up control, the command will be re-executed (the debugger gives up control again).

Ebcdic — Display the EBCDIC representation of memory

Description: This command displays the EBCDIC representation of memory to the right of the hexadecimal representation. **Ebcdic** is normally off.

Syntax: **Ebcdic** is specified as follows:

```
Ebcdic On or Off
```

On indicates that the memory displays in EBCDIC.

Off indicates memory displays in ASCII. This is the default setting.

Execution sequence: **Ebcdic** takes effect with the next memory display.

Errors: See message 032-002.

Find — Search storage

Description: The **Find** command searches storage for an argument beginning at the address specified. If the argument is found, the search stops and the storage containing the argument is displayed. (See Figure 8-26 on page 8-52.) The address of the storage is placed into a variable called FX.

If the search passes over a page that is not in storage, message 032-012 is displayed. If ten consecutive pages are not in memory, message 032-013 is displayed. You must then indicate if you want to continue the search.

Syntax: **Find** is specified as follows:

```
Find argument address end-address alignment
```

Argument can be a hexadecimal number or a character string.

Address is defined as follows:

xxxxxxxx x must be a valid hexadecimal digit. Leading zeroes can be omitted.

R xxxxxxxx R implies a real address and is the default if mode = Real.

V xxxxxxxx V implies a virtual address and is the default if mode = Virtual.

+xxxxxxxx + means that the value is a displacement relative to the setting of the Origin.

End-address is specified the same way as **Address** and is the upper address limit.

Alignment provides the boundary alignment of the argument. This value must be 1, 2, or 4, indicating byte, halfword, or word alignment, respectively.

Execution Results: A screen containing the found data is displayed. If the data is not found, message 032-011 displays.

Errors: See messages 032-002, 032-004, 032-011, 032-012, and 032-013.

Error handling: The command is rejected if the parameters are invalid.

Comments:

- An asterisk (*) may be substituted for any of the parameters. An asterisk in any of the command fields causes **Find** to use the same value that was used for the previous **Find**.

For example, if you enter:

```
Find D96E 5782C 80000
```

and the argument (D96E) is found at address 6B000, you can enter:

```
Find * 6B001
```

to find the next occurrence of D96E. Note that you can also use FX + 1 (argument storage address plus 1) for 6B001.

- The argument length is determined as follows:
  - When a hexadecimal constant is the argument, the argument length is 1-4 bytes, depending on the length of the data entered. For example, each of the following has a 2-byte argument.

    ```
    Find 123
    Find 1234
    Find 012
    ```

    Find 123 and **Find 0123** mean the same thing.
  - When the argument is a character string, the number of characters in the string is the argument length.
  - If you enter the argument as an expression, the results are unpredictable.
- The following defaults apply to the first issuance of the **Find** command:
  - Address = 0
  - End-address =
    - The amount of real memory available if translation is off.
    - The start address ORed with 0x0FFFFFFF if translation is on.
  - Alignment = 1 (byte alignment).

The following example shows the result of a successful **Find** command.

```
Argument found at FX
@@@@@@@@  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
@@@@@@@@  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

Figure 8-26. Find Display (Argument Found)

Figure notes:

- FX appears as an address. This address goes into the variable FX.
- The first @@@@@@@@ is rounded down from FX to a 16-byte boundary. If in real mode, @@@@@@@@ will be displayed as R @@@@@@.

Go — Start executing the program under test

Description: The **Go** command is used to resume execution of your program. Execution begins at the current IAR setting, which is the point from which the debugger was started (unless the IAR was modified). You may specify an address to override the IAR setting. The **STart** command does the same thing.

Syntax: The **Go** command is specified as follows:

```
Go address
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeros can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

If the address is not specified, execution resumes at the current IAR setting.

Execution sequence: If a breakpoint is set at the specified address, the following occurs:

- The breakpoint trap, 0xBD00, is not placed into the code at that point.
- Traps are placed into the code as if a **STEp** were being executed.
- When the debugger is started from this point, the original 0xBD00 is placed into the code. This means that message 032-005 may occur when executing the **Go** command.

Errors: See messages 032-002, 032-004, and 032-005.

Ior — Perform an IOR instruction

Description: The **Ior** command allows you to perform the processor's **IOR** (Input/Output Read) instruction. The data at the port address specified is displayed. See Figure 8-27.

Syntax: The **Ior** command is specified as follows:

```
Ior port address
```

Port address must be a valid hexadecimal number.

Errors: See messages 032-002 and 032-003.

The following example shows the results of a successful **Ior** command.

```
dddddddd read from xxxxxxxx
```

Figure 8-27. Display from the Ior Command

Figure notes:

- dddddddd is the data.
- xxxxxxxx is the address.

IOW — Perform an IOW instruction

Description: The **IOW** command allows you to perform the processor's **IOW** (Input/Output Write) instruction. The data is written to the specified port address.

Syntax: **IOW** is specified as follows:

```
IOW port address data
```

Data and **port address** must be valid hexadecimal numbers.

Errors: See messages 032-002 and 032-003.

Loop – Start the debugger from this point

Description: The **Loop** command places a breakpoint at the current IAR address. All other breakpoints are ignored.

Syntax: **Loop** is specified as follows:

```
Loop n
```

n is the number of loops, in decimal, that execute before the debugger regains control. The default is 1.

Errors: See messages 032-002 and 032-005.

Map — Display a module map

Description: The **Map** command displays a map of the modules that are defined to the VRM. The map can be formatted beginning with the lowest module start address or with the lowest defined IOCN. In either case, the list is presented in ascending order.

Syntax: **Map** is specified as follows:

```
Map Address
Map IOCN
```

Address formats the map beginning with the lowest starting address of any VRM module. This is the default if no parameter is specified with the **Map** command.

Figure 8-28 shows a sample of a map specified with the **Address** parameter.

IOCN formats the map beginning with the lowest defined IOCN in the VRM. If the example shown in Figure 8-28 had been specified with the **IOCN** parameter, the entries would be presented beginning with IOCN 20, then 50, 81, 230, and so on.

Either of these methods can be used to get information on all the modules defined to the VRM by pressing **Enter** when the display screen fills with entries.

Errors: See messages 032-002 and 032-016.

```
IOCN  Mod. ID   Start     Entry pt.  Length    Uses  Devs  Copies  Attr
**************************************************************************
0050  04010014  00000000  01817800  00000000    0     0      0     P1.
0020  0402006A  01300000  01300060  00012000    1     0      0     .1.
0081  04010022  01312000  01312060  00002500    1     1      0     .1.
0230  0402004C  01314500  01314560  00000E00    2     2      0     .1.
 .      .          .         .          .       .     .      .      .
 .      .          .         .          .       .     .      .      .
 .      .          .         .          .       .     .      .      .
00A1  04010038  01320500  01320560  00001700    1     1      2     .R.
 .      .          .         .          .       .     .      .      .
 .      .          .         .          .       .     .      .      .
 .      .          .         .          .       .     .      .      .
00A1  040100A5  01339700  01320560  00001700    0     1     0038   .RC
```

Figure 8-28. Example of a System Module Map

Figure notes:

The Uses and Devices (Devs) columns indicate, in decimal, the respective number of uses or devices for the specified module.

The Copies column contains the index of the module copied (if position three of the attributes column is C). If position three is a period, the copies column indicates the number of copies (in decimal) of the module.

The attributes (Attr) column has three fields that describe the module. The values of these fields are defined as follows:

- Position 1 – indicates if this is a permanent module. A P in position 1 means the module is permanent and a period means the module is not permanent.
- Position 2 – indicates if the module is reusable or one-use only. An R means the module is reusable and a 1 means the module is one-use only.
- Position 3 – indicates if the module is a copy or an original. A C means the module is a copy and a period means the module is an original.

Next — Increase the IAR

Description: The **Next** command increases the IAR by the number specified.

Syntax: The **Next** command is specified as follows:

```
Next n
```

The **n** value must be a valid hexadecimal number. If a value for **n** is omitted, the default is 4.

Errors: See message 032-002.

Origin — Set the address origin

Description: The specified origin address is added to any hexadecimal expression beginning with a plus sign (+). The **Origin** command is especially useful when setting breakpoints. The value of the origin, and the origin displacement of the IAR, is displayed as shown in Figure 8-29 on page 8-65.

This command also sets the reserved variable **ORG**. The command **Origin** 652C0, for example, has the same value as **SEt** ORG 652C0.

Syntax: The **Origin** command is specified as follows:

```
Origin n
```

n is any hexadecimal value in the range 0 through FFFFFFFF.

Errors: See messages 032-002 and 032-003.

Quit — Terminate the debugger session

Description: The **Quit** command is used when you have completed debugging and want all breakpoints cleared. **Quit** performs the following actions:

- Sets **RESTore ON**. See "RESTore — Set the restore mode on or off" on page 8-63.
- Clears all breakpoints.
- Optionally takes a VRM dump before ending.
- Unpins the debugger.
- Issues a **Go** command. See "Go — Start executing the program under test" on page 8-53.

After you issue a **Quit** command, you must reload and restart the debugger before it can be used again. The debugger can be loaded with the **Cntl-Alt-Pad4** key sequence or the **Debug SVC**.

Syntax: The **Quit** command is specified as follows:

```
Quit <Dump>
```

If you specify the optional Dump parameter, a VRM dump is taken after the debugger session ends.

Errors: None.

Reset — Clear a user-defined variable

Description: The **Reset** command frees user-defined variables. Because only eight such variables are supported at one time, you may need to **Reset** a variable to make room for another.

Syntax: The **Reset** command is specified as follows:

```
Reset variable name
```

Variable name is the name of a user-defined variable.

Errors: See messages 032-002 and 032-003.

RESTore – Set the restore mode on or off

Description: The **RESTore** command determines whether to replace the contents of the display screen after debugging. When you end a debugger session with the **Quit** command, the screen is restored regardless of the **RESTore** setting.

When you leave the debugger with a **Go** or **Loop** command, you have the option of returning to the screen interrupted when the debugger was started.

The screen cannot be restored when executing a **STEp** instruction. The default setting is **On** (screen contents restored).

Syntax: The **RESTore** command is specified as follows:

```
RESTore On or Off
```

The **On** parameter indicates restore the screen.

The **Off** parameter indicates that the screen is not restored.

Errors: See message 032-002.

Screen — Display a Screen of Data

Description: When you issue the **Screen** command, the screen shown in Figure 8-29 on page 8-65 is displayed. The data is displayed starting at the address specified. If no address is given, the current display address is used.

Syntax: **Screen** is specified in any of the following ways:

```
Screen address
Screen + or -
Screen IAR
Screen Track variable name
Screen On or Off
Screen  On Half
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeros can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the origin.

Plus (+) displays the next 0x80 bytes of data.

Minus (-) displays the previous 0x80 data bytes.

IAR tracks the IAR. The results of specifying **Screen IAR** and **Screen Track IAR** are the same.

Track indicates that the screen display is to track the specified variable.

On and **Off** turn the display on or off. If off, the screen display does not appear when the debugger is started. Screen off is useful if a slow, asynchronous terminal is in use.

On Half displays only the top half of the display screen (the memory display is omitted).

All parameters are optional.

Errors: See message 032-002.

The following figure shows an example of a screen display.

```
ICS xxxx PCS xx MCS xx  IRB xxxx         ECR xxxxxxxx  Mode: (Virtual or Real)
R0  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
R8  xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
TS xxxx COU xxxxxxxx COUS xxxxxxxx
MQ xxxxxxxx CS (xxxx)                                  (1)

IAR xxxxxxxx (ORG+xxxxxxxx) ORG=xxxxxxxx
aaaaaaaa xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
                 | inst  (AIX inst)                    (2)
aaaaaaaa xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx

                   |                                   (3)
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
bbbbbbbb xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx  *ASCII or EBCDIC *
```

Figure 8-29. Format of a Screen Display

Figure notes:

1. Two character mnemonics corresponding to the bits set in the condition status (CS) are displayed here.
2. The instruction at the IAR is disassembled. The AIX assembler opcode is shown in parentheses. The first aaaaaaaa is the IAR value rounded down to the nearest multiple of 16 (0x10). The bbbbbbbb values are addresses of the storage being displayed. bbbbbbbb is a multiple of 16.
3. The actual address requested is marked with a vertical bar (|).

The address of the ECR is displayed only if an APC is configured.

SEt — Initialize a variable

Description: The **SEt** command allows you to define and set a value to a variable. If already defined, this command initializes the variable. The **SEt** command is also used to set one of the reserved variables (such as general purpose registers R0-R15).

Syntax: The **SEt** command is specified as follows:

```
SEt variable name value
```

Variable name is from 1 to 8 alphanumeric characters in length and cannot be a valid constant. The following are valid variables:

- A user-defined variable
- A general purpose register specified as R0-R15
- A segment register specified as S0-S15
- The processor control registers (ICS, PCS, MCS, IRB, TS, COU, COUS, MQ, CS, or IAR)
- The debugger variables **FX** and **ORG**.

Value is any numeric or string value, an expression, or another variable, placed into the specified variable.

Errors: See messages 032-002 and 032-003.

SHow — Show the screen that was up (PC monochrome only)

Description: The **SHow** command displays the screen that was being displayed when the debugger was started. The **SHow** command is valid only when used with a PC monochrome display. After viewing the original screen, press any key to return to the debugger.

Syntax: The **SHow** command is specified as follows:

```
SHow
```

The **SHow** command has no parameters.

Errors: None.

Error handling: The **SHow** command has no effect if the display is not PC monochrome.

SRegs — Display segment registers

Description: The **SRegs** command displays the contents of the memory manager's segment registers and other control registers. Figure 8-30 on page 8-69 shows a sample **SRegs** display.

Syntax: **SRegs** is specified as follows:

```
SRegs
```

The **SRegs** command has no parameters.

Errors: None

The values shown in the following **SRegs** example are for illustration purposes only.

```
SR# P AC SID S K    SR# P AC SID S K    SR# P AC SID S K    SR# P AC SID S K
 0  1 RI 000 0 0     4  1 RI 000 0 0     8  1 RI FFF 0 0     C  1 RI FFF 0 0
 1  1 RI 001 0 0     5  1 RI FFF 0 0     9  1 RI FFF 0 0     D  1 RI FFF 0 0
 2  1 RI 000 0 0     6  1 RI 000 0 0     A  1 RI FFF 0 0     E  1 RI FFF 0 0
 3  1 RI 000 0 0     7  1 RI 000 0 0     B  1 RI FFF 0 0     F  1 RI FFF 0 0

IOBAR:   ......81
SER:     ...00000   _ _ _ _ _ _ _ _ _ _
SEAR:    ########
TRAR:    #.######
TID:     ......00
TCR:     ....0000          IPT:  0FE000    _ _ _ _ _
RAM:     .....000     RAM Addr:  000000       SIZE:  1M
ROS:     .....000     ROS Addr:  F00000       SIZE:  16K
RMDR:    ....0000
```

Figure 8-30. SRegs Display

Figure notes and legend:

- The first four lines of data show the segment register contents. The fields are defined as follows:
  - SR# — segment register number. F (15) is not used.
  - P — Segment present bit
  - AC — Accessibility:
    - R = accessible to the main processor
    - I = accessible to I/O
  - SID — Segment ID
  - S — Special bit
  - K — Key bit.
- The following lines are defined as follows:
  - IOBAR — I/O base address register
  - SER — Storage exception register
  - SEAR — Storage exception address register
  - TRAR — Translation real address register
  - TID — Transaction ID
  - TCR — Translation control register
  - RAM — RAM specification register
  - ROS — ROS specification register
  - RMDR — RAS mode diagnostic register

IPT — Inverted page table address
- - - — Represents bits that will be displayed as one-character mnemonics if set.
RAM Addr — Indicates the beginning of random access memory
ROS Addr — Indicates the beginning of read-only storage.

ST — Store a fullword into memory

Description: **ST** stores a fullword of data into memory by using the processor's **ST** instruction. If the address specified is not word-aligned, it is rounded down to a fullword. **ST** is the correct way to place a fullword of data into I/O memory.

Syntax: **ST** is specified as follows:

```
ST address data
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeros can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

Data is a hexadecimal value to be placed, right-aligned, into the fullword specified by the address.

Errors: See messages 032-002, 032-003, and 032-004.

STArt — Start executing the program under test

Description: See the **Go** command ("Go — Start executing the program under test" on page 8-53).

STC — Store one byte into memory

Description: **STC** stores a byte of data into memory by using the processor's **STC** instruction. **STC** is the correct way to place a byte of data into I/O memory.

Syntax: STC is specified as follows:

```
STC address data
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeros can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

Data is a hexadecimal value to be placed into memory.

Errors: See messages 032-002, 032-003, and 032-004.

STEp — Execute instructions single-step

Description: The **STEp** command executes the instruction pointed to by the IAR, then returns to the debugger. You may optionally execute a series of instructions or an entire subroutine before returning. If **s** is specified, and the instruction is not a branch-and-link, then **STEp** acts as if no parameter had been given. When stepping over a branch-and-execute instruction, both the branch and its subject instruction are executed normally.

Syntax: **STEp** is specified as follows:

```
STEp n or s
```

n is a positive, decimal number.

s executes a subroutine as if it were one instruction.

Errors: See messages 032-002 and 032-005.

STH — Store a halfword into memory

Description: The **STH** command stores a halfword of data into memory by using the processor's **STH** instruction. If the address specified is not halfword-aligned, it will be rounded down to a halfword boundary. **STH** is the correct way to place a halfword into I/O memory space.

Syntax: **STH** is specified as follows:

```
STH address data
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeros can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

Data is a hexadecimal value to be placed, right-aligned, into the halfword specified by the address.

Errors: See messages 032-002, 032-003, and 032-004.

STOp — Set a breakpoint

See the **BReak** command ("BReak — Set a breakpoint" on page 8-18).

STOPS — List the current breakpoints

See the BREAKS command ("BREAKS — List the current breakpoints" on page 8-19).

SWap — Switch to or from an RS-232 port

Description: The **SWap** command allows you to switch terminals from the default display to an RS-232 port display. The debugger can use an RS-232 port if the terminal is asynchronous and is attached to the primary asynchronous port (at address 0xF00003F8). The **SWap** command also returns you to the original terminal from the RS-232 device.

The protocol for terminal switching is defined as follows:

Data rate: 9600 baud

Data bits: 7

Parity: Space

Stop bits: 2

Receive: Receives only if DSR (data set ready) is active.

Transmit: The following actions occur:

1. Raises DTR (data terminal ready) and RTS (request to send)
2. Waits for DSR (data set ready) and CTS (clear to send)
3. Begins transmission.

Syntax: **SWap** is specified as follows:

```
SWap
```

The **SWap** command has no parameters.

Errors: None.

Tlb — Display the translate lookaside buffer

Description: **Tlb** displays either all the translation lookaside buffer entries, or only those marked as valid by the hardware. Figure 8-31 on page 8-80 shows a sample **TLB** display.

Syntax: **Tlb** is specified as follows:

```
Tlb ALL
```

ALL, if specified, indicates that all **TLB** entries are to be displayed. If **ALL** is not specified, the entries marked by the memory manager as invalid show up as dashes (−).

Errors: None.

The following figure is an example of a TLB display.

```
     Virtual       Real   V K  W TI Lock        Virtual       Real   V K  W TI Lock
0  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
1  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
2  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
3  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
4  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
5  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
6  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
7  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
8  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
9  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
A  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
B  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
C  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
D  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
E  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
F  000.0000000  000000 1 0  0 00 0000    0  000.0000000  000000 1 0  0 00 0000
```

Figure 8-31. Translation Lookaside Buffer Display

Figure notes and legend:

- The display shown has all the entries marked as valid.
- The other values are for illustration purposes only.
- The following are the field descriptions:

  Virtual — Consists of the segment ID and the 28-bit displacement (iii.ddddddd)
  Real — The real storage address
  V — Valid bit
  W — Line Write protection bit
  TI — Transaction ID
  Lock — Lock word for line protect.

TOuch — Page-in the referenced memory

Description: The **TOuch** command allows you to page-in memory that cannot currently be referenced by the debugger. The machine state must permit page faulting for this command to work. **TOuch** is allowed only if translation is on, interrupts are enabled, and you are running as a process. If you are debugging in a device driver, for example, you cannot successfully issue the **TOuch** command. In addition, if a memory reference would normally cause an abend, bringing the memory in with **TOuch** will also cause the abend.

Syntax: **TOuch** is specified as follows:

```
TOuch address
```

Address is defined as follows:

xxxxxxxx	x must be a valid hexadecimal digit. Leading zeros can be omitted.
R xxxxxxxx	R implies a real address and is the default if mode = Real.
V xxxxxxxx	V implies a virtual address and is the default if mode = Virtual.
+xxxxxxxx	+ means that the value is a displacement relative to the setting of the Origin.

Errors: See messages 032-002, 032-003, and 032-017.

TRace — Display formatted trace table entries

Description: The **TRace** command displays formatted trace table entries. This command is typically used to display 8 formatted trace table entries. The format of an entry is shown in Figure 8-32.

Note that the VRM debugger formats trace table entries requested by the VRM **_vrmtrc** routine, not the VRM **_trcvrm** routine.

Syntax: **TRace** is specified as follows:

```
TRace + or -
TRace L
```

Plus (+) means display the next 8 entries.

Minus (-) means display the previous 8 entries.

L means display the last (most recent) 8 entries.

If no parameter is specified, the 8 trace table entries shown previously are shown again. The very first issuance of the **TRace** command executes the same way as **TRace L**.

Errors: None.

```
L ---Type--- IAR  ######## in Proc. ######## ICS: #### CS: ####
  VICS: ######## PO@/SVC#: ######## PB @ ######## ELevel: ########
```

Figure 8-32. Trace Entry Display Format

Figure notes:

- All the '#' values represent hexadecimal digits.
- L has an actual value of 'L' only if the entry is the last (most recent) entry in the table; otherwise, this field is blank.
- Type may be any of the following values:

  Level 0 - Level 0 interrupt occurred
  Level 1 - Level 1 interrupt occurred
  Level 2 - Level 2 interrupt occurred
  Level 3 - Level 3 interrupt occurred
  Level 4 - Level 4 interrupt occurred
  Level 5 - Level 5 interrupt occurred

Level 6 - Level 6 interrupt occurred
SVC - SVC interrupt
Pgm. Chk. - Program check interrupt
Mch. Chk. - Machine check interrupt
VRM Disp. - A VRM process was dispatched
VM Disp. - A virtual machine process was dispatched
User trace - A debugger trace point was encountered. (See "BReak — Set a breakpoint" on page 8-18 for more information on breakpoints.)
U #### - A user-defined type of type #### has occurred.

User-defined entries are displayed as shown in Figure 8-33.

- For 'in Proc.', the ######## provides the process or SLIH ID. The ICS and CS values are also shown. The 'PB @' field gives the process block address when a process is dispatched.
- VICS is the virtual ICS for VM Disp. and SVC entries. In the case of "Pgm. Chk." and "Mch. Chk." type entries, VICS is replaced by the PCS and MCS, respectively.
- P0@/SVC# provides the real address of the virtual machine's page 0 for VM Dispatch entries. For SVC entries, this field provides the SVC code.
- PB @ provides the address of the process block (used only for VM Dispatch and SVC entries).
- ELevel indicates the the virtual machine's execution level (used only for VM Dispatch and SVC entries).

The following figure shows how a user-defined trace entry is displayed.

```
U ####    ######## ######## ######## ########
          ######## ######## ######## ########
```

Figure 8-33. User-defined Trace Entry Display

Vars — Display a list of the user-defined variables

Description: The **Vars** command displays the user-defined variables and their values. The value of FX, the variable set by the **Find** command, is also displayed. Figure 8-34 shows a sample screen produced by the **Vars** command.

Syntax: **Vars** is specified as follows:

```
Vars
```

This command has no parameters.

Errors: None.

The following figure shows an example of the screen produced by the **Vars** command.

```
Listing of the user-defined variables:
  vvvvvvv1  Hex/Dec=xxxxxxxx
  vvvvvvv2  Hex=xxxxxxxx
  v3        Dec=xxxxxxxx
  strvar    str=abcdef
  FX        Hex=xxxxxxxx
  ORG            Hex=xxxxxxxx
There are n free variables.
```

Figure 8-34. Vars Display

Figure notes:

- The value of vvvvvvv1 is valid both in hexadecimal and decimal.
- The value of vvvvvvv2 is only valid in hexadecimal.
- The value strvar represents a quoted string.
- FX is the **Find** command's result variable.
- The last line gives the number (n) of free variable slots (8 if no variables have been defined).

Xlate — Display the real address

Description: If the specified virtual address maps to a real address, this command displays the real address. Figure 8-35 shows the resulting display.

Syntax: **Xlate** is specified as follows:

```
Xlate address
```

Address is any hexadecimal number of up to 8 digits.

Errors: See messages 032-002 and 032-003.

The following figure is an example of the screen that appears when you request the **Xlate** function.

```
vvvvvvvv -Virtual- rrrrrrrr -Real-
vvvvvvvv -Virtual- no real mapping
```

Figure 8-35. Xlate Display

Figure note:

The first line is displayed if the address has a real mapping. If the address has no real mapping, the second line is displayed.

Appendix A. Index of Supervisor Call Instructions

The following table lists the SVCs by type and, within each type, numerically by SVC code.

SVC Code	Description
Execution Control SVCs	
FFAF	“Free Floating-Point Register Sets SVC” on page 4-12
FFB0	“Allocate Floating-Point Register Sets SVC” on page 4-10
FFBA	“Post SVC” on page 4-14
FFD6	“No Operation SVC” on page 4-13
FFF9	“Return SVC” on page 4-16
FFFA	“Dispatch SVC” on page 4-11
FFFB	“Virtual Machine Wait SVC” on page 4-22
FFFC	“Return From Interrupt SVC” on page 4-17
FFFD	“Soft Interrupt SVC” on page 4-21
FFFE	“Set Interrupt SVC” on page 4-18
FFFF	“Set Timer SVC” on page 4-19
Input/Output SVCs	
FFB8	“Ring Queue Put Word SVC” on page 4-62
FFB9	“Ring Queue Get Word SVC” on page 4-61
FFBF	“Send Command SVC” on page 4-63
FFD9	“Define Code SVC” on page 4-49
FFF3	“Cancel I/O SVC” on page 4-48
FFF4	“Start I/O SVC” on page 4-67
FFF5	“Query Device SVC” on page 4-58
FFF6	“Detach Device SVC” on page 4-57
FFF7	“Attach Device SVC” on page 4-47
FFF8	“Define Device SVC” on page 4-51

Memory Management SVCs

FFB2 “UnMap Page Range SVC” on page 4-44

FFC3 “Map Page Range SVC” on page 4-34

FFD2 “Discard Page Range SVC” on page 4-30

FFD3 “Purge Segments SVC” on page 4-41

FFE0 “Purge Page Range SVC” on page 4-39

FFE2 “UnPin Page Range SVC” on page 4-45

FFE3 “Pin Page Range SVC” on page 4-37

FFE4 “Clear Segment Registers SVC” on page 4-25

FFE5 “Load Segment Registers SVC” on page 4-32

FFE9 “Query Page Protect SVC” on page 4-43

FFEA “Protect Pages SVC” on page 4-38

FFEE “Change Segment Size SVC” on page 4-24

FFEF “Copy Segment SVC” on page 4-26

FFF1 “Destroy Segment SVC” on page 4-29

FFF2 “Create Segment SVC” on page 4-27

Virtual Machine Communications SVCs

FFDD “Set Message Receive SVC” on page 4-75

FFDE “Send Address Message SVC” on page 4-73

FFDF “Send Immediate Message SVC” on page 4-74

Machine Control SVCs

FFB6 "Debug a Virtual Machine SVC" on page 4-77

FFBD "Update VRM SVC" on page 4-87

FFC2 "IPL Virtual Machine SVC" on page 4-78

FFD7 "Re-IPL VRM SVC" on page 4-85

FFD4 "Machine Identification SVC" on page 4-82

FFD8 "Query Virtual Machine SVC" on page 4-83

FFDC "Terminate Virtual Machine SVC" on page 4-86

NVRAM Control SVCs

FFB4 "Write Data to NVRAM SVC" on page 4-90

FFB5 "Read NVRAM SVC" on page 4-89

Virtual Terminal SVCs

These SVCs are described in *Virtual Resource Manager Device Support.*

FFCB KSR Output Short SVC

FFCC VT Output SVC

FFC6 VT Query SVC

FFC8 VT Set Structure SVC

Appendix B. TOC Object Module Information

This appendix provides information about the object module format used in the VRM. Programmers adding code to the VRM or using the VRM debugger may find this information to be useful.

TOC Object Module Format

The VRM uses an object module format different from that used by AIX Operating System. The AIX Operating System object module format (**a.out**) is described in *IBM RT PC AIX Operating System Technical Reference.* This appendix describes the VRM object module format, which is called the Table of Contents (TOC) format.

In TOC object modules, address constants are accumulated into a common area called the TOC. This area is addressed through a dedicated register and is similar to the **a.out** constant pool, with one major difference:

- When one or more TOC modules are bound together, the TOCs for each module are accumulated into one TOC in the resulting module.
- When one or more **a.out** modules are bound together, each module retains its own constant pool. All of the constant pools exist in the data section of the resulting module.

For TOC object modules, all of the loader relocation requirements are confined to the TOC area. The TOC area is contained in the read/write section (Section 2) of the object module. Only a single address constant is required for any variable in a bound module.

In an **a.out** object module, a variable might have several address constants for a bound module. This is because an **a.out** module might have more than one constant pool in its data section, and the variable will have an address constant in the constant pool of each module in which it is referenced.

TOC Object Module Definition

A TOC object module is divided into a header plus three additional sections, as shown below:

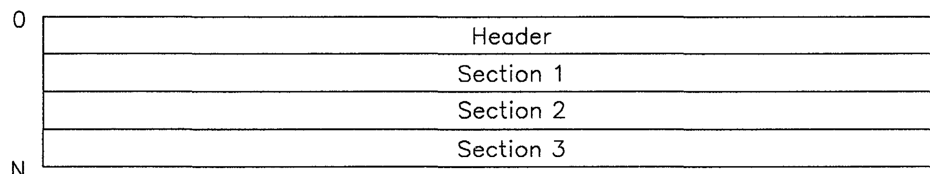


Figure B-1. TOC Object Module Structure

The header contains information required to run the object module. This includes various addresses such as the starting address and the entry address. It also contains the length of each section in the object module and the number of objects each section contains.

Section 1 contains the read-only data of a TOC object module. Read-Only data consists of the object code and the internal constants of the module. This section corresponds to the **a.out** text section.

Section 2 contains the read/write data of a TOC object module. This data consists of the read/write data of the module and the TOC of the module. Any data of unknown type is also put into this section. This section corresponds to the **a.out** data section.

Section 3 contains the loader information of a TOC object module. This section contains the external symbol names, external symbol dictionary (ESD) entries, and the relocation dictionary (RLD) entries required by the loader. This section also contains the import and export information for the object module. This section is similar to the **a.out** symbol table, and the **a.out** data relocation area.

The various parts of a TOC object module are described in greater detail on the following pages.

TOC Header Definition

The header is 96 bytes long and has the following structure:

Offset	Field	
0	Version	
4	Starting Address	
8	Reserved	
12	TOC Address	
16	Common Length	
20	Entry Address	
24	Entry Type	
28		
32	Module Type	Maximum Alignment
36	Reserved	
40	Reserved	
44	Reserved	
48	Reserved	
52	Reserved	Object Machine ID
56	Section 1 Length	
60	Section 2 Length	
64	Section 3 Length	
68	Reserved	
72	String Length	
76	Reserved	
80	ESD Count	
84	Reserved	
88	RLD Count	
92	Reserved	
96		

Figure B-2. TOC Object Module Header

The fields that make up the header are defined below:

Version: The type of object module format. This can be one of two values:

- 0x00000306 — IBM-supplied object module
- 0x00010306 — User-supplied object module.

Starting address:
The starting address, or origin, of the object module code.

TOC address: The address of the Table of Contents (TOC) in Section 2. In a converted object module, this is the constant pool pointer to the main entry point.

Common length:
The length of the common section past Section 2. This is similar to the BSS section in **a.out** object modules.

Entry address:
Address of the entry point into the object module.

Entry type: This field is used by the VRM. It is always set to 0x4040404040404040.

Module type: This defines whether or not the module is reusable or reentrant. This field can be set to one of three values:

- 0xF1D3 — Module is not reusable
- 0xE2D9 — Module is serially reusable
- 0x09C5 — Module is reentrant.

Maximum alignment:
The most stringent byte alignment required by any control section (CSECT) in the object module. This can be one of four values:

- 0 — Byte-aligned
- 1 — Halfword-aligned
- 2 — Word-aligned
- 3 — Doubleword-aligned.

Object machine ID:
Identifies the target machine used to run the system. This can be one of two values:

- 0 — Unknown system
- 800 — IBM system.

Section 1 length:
Length of section 1 in bytes.

Section 2 length:
Length of section 2 in bytes.

Section 3 length:
Length of section 3 in bytes.

String length: Number of characters of string storage.

ESD count: Number of external symbol dictionary (ESD) entries. Each ESD entry is five words in length.

RLD count: Number of relocation dictionary (RLD) entries. Each RLD entry is two words in length.

TOC Section 1 Definition

Section 1 contains the read-only portion of the object module. The CSECTs in this section are ordered by storage class as follows:

- PR — Program code
- RO — Read-only data.

TOC Section 2 Definition

Section 2 contains the read/write portion of the object module. The CSECTs in this section are ordered by storage class as follows:

- RW — Read/Write data
- TC — Table of Contents.

TOC Section 3 Definition

Section 3 contains the loader tables used by the **Define Code SVC** and the loadlist processor. This section consists of three parts: string storage, ESD entries, and RLD entries.

String Storage

This part contains all of the external symbol names used by the object module. The symbol names are stored as ASCII values. An entry in string storage consists of a 1-byte length field and an n-byte string field. The length does not include the byte required for the length field. If the last entry does not end at a word boundary, the remainder of the last word in the entry is padded with blanks.

ESD Entries

This part contains all of the external symbol dictionary (ESD) entries used by the object module. The ESD entries for CSECTS of non-zero length are in order of increasing address and do not overlap. Each ESD entry is five words in length and is aligned on a word boundary within the object module. The format of an ESD entry is shown in Figure B-3 on page B-6.

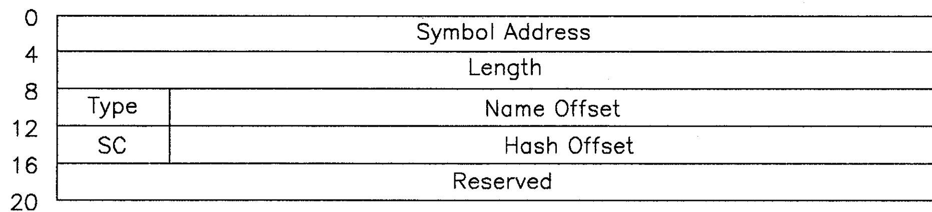


Figure B-3. Format of an ESD Entry

The fields of an ESD entry have the following definitions:

Symbol address:
Absolute memory address of the symbol.

Length: Number of bytes required by the CSECT.

Type: 4-bit flag set to one of the following ESD types:

- 0000 — ER (external reference)
- 0001 — SD (section definition)
- 0010 — LD (label definition)
- 0011 — CM (common).

Name offset: The byte offset of the symbol name in the string storage section.

Storage class: The storage class number for the ESD entry. This field can have one of the following values:

- 0000 — Program code
- 0001 — Read-only data
- 0011 — Table of Contents
- 0101 — Read/Write data.

Hash offset: This value is the byte offset of the symbol's declaration hash in the string storage section.

RLD Entries

This part contains all of the relocation dictionary (RLD) entries used by the object module. These entries identify the relocatable address constants that must be relocated at run time. The RLD entries are in order of increasing address. Each RLD entry is two words in length and is aligned on a word boundary within the object module.

The format of an RLD entry is shown below:

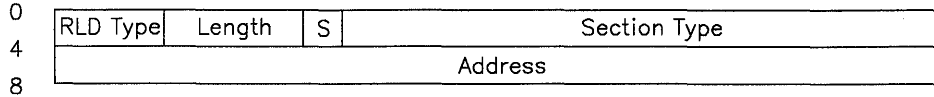


Figure B-4. Format of an RLD Entry

The fields of an RLD entry have the following definitions:

RLD type: 4-bit flag set to 0000. This indicates positive relocation. Converted object modules must use positive relocation for all imported symbols. TOC modules can also have a value of 0101 in this field. This indicates positive relocation of the symbol's TOC, not the symbol itself.

Length: 5-bit length of the address constant minus 1. This is always set to 11111.

Sign: 1 bit for the sign of the address constant. This is always set to 0 to indicate positive.

Section type: The location of the address constant in the object module. This can be one of the following values:

- 1 — Relocate relative to section 1.
- 2 — Relocate relative to section 2.
- 3 to N — Relocate relative to a specific ESD entry in the ESD section of section 3. Note that this value is two greater than the actual ESD being referenced. For example, a value of 3 indicates the first ESD entry. A value of 5 indicates the third ESD entry.

Address: Address of the address constant.

The following restriction apply to address constants:

- Only unsigned fullword address constants are supported.
- The address constant must be on a word boundary.
- There should be no RLD entries for section 1.
- RLD entries for section 2 must use positive relocation.
- An imported symbol must have a single positive relocation RLD entry.

Subroutine Linkage Conventions

The subroutine linkage conventions described here must be followed by all code written using the RT PC C language compiler or AIX Operating System assembler, bound with the AIX Operating System binder and then converted by the **a.out** converter to run in the VRM. All VRM services assume these conventions are followed.

If a register is not saved during the call, then its contents are changed during the call. Conversely, if a register is saved, then its contents are not changed during the call, and you can use the register for "scratch" data. Examples of scratch data are register variables, the current constant pool pointer, temporary values needed across the call, and various base registers.

By convention, the following registers are used for the following functions. Note that there are separate frame and argument pointer registers.

GPR0 = Pointer to the constant pool

GPR1 = Pointer to the stack

GPR15 = Address to return to after the call.

GPR2 contains any returned value. Note that GPRs 1 and 6 through 14 must be saved and restored after the call.

C routine prologs conventionally save the constant pool pointer in register 14. However, the calling sequence does not require this.

You can use the **.set** pseudo-op to define the names of these registers. For example,

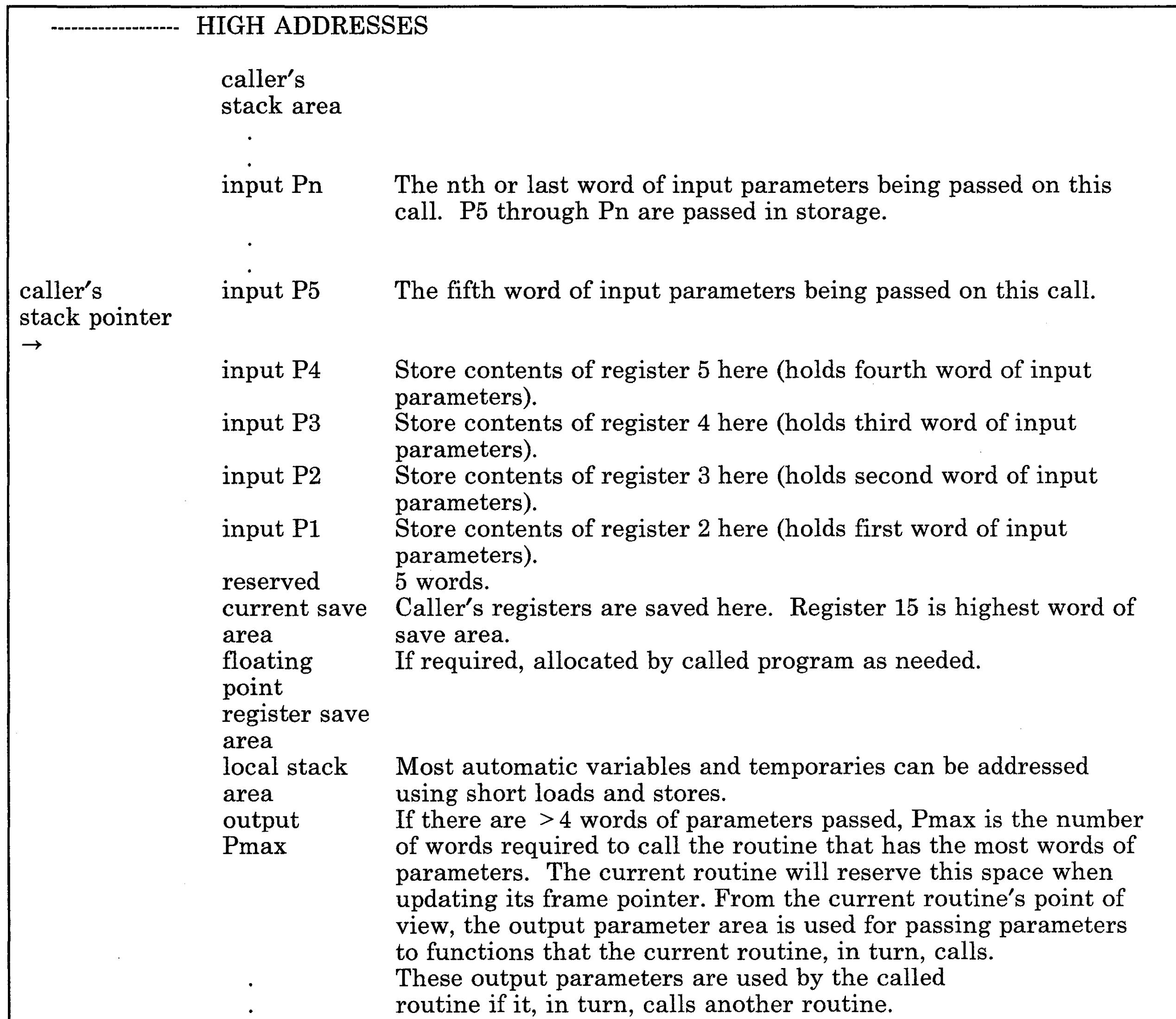
```
.set link,15
```

allows you to use the symbol "link" to refer to GPR 15.

Everything in the stack is aligned on word boundaries. The length of each area defined in the stack is padded to an even number of words.

The Stack Frame

Figure B-5 represents the contents of a stack frame. In this figure, the current routine has acquired a stack frame which allows it to call other functions. If no functions are called, and there are no local variables, then the function need not allocate a stack frame or adjust the frame pointer. It can still use the register save area at the bottom of the caller's stack frame, if needed.



------------------	HIGH ADDRESSES	
	caller's stack area	
	.	
	.	
	input Pn	The nth or last word of input parameters being passed on this call. P5 through Pn are passed in storage.
	.	
	.	
caller's stack pointer →	input P5	The fifth word of input parameters being passed on this call.
	input P4	Store contents of register 5 here (holds fourth word of input parameters).
	input P3	Store contents of register 4 here (holds third word of input parameters).
	input P2	Store contents of register 3 here (holds second word of input parameters).
	input P1	Store contents of register 2 here (holds first word of input parameters).
	reserved	5 words.
	current save area	Caller's registers are saved here. Register 15 is highest word of save area.
	floating point register save area	If required, allocated by called program as needed.
	local stack area	Most automatic variables and temporaries can be addressed using short loads and stores.
	output Pmax	If there are >4 words of parameters passed, Pmax is the number of words required to call the routine that has the most words of parameters. The current routine will reserve this space when updating its frame pointer. From the current routine's point of view, the output parameter area is used for passing parameters to functions that the current routine, in turn, calls.
	.	These output parameters are used by the called
	.	routine if it, in turn, calls another routine.

Figure B-5 (Part 1 of 2). Contents of a Stack Frame

current stack pointer →	output P5	If there are >4 words of parameters passed, this is the fifth word of input parameters passed to the next called routine.
	output P4	Store contents of register 5 here (holds fourth word of input parameters passed to the next called routine). This word is always reserved, whether this routine calls others or not.
	output P3	Store contents of register 4 here (holds third word of input parameters passed to the next called routine). This word is always reserved, whether this routine calls others or not.
	output P2	Store contents of register 3 here (holds second word of input parameters passed to the next called routine). This word is always reserved, whether this routine calls others or not.
	output P1	Store contents of register 2 here (holds first word of input parameters passed to the next called routine). This word is always reserved, whether this routine calls others or not.
	reserved	5 words.
	next save area	Functions you call will save here. They will determine the size (up to a maximum of 64 bytes).
--------------------	LOW ADDRESSES	

Figure B-5 (Part 2 of 2). Contents of a Stack Frame

The first four words of arguments are passed in registers 2 through 5. Any other arguments are passed on the stack; the frame pointer points directly to these arguments. The called routine's arguments can be saved in two ways:

- The called routine can save its first four arguments in the four words of memory below the frame pointer, or
- The calling routine can save these arguments by putting them in the preserved registers 6 through 14.

Note that since the frame size is known, input parameters can be addressed using the current stack pointer as a base register.

The Constant Pool

Each routine uses its own pool of constants and pointers. At call time, the routine's caller must pass the address of the constant pool—that is, the pcp—in register 0.

The first word in a routine's constant pool must contain the address of the beginning of the code for the routine (that is, the routine's entry point). When used as a C language function pointer, the "address" of the routine is in fact the address of its constant pool.

All addresses of external routines end up in the constant pool of the calling routine. These are 32-bit addresses.

Stack Overflow

The method of checking for stack overflow depends on the operating system. The linkage convention does not specify which method will be used.

The VRM does not provide an in-line check for overflow. This is because the VRM is using page faults to detect stores past the end of the stack segment. Some frame-size limits are imposed to bound the amount of protected storage which must be provided.

Appendix C. Key Sequences for System Functions

The following keystroke combinations perform the indicated system functions when entered. For the three-key sequences, the first two keys must be pressed down and held, then the third key is pressed to get the required function. For the two-key sequences, the keys must be pressed simultaneously. Note that certain key sequences are appropriate only in specific environments. Also, keys described as Pad**n**, where **n** is a number, indicate the keys found in the numeric pad to the right of the main keyboard area.

Note: From the standard RT PC keyboard, most functions initiated with the `Ctrl + Alt + (some other key)` sequence require that you press the Alt key found on the **left** side of the keyboard. Exceptions to this rule are indicated with the key sequence definition.

Keystroke Combination	System Function
Ctrl + Alt + Pause	Performs a soft re-IPL of the central processing unit This keystroke combination re-IPLs the VRM and initiates an IPL from the default IPL sequence or from the sequence defined with a `Ctrl + Alt + (a,b,c,d,e)` combination. The IPL initiated with this key sequence skips some of the standard tests done during a typical IPL.
Ctrl + Alt + Home	Performs a re-IPL of virtual machines This keystroke combination terminates all running virtual machines, then re-IPLs any automatically IPLed virtual machines. This sequence does not re-IPL the VRM.
Ctrl + Alt + Pad6	Performs a power on reset re-IPL This keystroke combination initiates an IPL as if the machine was just powered on. It re-IPLs the VRM, performs all IPL testing, and proceeds with the default IPL sequence or the the sequence defined with `Ctrl + Alt + (a,b,c,d,e)`.
Ctrl + Alt + Pad7	Performs a dump of all real memory This key sequence performs a system dump initiated by the user or in response to a system abend. If the system abends, the LEDs will flash C6 alternately with the abend code (see *IBM RT PC Messages Reference* for a description of the abend codes). At this time you should load a dump diskette and enter `Ctrl + Alt + Pad7`, which begins the dump.

If the dump is user-initiated, the key sequence is entered twice. The first key sequence causes the LEDs to flash C6 alternately with 00. At this time you should load a dump diskette into the primary diskette drive. Then you issue this key sequence a second time to actually perform the dump.

In either case, a C7 flashed on the LEDs indicates that the dump diskette is full. Remove the first dump diskette and insert another one. When the diskette door is closed, the dump resumes.

Ctrl + Alt + Pad8 Performs a system dump of selected areas of memory

This key sequence also performs a system dump initiated by the user or in response to a system abend, but only of selected areas in memory. If the system abends, the LEDs will flash C6 alternately with the abend code (see *IBM RT PC Messages Reference* for a description of the abend codes). At this time you should load a dump diskette and enter `Ctrl + Alt + Pad8`, which begins the dump.

If the dump is user-initiated, the key sequence is entered twice. The first key sequence causes the LEDs to flash C6 alternately with 00. At this time you should load a dump diskette into the primary diskette drive. Then you issue this key sequence a second time to actually perform the dump.

Ctrl + Alt + Pad4 Invokes the VRM debugger

Ctrl + Alt + End Performs a dump of the first virtual machine

Ctrl + Alt + Del Performs an IPL of the optional Coprocessor.

Ctrl + Alt + Action Exits the Coprocessor mode.

Alt + Action Changes the active display screen to the next virtual terminal (if any). You can use either **Alt** key to initiate this function.

Shift + Action Changes the active display screen to the previous virtual terminal (if any)

Ctrl + Action Changes the active display screen to the command virtual terminal (if defined).

The following key sequences only update the data in NVRAM. After issuing this key sequence, you can perform an IPL from the specified device (if it is configured in the system) by entering the key sequence **Ctrl + Alt + Pause**.

Ctrl + Alt + a Updates NVRAM for diskette device #1

Ctrl + Alt + b Updates NVRAM for diskette device #2

Ctrl + Alt + c Updates NVRAM for fixed-disk device #1

Ctrl + Alt + d	Updates NVRAM for fixed-disk device #2
Ctrl + Alt + e	Updates NVRAM for fixed-disk device #3

Appendix D. C Language VRM Subroutines

This section describes the VRM runtime routines in C language programming conventions. The routines are presented alphabetically by function call. The C language declarations for external variables are also provided in "External Variables" on page D-21. Note that structures and defines used with these calls can be found in the AIX Operating System file **vrm.h**.

_assign (Assign device to coprocessor)

```
int _assign (option, device_id)
unsigned int option;
unsigned int device_id;
```

_attchq (Attach queue)

```
int _attchq (from_id, to_id, path_id, ack_parms)
unsigned int from_id;
unsigned int to_id;
unsigned int *path_id;
struct ack_prm *ack_parms;
```

_attchs (Attach segment)

```
int _attchs (segment_id)
unsigned int segment_id;
```

_badblk (Minidisk bad blocks)

```
int _badblk (disk_iodn, bad_block, io_option, minidisk_iodn)
unsigned int disk_iodn;
unsigned int bad_block;
unsigned int io_option;
unsigned int minidisk_iodn;
```

_bffree (Free allocated buffer)

```
int _bffree (address_to_free)
unsigned int address_to_free;
```

_bfget (Allocate buffer from the buffer pool)

```
int _bfget (buffer_address)
unsigned int buffer_address;
```

_bind (Bind module)

```
int _bind (import_module_id, export_modules, num_elements,
           symbols, size_symbols)
unsigned int import_module_id;
unsigned int export_modules[];
int num_elements;
char *symbols;
int size_symbols;
```

_bvint (Broadcast virtual interrupt)

```
int _bvint (bvint_qelem)
struct deque_qe *bvint_qelem;
```

_canclq (Cancel enqueue)

```
int _canclq (path_id, option)
unsigned int path_id;
unsigned int option;
```

_change (Change attributes)

```
int _change (object_id, flags, priority, entrypts, num_elements)
unsigned int object_id;
unsigned int flags;
unsigned int priority;
struct entry_pt entrypts[];
int num_elements;
```

_chgmsk (Change bus mask)

```
int _chgmsk (command, level_number)
unsigned int command;
unsigned int level_number;
```

_chkblk (Check for bad blocks)

```
int _chkblk (chkblk_qelem, num_blocks, call_type, disk_iodn)
struct enque_qe *chkblk_qlem;
unsigned int *num_blocks;
unsigned int call_type;
unsigned int disk_iodn;
```

_cnltmr (Cancel interval timer)

```
int _cnltmr (timer_id)
unsigned int timer_id;
```

_copy (Copy module)

```
int _copy (option, module_id, new_module_id)
unsigned int option;
unsigned int module_id;
unsigned int *new_module_id;
```

_creatp (Create process)

```
int _creatp (name, process_id )
char name[4];
unsigned int *process_id;
```

_creatq (Create queue)

```
int _creatq (server_id, priorities, max_paths, module_id,
             queue_id, ecb_mask)
unsigned int server_id;
unsigned int priorities;
unsigned int max_paths;
unsigned int module_id;
unsigned int *queue_id;
unsigned int *ecb_mask;
```

_creats (Create semaphore)

```
int _creats (initial_value, semaphore_id)
unsigned int initial_value;
unsigned int *semaphore_id;
```

_ctldvt (Control device timer)

```
int _ctldvt (option, queue_id)
unsigned int option;
unsigned int queue_id;
```

_cveara (Convert effective address to real address)

```
int _cveara (address, real_address)
unsigned int address;
unsigned int *real_address;
```

_dalct (Allocate device dependent data)

```
int _dalct (device_structure, dds_length, data_length)
struct dds *device_structure;
int dds_length;
int data_length;
```

_defind (Define device)

```
int _defind (option, segment_id, dds_address, queue_id, module_id,
             device_id )
unsigned int option;
unsigned int segment_id;
unsigned int dds_address;
unsigned int queue_id;
unsigned int module_id;
unsigned int *device_id;
```

_deque (Dequeue element)

```
int _deque (queue_id, option, deque_qelem)
unsigned int queue_id;
unsigned int option;
struct deque_qe *deque_qelem;
```

_detchq (Detach queue)

```
int _detchq (path_id)
unsigned int path_id;
```

_detchs (Detach segment)

```
int _detchs (segment_id)
unsigned int segment_id;
```

_dmamov (Set up region mode)

```
int _dmamov (seg_id, source, target, length)
unsigned int seg_id;
unsigned int source;
unsigned int target;
unsigned int length;
```

_dmptbl (Save/Get dump table entry)

```
int _dmptbl (command, component_id, length, buffer, buffer_length)
int command;
int component_id;
int *length;
char *buffer_address;
int buffer_length;
```

_dstryq (Destroy queue)

```
int _dstryq (queue_id)
unsigned int queue_id;
```

_dstrys (Destroy semaphore)

```
int _dstrys (semaphore_id)
unsigned int semaphore_id;
```

_enque, _enq (Enqueue element)

```
int _enque (enque_qelem)
union queue_element *enque_qelem;
         -or-
int _enq (enque_qelem)
union queue_element *enque_qelem;
```

_erecv (Receive data from bus memory)

```
void _erecv (target_address, target_length,
               source_address, source_length);
char *to_buffer;
int to_length;
char *from_buffer;
int from_length;
```

_errvrm (Generate error entry)

```
void _errvrm (dds_address)
unsigned int dds_address;
```

_initp (Initialize process)

```
int _initp (process_id, module_id, init_parms, parms_length,
             parms_segid, parms_offset)
unsigned int process_id;
unsigned int module_id;
char *init_parms;
int parms_length;
unsigned int *parms_segid;
unsigned int *parms_offset;
```

_loadb (Load byte)

```
int _loadb (address)
unsigned int address;
```

_loadh (Load halfword)

```
int _loadh (address)
unsigned int address;
```

_loadw (Load word)

```
int _loadw (address, return_code)
unsigned int address;
int *return_code;
```

_lsr (Load segment register)

```
int _lsr (segment_id, segment_reg, protection)
unsigned int segment_id;
unsigned int segment_reg;
unsigned int protection;
```

_mallc (Allocate memory)

```
unsigned char *_mallc (length, alignment)
int length;
int alignment;
```

_mapsys (Map system memory)

```
int _mapsys (operation, module_id, bus_address, length, offset)
unsigned int operation;
unsigned int module_id;
unsigned int bus_address;
unsigned int length;
unsigned int *offset;
```

_mdmchk (Minidisk manager check)

```
int _mdmchk (qe_type, mdmchk_qelem)
unsigned int qe_type;
struct enque_qe *mdmchk_qelem;
```

_mfree (Free allocated memory)

```
void _mfree (storage, length)
unsigned char *storage;
int length;
```

_mvbuff (Move buffer)

```
int _mvbuff (to_buffer, reserved, from_buffer, from_length)
char *to_buffer;
unsigned int reserved;
char *from_buffer;
int from_length;
```

_peekq (Peek at queue element)

```
unsigned int _peekq (QID, offset, addr_of_copied_qe)
unsigned int QID;
unsigned int offset;
unsigned int addr_of_copied_qe;
```

_pinpgs (Pin pages)

```
int _pinpgs (segment_id, first_page, num_pages)
unsigned int segment_id;
int first_page;
int num_pages;
```

_post (Post event control bit)

```
int _post (ecb_mask, target_pid)
unsigned int ecb_mask;
unsigned int target_pid;
```

_qra (Convert virtual address to real address)

```
int _qra (segment_id, address, real_address)
unsigned int segment_id;
unsigned int address;
unsigned int *real_address;
```

_qryds (Query device status)

```
int _qryds (device_id, qryds_struct, qryds_length)
unsigned int device_id;
struct qds *qryds_struct;
int qryds_length;
```

_qsid (Query segment ID)

```
int _qsid (address, segment_id)
unsigned int address;
unsigned int *segment_id;
```

_queryd (Query device identifier)

```
int _queryd (iodn, device_id)
unsigned int iodn;
unsigned int *device_id;
```

_queryi (Query ID)

```
int _queryi (server_id, number_queues, qid_array, num_elements)
unsigned int server_id;
int *number_queues;
struct queue_id qid_array[];
int num_elements;
```

_querym (Query module ID)

```
int _querym (iocn, module_id)
unsigned int iocn;
unsigned int *module_id;
```

_queryp (Query attached path)

```
int _queryp (path_info)
struct queryp_struct *path_info;
```

_queryv (Query virtual machine)

```
int _queryv (qry_type, qry_struct, qry_size, vm_name)
unsigned int qry_type;
char *qry_struct;
int qry_size;
unsigned int vm_name;
```

_rdblk (Read block)

```
void _rdblk (io_address, real_address)
unsigned int io_address;
unsigned int real_address;
```

_rdwds (Read words)

```
void _rdwds (io_address, real_address, length)
unsigned int io_address;
unsigned int real_address;
unsigned int length;
```

_readq (Read queue element)

```
int _readq (queue_id, readq_qelem)
unsigned int queue_id;
union queue_element *readq_qelem;
```

_recv (Receive semaphore)

```
int _recv (semaphore_id)
unsigned int semaphore_id;
```

_rqc (Ring queue create)

```
unsigned int _rqc (length, ecb, pid)
int length;
unsigned int ecb;
unsigned int pid;
```

_rqd (Ring queue delete)

```
unsigned int _rqd (rqpointer)
unsigned int rqpointer;
```

_rqgetw (Ring queue get word)

```
unsigned int _rqd (rqpointer)
unsigned int rqpointer;
```

_rqputw (Ring queue put word)

```
unsigned int _rqd (rqpointer, data)
unsigned int rqpointer;
unsigned int data;
```

_rsr (Restore segment register)

```
void _rsr (segment_reg, segment_value)
unsigned int segment_reg;
unsigned int segment_value;
```

_send (Send semaphore)

```
int _send (semaphore_id)
unsigned int semaphore_id;
```

_setdvt (Set device timer)

```
int _setdvt (interval, timer_id, queue_id)
unsigned int interval;
unsigned int timer_id;
unsigned int queue_id;
```

_settmr (Set interval timer)

```
int _settmr (interval, option, timer_id)
unsigned int interval;
unsigned int option;
unsigned int timer_id;
```

_signal (Signal)

```
int _signal (target_id, signal_mask)
unsigned int target_id;
unsigned int signal_mask;
```

_sio (Schedule off-level I/O processing)

```
int _sio (parm_list)
struct prm_list *parm_list;
```

_sleep (Wait for interval timer)

```
int _sleep ()
```

_ssr (Save segment register)

```
int _ssr (segment_reg)
unsigned int segment_reg;
```

_stdma (Start DMA transfer)

```
int _stdma (operation, segment_id, transfer_addr, length, bus_address)
struct stdma_type operation;
unsigned int segment_id;
unsigned int transfer_addr;
unsigned int length;
unsigned int *bus_address;
```

_storeb (Store byte)

```
int _storeb (location, data)
unsigned char location[1];
unsigned char data;
```

_storeh (Store halfword)

```
int _storeh (location, data)
unsigned char location[2];
unsigned short data;
```

_storew (Store word)

```
int _storew (location, data)
unsigned char location[4];
unsigned int data;
```

_trcgen (Generate generic trace entry)

```
int _trcgen (buf_adr, trc_id, trc_data_hdr, trc_data)
unsigned int *buf_adr;
unsigned short trc_id;
struct
{
  unsigned short IODN;
  unsigned short IOCN;
  unsigned int hdr_data;
  unsigned int bytes_in_data;
} trc_data_hdr;
unsigned int *trc_data;
```

_trcvrm (Generate trace entry)

```
int _trcvrm (trace_id, trace_entry, trace_length)
unsigned int trace_id;
char *trace_entry;
int trace_length;
```

_uname (Machine identification)

```
unsigned int _uname (uname_area, area_length)
int uname_area;
int area_length;
```

_unpin (Unpin page range)

```
int _unpin (segment_id, first_page, num_pages)
unsigned int segment_id;
int first_page;
int num_pages;
```

_upinio (Unpin I/O Buffer)

```
int _upinio (seg_id, start_addr, num_bytes)
unsigned int seg_id;
unsigned int start_addr;
int num_bytes;
```

_upnccb (Unpin command control block)

```
int _upnccb (segment_id, ccb_address)
unsigned int segment_id;
unsigned int ccb_address;
```

_vrmtrc (Internal VRM trace)

```
void _vrmtrc (trace_entry)
struct trc_entry *trace_entry;
```

_wait, _waitq (Wait for event or queue)

```
int _wait (ecb_mask, clear_ecbs, return_ecb)
unsigned int ecb_mask;
unsigned int clear_ecbs;
unsigned int *return_ecb;
        -or-
int _waitq (queue_id, waitq_qelem)
unsigned int queue_id;
union queue_element *waitq_qelem;
```

_wrblk (Write block)

```
void _wrblk (io_address, real_address)
unsigned int io_address;
unsigned int real_address;
```

_wrwds (Write words)

```
void _wrwds ( io_address, real_address, length)
unsigned int io_address;
unsigned int real_address;
unsigned length;
```

External Variables

The VRM supports the following external variables. These variables are found in page 0 of the VRM. Each variable is defined along with its C language call format.

_model A 32-bit external variable that describes which RT PC machine is being used. A zero indicates an IBM RT PC 6150 and a negative one indicates an IBM RT PC 6151.

```
extern unsigned int _model;
```

_curid A 32-bit unsigned external variable that contains the ID of the process or SLIH currently running.

```
extern unsigned int _curid;
```

_trace A 32-bit unsigned external variable that indicates the event types that are traced in the VRM.

```
extern unsigned int _trace;
```

This 32-bit field is defined as follows:

Bit	Event Type
0-1	Reserved
2	PC Network
3-12	Reserved
13	Asynchronous devices
14	Coprocessor option
15	Reserved
16	Process dispatcher
17-30	Reserved
31	User-defined events.

_time An external array of four elements that provide current time information.

```
extern unsigned int _time[4];
    _time[0] = time of year
    _time[1] = time of IPL
    _time[2] = delta time since IPL
    _time[3] = time of year, extended
```

_sysmem Points to the beginning of an area of 2^{24} bytes that contains system memory for the coprocessor option (segment 14).

```
extern unsigned char _sysmem;
```

_busio Points to the beginning of an area of 2^{24} bytes that contains the I/O bus.

```
extern unsigned char _busio;
```

_busmem Points to the beginning of an area of 2^{24} bytes that contains the I/O bus memory.

```
extern unsigned char _busmem;
```

_floatp Points to the beginning of an area of 2^{24} bytes used by the optional floating point accelerator.

```
extern unsigned char _floatp;
```

_pcbsid A 32-bit external variable that indicates the segment ID of the POST control block in bits 16-31.

```
extern unsigned int _pcbsid;
```

_cputyp A 32-bit external variable with the following bits defined:

- Bit 31 indicates whether the VRM will process page faults requiring I/O (bit 31 = 1) or whether the VRM will abend when a page fault requiring I/O is encountered (bit 31 = 0).
- Bit 30 indicates whether the Advanced Processor Card (bit 30 = 1) or the Processor and Memory Management Card (bit 30 = 0) is configured.
- Bit 29 indicates whether the processor supports loop mode. When bit 29 = 1, the processor supports loop mode and the VRM may optimize loops to run efficiently. When bit 29 = 0, the processor does not support loop mode.

```
extern unsigned int _cputyp;
```

The following variables are used for communication between the VRM trace process and the AIX Operating System file **/dev/trace**. These variables exist in page 0 of the virtual machine.

- **vseq**

  This halfword contains the VRM sequence number. This value is incremented by **_trcvrm** after a trace entry is generated.

- **useq**

  This halfword contains the virtual machine sequence number. This value is incremented by **/dev/trace** after a trace entry is generated.

- **vcnt, ucnt**

  These two 8-bit variables control trace buffer synchronization between the VRM trace process and the **/dev/trace** file of the AIX Operating System. Whenever the trace process sends a buffer to the virtual machine, it increments **vcnt**. When the AIX Operating System processes a trace buffer, it increments **ucnt**. VRM trace entry overrun occurs when:

  ```
  vcnt - ucnt = 2
  ```

External Variables

The VRM supports the following external variables. These variables are found in page 0 of the VRM. Each variable is defined along with its C language call format.

_model A 32-bit external variable that describes which RT PC machine is being used. A zero indicates an IBM RT PC 6150 and a negative one indicates an IBM RT PC 6151.

```
extern unsigned int _model;
```

_curid A 32-bit unsigned external variable that contains the ID of the process or SLIH currently running.

```
extern unsigned int _curid;
```

_trace A 32-bit unsigned external variable that indicates the event types that are traced in the VRM.

```
extern unsigned int _trace;
```

This 32-bit field is defined as follows:

Bit	**Event Type**
0-1	Reserved
2	PC Network
3-12	Reserved
13	Asynchronous devices
14	Coprocessor option
15	Reserved
16	Process dispatcher
17-30	Reserved
31	User-defined events.

_time An external array of four elements that provide current time information.

```
extern unsigned int _time[4];
    _time[0] = time of year
    _time[1] = time of IPL
    _time[2] = delta time since IPL
    _time[3] = time of year, extended
```

_sysmem Points to the beginning of an area of 2^{24} bytes that contains system memory for the coprocessor option (segment 14).

```
extern unsigned char _sysmem;
```

_busio Points to the beginning of an area of 2^{24} bytes that contains the I/O bus.

```
extern unsigned char _busio;
```

_busmem Points to the beginning of an area of 2^{24} bytes that contains the I/O bus memory.

```
extern unsigned char _busmem;
```

_floatp Points to the beginning of an area of 2^{24} bytes used by the optional floating point accelerator.

```
extern unsigned char _floatp;
```

_pcbsid A 32-bit external variable that indicates the segment ID of the POST control block in bits 16-31.

```
extern unsigned int _pcbsid;
```

_cputyp A 32-bit external variable that indicates in bit 31 whether the VRM (bit 31 = 1) or the diagnostic control program (bit 31 = 0) is running and also indicates whether the Advanced Processor Card (bit 30 = 1) or the Processor and Memory Management Card (bit 30 = 0) is configured.

```
extern unsigned int _cputyp;
```

The following variables are used for communication between the VRM trace process and the AIX Operating System file **/dev/trace**. These variables exist in page 0 of the virtual machine.

- **vseq**

  This halfword contains the VRM sequence number. This value is incremented by **_trcvrm** after a trace entry is generated.
- **useq**

  This halfword contains the virtual machine sequence number. This value is incremented by **/dev/trace** after a trace entry is generated.
- **vcnt, ucnt**

  These two 8-bit variables control trace buffer synchronization between the VRM trace process and the **/dev/trace** file of the AIX Operating System. Whenever the trace process sends a buffer to the virtual machine, it increments **vcnt**. When the AIX Operating System processes a trace buffer, it increments **ucnt**. VRM trace entry overrun occurs when:

  ```
  vcnt - ucnt = 2
  ```

Glossary

address. A name, label, or number identifying a location in storage, a device in a system or network, or any other data source.

allocate. To assign a resource, such as a disk file or a diskette file, to perform a specific task.

All Points Addressable (APA) display. A display that allows each pel to be individually addressed. An APA display allows for images to be displayed that are not made up of images predefined in character boxes. Contrast with *character display.*

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

American National Standards Institute (ANSI). An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards.

ANSI. See *American National Standards Institute.*

a.out. An output file produced by default for certain instructions. By default, this file is executable and contains information for the symbolic debugger.

application. (1) A particular task, such as inventory control or accounts receivable. (2) A program or group of programs that apply to a particular business area, such as the Inventory Control or the Accounts Receivable application.

argument. Numbers, letters, or words that expand or change the way commands work.

array. An arrangement of elements in one or more dimensions.

ASCII. See *American National Standard Code for Information Interchange.*

assembler language. A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond directly to the storage and registers of the machine.

asynchronous transmission. In data communications, a method of transmission in which the bits included in a character or block of characters occur during a specific time interval. However, the start of each character or block of characters can occur at any time during this interval. Contrast with *synchronous transmission.*

attribute. A characteristic. For example, the attribute for a displayed field could be blinking.

bad block. A portion of a disk that can never be used reliably.

base register. A general purpose register that the programmer chooses to contain a base address. See also *index register.*

basic addressable unit (BAU). The smallest piece of storage that can be addressed.

BAU. See *basic addressable unit.*

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

bit. Either of the binary digits 0 or 1 used in computers to store information. See also *byte.*

block. (1) A group of records that is recorded or processed as a unit. Same as *physical record.* (2) In data communications, a group of records that is recorded, processed, or sent as a unit. (3) A block is 512 bytes long.

boundary alignment. The position in main storage of a fixed-length field (such as halfword or doubleword) on an integral boundary for that unit of information. For example, a word boundary is a storage address evenly divisible by four.

bps. Bits per second.

branch. In a computer program an instruction that selects one of two or more alternative sets of instructions. A conditional branch occurs only when a specified condition is met.

breakpoint. A place in a computer program, usually specified by an instruction, where execution may be interrupted by external intervention or by a monitor program.

buffer. (1) A temporary storage unit, especially one that accepts information at one rate and delivers it at another rate. (2) An area of storage, temporarily reserved for performing input or output, into which data is read, or from which data is written.

bus. One or more conductors used for transmitting signals or power.

byte. The amount of storage required to represent one character; a byte is 8 bits.

call. (1) To activate a program or procedure at its entry point. Compare with *load.* (2) In data communications, the action necessary in making a connection between two stations on a switched line. by plugging it into a slot in the system unit.

channel. A path along which data passes. Also a device connecting the processor to input and output devices.

character. A letter, digit, or other symbol.

character display. A display that uses a character generator to display predefined character boxes of images (characters) on the screen. This kind of display can not address the screen any less than one character box at a time. Contrast with *All Points Addressable display.*

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character string. A sequence of consecutive characters.

C language. A general-purpose programming language that is the primary language of the AIX Operating System.

color display. A display device capable of displaying more than two colors and the shades produced via the two colors, as opposed to a monochrome display.

command line. The area to the right of a prompt for entering commands or program names.

compile. (1) To translate a program written in a high-level programming language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

compiler. A program that translates instructions written in a high-level programming language into machine language.

concatenate. (1) To link together. (2) To join two character strings.

configuration. The group of machines, devices, and programs that make up a computer system. See also *system customization.*

constant. A data item with a value that does not change. Contrast with *variable.*

control block. A storage area used by a program to hold control information.

control character. A non-printing character that performs formatting functions in a text file.

control program. Part of the AIX Operating System system that determines the order in which basic functions should be performed.

counter. A register or storage location used to accumulate the number of occurrences of an event.

cursor. (1) A movable symbol (such as an underline) on a display, used to indicate to the operator where the next typed character will be placed or where the next action will be directed. In Usability Services, the cursor is called a text cursor. (2) A marker that indicates the current data access location within a file. See also *pointing cursor.*

customize. To describe (to the system) the devices, programs, users, and user defaults for a particular data processing system.

cylinder. All fixed disk or diskette tracks that can be read or written without moving the disk drive or diskette drive read/write mechanism.

daemon. See *daemon process.*

daemon process. A process begun by the root or the root shell that can be stopped only by the root. Daemon processes generally provide services that must be available at all times such as sending data to a printer.

data stream. All information (data and control information) transmitted over a data link.

deadlock. An error condition in which processing cannot continue due to unresolved contention for the use of a resource.

debug. (1) To detect, locate, and correct mistakes in a program. (2) To find the cause of problems detected in software.

debugger. A device used to detect, trace, and eliminate mistakes in computer programs or software.

default. A value that is used when no alternative is specified by the operator.

default value. A value stored in the system that is used when no other value is specified.

device driver. A program that operates a specific device, such as a printer, disk drive, or display.

device manager. Collection of routines that act as an intermediary between device drivers and virtual machines for complex interfaces. For example, supervisor calls from a virtual machine are examined by a device manager and are routed to the appropriate subordinate device drivers.

device name. A name reserved by the system that refers to a specific device.

direct memory access (DMA) device. A component that can read or write to system storage directly, without processor intervention. Two device types are identified: 1) an alternate controller resides on a hardware adapter and 2) a system DMA controller resides on the system planar board. DMA capability permits simultaneous use of input/output devices and the processor.

directory. A type of file containing the names and controlling information for other files or other directories.

disable. To make nonfunctional.

displacement. A positive or negative number that can be added to the contents of a base register to calculate an effective address.

display device. An output unit that gives a visual representation of data.

DMA. See *direct memory access.*

dump. (1) To copy the contents of all or part of storage, usually to an output device.
(2) Data that has been dumped.

EBCDIC. See *extended binary-coded decimal interchange code.*

ECB. See *event control bit.*

effective address. A real storage address that is computed at runtime. The effective address consists of contents of a base register, plus a displacement, plus the contents of an index register if one is present.

emulation. Imitation; for example, when one computer imitates the characteristics of another computer.

enable. To make functional.

entry point. An address or label of the first instruction performed upon entering a computer program, a routine, or a subroutine. A program may have several different entry points, each corresponding to a different function or purpose.

escape character. A character that changes the meaning of the characters that follow. For example the backslash character, used to indicate to the shell that the next character is not intended to have the special meaning normally assigned to it by the shell.

event. The enqueueing of an element.

event control bit (ECB). A bit assigned to each queue to signal the arrival or departure of an element.

exception handler. A set of routines used to detect deadlock conditions or to process abnormal condition processing. This allows the normal execution of processes to be interrupted and resumed.

expression. A representation of a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

external symbol. A symbol that is defined in a file other than the file in which the symbol occurs. An ordinary symbol that represents an external reference.

file. A collection of related data that is stored and retrieved by an assigned name.

file system. The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

first level interrupt handler (FLIH). A routine that receives control of the system as a result of a hardware interrupt. One FLIH is assigned to each of the six interrupt levels.

flag. A modifier that appears on a command line with the command name that defines the action of the command. Flags in the AIX Operating System almost always are preceded by a dash.

FLIH. See *first level interrupt handler.*

font. A family or assortment of characters of a given size and style.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) The pattern which determines how data is recorded.

function. A synonym for procedure. The C language treats a function as a data type that contains executable code and returns a single value to the calling function.

gather. For input/output operations, reading data from noncontiguous memory l locations to write to a device. See *scatter* (antonym).

general purpose register (GPR). An explicitly addressable register that can be used for a variety of purposes (for example, as an accumulator or an index register). II has sixteen 32-bit GPRs. See *register.*

generation. For some remote systems, the translation of configuration information into machine language.

global variable. A symbol defined in one program module, but used in other independently assembled program modules.

GPR. See *general purpose register.*

granularity. The extent to which a larger entity is subdivided. For example, a yard broken into inches has finer granularity than a yard broken into feet.

graphic character. A character that can be displayed or printed.

header record. A record at the beginning of a file that details the sizes, locations, and other information that follows in the file.

hex. See *hexadecimal.*

hexadecimal. Pertaining to a system of numbers to the base sixteen; hexadecimal digits range from 0 (zero) through 9 (nine) and A (ten) through F (fifteen).

high-order. Most significant; leftmost. For example, bit 0 in a register.

history file. (1) A file that exists for each licensed program product supplied by IBM that documents the version, release, and level of the product. Because information is added to this file whenever updates are made to the program product, the history file reflects all changes made to the currently installed version and release of the program product. (2) A file containing a log of system actions and operator responses. (3) A file that displays all versions of a structured file.

IAR. See *instruction address register.*

I/O. See *input/output.*

ID. Identification.

immediate data. Data appearing in an instruction itself (as opposed to the symbolic name of the byte of data). The data is immediately available from the instruction and therefore does not have to be read from memory.

index. (1) A table containing the key value and location of each record in an indexed file. (2) A computer storage position or register, whose contents identify a particular element in a set of elements.

informational message. A message providing information to the operator, that does not require a response.

initial program load (IPL). The process of loading the system programs and preparing the system to run jobs. See *initialize.*

initialize. To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

input-output channel controller (IOCC). A hardware component that supervises communication between the input/output bus and the processor.

input-output code number (IOCN). A value supplied by the virtual machine to a VRM component. This number uniquely identifies the code associated with a component and can be considered a module name.

input-output device number (IODN). A value assigned to a device driver by the virtual machine or to a virtual device by the Virtual Resource Manager. This number uniquely identifies the device regardless of whether it is real or virtual.

input/output subsystem. That part of the VRM comprised of processes and device

managers that provides the mechanisms for data transfer and I/O device management and control.

instruction address register (IAR). A system control register containing the address of the next instruction to be executed. The IAR (sometimes called a "program counter") can be accessed via a supervisor call in supervisor state, but cannot be directly addressed in problem state.

integer. A positive or negative whole number or zero.

interactive. Pertains to an activity involving requests and replies as, for example, between a system user and a program or between two programs.

interface (n). A shared boundary between two or more entities. An interface might be a hardware component to link two devices together or it might be a portion of storage or registers accessed by two or more computer programs.

interrupt. (1) To temporarily stop a process. (2) In data communications, to take an action at a receiving station that causes the sending station to end a transmission. (3) A signal sent by an I/O device to the processor when an error has occurred or when assistance is needed to complete I/O. An interrupt usually suspends execution of the currently executing program.

invoke. To start a command, procedure, or program.

IOCC. See *input-output channel controller.*

IOCN. See *input-output code number.*

IODN. See *input-output device number.*

IPL. See *initial program load.*

kernel. The memory-resident part of the AIX Operating System containing functions needed immediately and frequently. The kernel supervises the input and output, manages and controls the hardware, and schedules the user processes for execution.

key pad. A physical grouping of keys on a keyboard (for example, numeric key pad, and cursor key pad).

load. (1) To move data or programs into storage. (2) To place a diskette into a diskette drive, or a magazine into a diskette magazine drive. (3) To insert paper into a printer.

local. Pertaining to a device directly connected to your system without the use of a communications line. Contrast with *remote.*

log. To record; for example, to log all messages on the system printer. A list of this type is called a log, such as an error log.

logical link control. In RT PC, a protocol process that allows data transfer on a given physical link. A logical link control (LLC) may reside in the operating system or the VRM and is controlled by the block I/O device manager.

loop. A sequence of instructions performed repeatedly until an ending condition is reached.

low-order. Least significant; rightmost. For example, in a 32 bit register (0-31), bit 31 is the low-order bit.

main storage. The part of the processing unit where programs are run.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

menu. A displayed list of items from which an operator can make a selection.

minidisk. A logical division of a fixed disk that may be further subdivided into one or more partitions. See *partition.*

mm. Millimeter.

modem. See *modulator-demodulator.*

modulator-demodulator (modem). A device that converts data from the computer to a signal that can be transmitted on a communications line, and converts the signal received to data for the computer.

module. A discrete programming unit that usually performs a specific task or set of tasks. Modules are subroutines and calling programs that are assembled separately, then linked to make a complete program. See *object module.*

multiprogramming. The processing of two or more programs at the same time.

nonswitched line. A connection between computers or devices that does not have to be established by dialing. Contrast with *switched line.*

nonvolatile random access memory (NVRAM). A portion of random access storage that retains its contents after electrical power to the machine is shut off.

null. Having no value, containing nothing.

null character (NUL). The character hex 00, used to represent the absence of a printed or displayed character.

NVRAM. See *nonvolatile random access memory.*

object module. A set of instructions in machine language. The object module is produced by a compiler or assembler from a subroutine or source module and can be input to the linker. The object module consists of object code. See *module.*

octal. A base eight numbering system.

op code. See *operation code.*

operand. An instruction field that represents data (or the location of data) to be manipulated or operated upon. Not all instructions require an operand field.

operating system. Software that controls the running of programs; in addition, an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

operating system state. One of two virtual machine protection states that run in the processor's unprivileged state. The kernel is the only entity that runs in the operating system state. See *problem state.*

operation code. A numeric code indicating to the processor which operation should be performed.

overflow condition. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

pad. To fill unused positions in a field with dummy data, usually zeros or blanks.

page. (1) A block of instructions, data, or both.

page fault. A program interruption that occurs when a page that is not in memory is referred to by an active page.

page space minidisk. The area on a fixed disk that temporarily stores instructions or data currently being run. See also *minidisk.*

paging. The action of transferring instructions, data, or both between real storage and external page storage.

parameter. Information that the user supplies to a panel, command, or function.

partition. A logical division of a minidisk.

path name. See *full path name* and *relative path name.*

PEL. See *picture element*

picture element (pel). In computer graphics, the smallest element of a display space that can be independently assigned color and intensity.

PID. See *process ID.*

POST. See *power-on self test.*

power-on self test (POST). An internal diagnostic program activated each time the system is turned on.

priority. The relative ranking of items. For example, a job with high priority in the job queue will be run before one with medium or low priority.

privileged instructions. System control instructions that can only run in the processor's privileged state. Privileged instructions generally manipulate virtual machines or the memory manager; they typically are not used by application programmers. See *privileged state.*

privileged state. A hardware protection state in which the processor can run privileged instructions.

problem state. One of two virtual machine protection states that run in the processor's unprivileged state. User-written application programs typically run in the problem state. See *operating system state.*

process. (1) A sequence of actions required to produce a desired result. (2) An entity receiving a portion of the processor's time for executing a program. (3) An activity within the system begun by entering a command, running a shell program, or being started by another process.

process ID (PID). A unique number assigned to a process that is running.

program status block (PSB). A control block that describes a virtual interrupt condition.

protection state. An arrangement for restricting access to or use of all or part of the hardware instruction set. Hardware protection states are privileged state and unprivileged state. Virtual machine protection states are operating system state and problem state.

protocol. In data communications, the rules for transferring data.

protocol procedure. A procedure that implements a function for a device manager. For example, a virtual terminal manager may use a protocol procedure to interpret the meaning of keystrokes.

PSB. See *program status block.*

queue. A line or list formed by items waiting to be processed.

raster array. In computer graphics, a predetermined arrangement of lines that provide uniform coverage of a display space.

reentrant module. A module that allows the same copy of itself to be used concurrently by two or more tasks. Contrast with *serially reusable.*

register. (1) A storage area, in a computer, capable of storing a specified amount of data such as a bit or an address. Each register is 32 bits long. (2) See *general purpose register.*

relative address. An address specified relative to the address of a symbol. When a program is relocated, the addresses themselves will change, but the specification of relative addresses remains the same.

relocatable. A value, expression, or address is relocatable if it does not have to be changed when the program is relocated.

remote. Pertaining to a system or device that is connected to your system through a communications line. Contrast with *local.*

retry. To try the operation again that caused the device error message.

return code. A value that is returned to a subroutine or function to indicate the results of an operation issued by that program.

routine. A set of statements in a program causing the system to perform an operation or a series of related operations.

scatter. For input/output operations, reading data from a device and locating it in noncontiguous memory addresses. See *gather* (antonym).

SDT. See *static debugger trap.*

second level interrupt handler (SLIH). A routine that handles the processing of an interrupt from a specific adapter. An SLIH is called by the first level interrupt handler associated with that interrupt level.

sector. (1) An area on a disk track or a diskette track reserved to record information. (2) The smallest amount of information that can be written to or read from a disk or diskette during a single read or write operation.

segment. A contiguous area of virtual storage allocated to a job or system task. A program segment can be run by itself, even if the whole program is not in main storage.

semaphore. Entity used to control access to system resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

serially reusable module. A module that can be accessed by only one task at a time. Contrast with *reentrant.*

session. (1) The period of time during which programs or devices can communicate with each other. (2) A name for a type of resource that controls local LU's, remote LU's, modes, and attachments.

single-step instruction execution. A method of operating a computer in which each instruction is performed in response to a single manual operation. No sequential execution of instructions is allowed.

SLIH. See *second level interrupt handler.*

special character. A character other than a letter or number. For example; *, +, and % are special characters.

stack. An area in storage that stores temporary register information and returns addresses of subroutines.

stack parameter. A parameter to a VRM subroutine that must be passed on the stack. Usually, VRM subroutines pass parameters in general purpose registers.

stack pointer. A register providing the current location of the stack.

static debugger trap (SDT). A trap instruction placed in a pre-defined point in code invokes the debugger at that point. The trap instruction causes a program check when executed; debugger is invoked as a result of the program check.

string. A linear sequence of entities such as characters or physical elements. Examples of strings are alphabetic string, binary element string, bit string, character string, search string, and symbol string.

subroutine. (1) A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program. (2) A routine that can be part of another routine.

subsystem. A secondary or subordinate system, usually capable of operating independently of, or synchronously with, a controlling system.

supervisor call (SVC). An instruction that interrupts the program being executed and passes control to the supervisor so it can perform a specific service indicated by the instruction.

SVC. See *supervisor call.*

switched line. In data communications, a connection between computers or devices

established by dialing. Contrast with *nonswitched line.*

synchronous. (1) Pertaining to two or more processes that depend upon the occurrences of specific events such as common timing signals.

system customization. A process of specifying the devices, programs, and users for a particular data processing system.

system dump. A printout of storage from all active programs (and their associated data) whenever an error stops the system. Contrast with *task dump.*

thrashing. A condition in which the system is doing so much paging that little useful work can be done.

TLB. See *translation lookaside buffer.*

trace. To record data that provides a history of events occurring in the system.

track. A circular path on the surface of a fixed disk or diskette on which information is magnetically recorded and from which recorded information is read.

translation lookaside buffer (TLB). Hardware that contains the virtual-to-real address mapping.

trap. An unprogrammed, hardware-initiated jump to a specific address. Occurs as a result of an error or certain other conditions.

typematic key. A key that repeats its function multiple times when held down.

type declaration. The specification of the type and, optionally, the length of a variable or function in a specification statement.

typestyle. Characters of a given size, style and design.

unprivileged instruction. Ordinary instructions such as load store, add, and shift typically used by application programs.

unprivileged state. A hardware protection state in which the processor can only run unprivileged instructions. The processor's unprivileged state supports the virtual machine's operating system state and problem state.

update file. A file that adds or revises information in a program product already resident on the VRM minidisk. An update file documents the version, release, and level of updates to be installed and is required for program product update diskettes that use VRM Install update facilities. This file is not used for programs updated from the AIX Operating System.

variable. A name used to represent a data item with a value that can change while the program is running. Contrast with *constant.*

vector. An array of one dimension.

virtual device. A device that appears to the user as a separate entity but is actually a shared portion of a real device. For example, several virtual terminals may exist simultaneously, but only one is active at any given time.

virtual machine. A functional simulation of a computer and its related devices. A virtual machine usually includes an operating system and one or more virtual devices.

virtual machine interface (VMI). A software interface between work stations and the operating system The VMI shields operating system software from hardware changes and low-level interfaces and provides for concurrent execution of multiple virtual machines.

virtual memory manager. Hardware that manages virtual memory by providing translation from a virtual address to a real address.

virtual resource manager (VRM). A set of programs that manage the hardware resources

(main storage, disk storage, display stations, and printers) of the system so that these resources can be used independently of each other.

virtual storage. Addressable space that appears to be real storage. From virtual storage, instructions and data are mapped into real storage locations.

VMI. See *virtual machine interface.*

VRM. See *virtual resource manager.*

word. A contiguous series of 32 bits (four bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

Index

Special Characters

_assign 5-108
_attchq 5-18
_attchs 5-43
_badblk 5-104
_bffree 5-88
_bfget 5-87
_bind 5-76
_bvint 5-21
_canclq 5-22
_change 5-7
_chgmsk 5-109
_chkblk 5-103
_cnltmr 5-68
_copy 5-79
_creatp 5-10
_creatq 5-24
_creats 5-63
_ctldvt 5-69
_cveara 5-44
_dalct 5-107
_defind 5-110
_deque 5-28
_detchq 5-31
_detchs 5-46
_dmamov 5-117
_dmptbl 5-140, 5-143
_dstryq 5-30
_dstrys 5-64
_enq 5-35
_enque 5-35
_erecv 5-91
_errvrm 5-129, 5-132
_initp 5-11
_loadb 5-47
_loadh 5-48
_loadw 5-50
_lsr 5-49
_mallc 5-75
_mapsys 5-113
_mdmchk 5-105
_mfree 5-80
_mvbuff 5-51
_peekq 5-36
_pinpgs 5-52
_post 5-37
_qra 5-45
_qryds 5-116
_qsid 5-53
_queryd 5-115
_queryi 5-12
_querym 5-81
_queryp 5-38
_queryv 5-83
_rdblk 5-89
_rdwds 5-90
_readq 5-39
_recv 5-65
_rqc 5-93
_rqd 5-94
_rqgetw 5-95
_rqputw 5-96
_rsr 5-54
_send 5-66
_setdvt 5-70
_settmr 5-71
_signal 5-15
_sio 5-13
_sleep 5-73
_ssr 5-55
_stdma 5-97
_storeb 5-56
_storeh 5-57
_storew 5-58
_trcgen 5-133, 5-134
_trcvrm 5-135, 5-136
_uname 5-112
_unpin 5-61

_upinio 5-60
_upnccb 5-59
_vrmtrc 5-137
_wait 5-40
_waitq 5-40
_wrblk 5-100
_wrwds 5-101

A

a.out object module format B-1
abend routine 5-137
abnormal termination 3-17
add a fixed disk 6-5
address
  constants B-1
  conversion 5-44
  messages 1-17
advantages of VMI 1-8
AIX Operating System
  error log device driver 5-129
  trace log device driver 5-133, 5-135
allocate memory 5-75, 5-80
alternate DMA device 5-97
architecture, virtual machine 1-6
attach
  device 1-16
  queue 5-17
  segment 5-43

B

bad block
  checking for 5-103
  management 3-32
base address, system bus 2-8
bind module 5-76
block
  read 5-89
  write 5-100
breakpoint 8-8
broadcast virtual interrupt 5-21
buffer, allocating from buffer pool 5-87
bus mask, changing 5-109

C

C language 4-5
  external variables D-21
  subroutines D-1
calling register conventions B-8
cancel
  I/O 1-16, 4-48
  interval timer 5-68
  queue 5-22
case sensitivity, for debugger commands 8-9
CCB format 4-68
chained control blocks 5-34
change
  attributes 5-7
  bus mask 5-109
channels, direct memory access 3-10
characteristics, process management 3-13
check
  for bad blocks 5-103
  parameter 5-24
close
  a minidisk 6-11
cntl + alt key sequences C-1
code
  function 1-10
  supervisor call instruction 4-7
command
  control blocks (CCB) 4-67
  element 4-69
  header 4-68
  input/output 1-16
common
  queue element format 5-33
component IDs, reserved 5-142
condition status register 5-138
constant pool B-1, B-8, B-11
constants, address B-1
control
  device timer 5-69
  registers, virtual machine 2-4

section of TOC module B-4
control blocks
device 8-31
module 8-30
path 8-41
process 8-25
queue 8-39
second-level interrupt handler 8-28
semapahore 8-23
timer requests 8-24
convert
effective address to real address 5-44
virtual address to real address 5-45
coprocessor
assigning a device to 5-108
DMA 3-27
copy
a module 5-78
create
minidisk 6-7, 6-15
process 5-10
queue 5-23
semaphore 5-63

D

daemon, error 5-129
data
access operations 6-25
allocating device-dependent 5-107
corruption, prevention of 5-43, 5-46
string 8-9
deadlock 5-14
debug SVC 4-77
debugger 3-6
breakpoint 8-8
commands
alter 8-15
Ascii 8-16
back 8-17
break 8-18
breaks 8-19
clear 8-20
cntl-alt-pad4 key sequence 8-8
display 8-47
display formatted VRM control block 8-21
display variables 8-84
ditto 8-48
Ebcdic 8-49
find 8-50
go 8-53
help 8-13
Ior 8-54
IOW 8-55
loop 8-56
map 8-57
next 8-59
origin 8-60
print screen 8-13
quit 8-61
reset 8-62
restore 8-63
screen 8-64
set 8-66
show 8-67
sregs 8-68
st 8-71
start 8-72
stc 8-73
step 8-74
sth 8-75
stop 8-76
stops 8-77
swap to/from RS-232 port 8-78
touch 8-81
trace 8-82
translate lookaside buffer (TLB) 8-79
Xlate (translate) 8-85
defined 8-6
expressions 8-10
key sequence C-2
supervisor call (SVC) 8-8
trap 8-8
variables 8-10
define
code 1-15, 4-49
device 1-15, 4-51, 5-110
device structure 3-10, 4-52
minidisk characteristics 6-17

delete a minidisk 6-13
denormal number 7-14
depth count, interrupt 5-17
dequeue element 5-25
destroy
  queue 5-30
  semaphore 5-64
detach
  queue 5-31
  segment 5-46
device
  allocating a buffer from the buffer pool 5-87
  allocating device-dependent data 5-107
  characteristics section of DDS 4-53
  control blocks 8-31
  creation of 3-9
  definition 1-15
  direct memory access support 3-27
  directory 5-110
  identifier (DID) 3-7
  management
    allocate a buffer 5-87
    allocate device-dependent data 5-107
    assign device to coprocessor 5-108
    assigning an IODN 5-110
    change bus mask 5-109
    defining a device 5-110
    free allocated buffer 5-88
    machine identification 5-112
    map system memory 5-113
    query device ID 5-115
    query device status 5-116
    set up region mode 5-117
    start DMA transfer 5-97
  query 1-17
  query device structure 3-9
  status management 3-9
  timer 3-29
device driver 1-5, 3-6
device management
  check for bad blocks 5-103
  minidisk bad blocks 5-104
  minidisk manager check 5-105
  read block 5-89
  read words 5-90
  receive data from bus memory 5-91
  write block 5-100
  write words 5-101
direct memory access (DMA) 3-27
  alternate 5-97
  system DMA 5-97
directory, device 5-110
disable timer 4-20
disk space allocation 3-31
dispatcher
  process 1-5
  VRM process 5-137
divide by zero floating-point operation 7-7
dump key sequence C-1
dump table 5-140

E

effective address 1-13
  conversion 5-44
element, command 4-67
enable timer 4-20
enqueue element 5-32
error
  entry 5-129
  floating-point 7-10
  log 5-129
  process, VRM 5-119
errrecvr command 5-119
event
  control bit 3-18, 5-23, 5-37
  monitor subroutines 5-128
exception handler 3-16, 5-7
exceptions, floating-point 7-10
execution level
  interrupts 2-14
  priority mechanism 2-14
export modules 5-76
external
  page table 3-21
  symbol dictionary B-4
  variables D-21

F

first-level interrupt handler (FLIH) 5-137
floating point
  errors 7-10
  exceptions 7-7, 7-10
  hardware indicator 2-7
  operations
    allocate register sets 4-10
    free register sets 4-12
  register sets 2-8
  registers, saving B-8
  services 7-4
frame pointer B-8
free allocated buffer 5-88
from ID 5-17
function code 1-10
functions, key sequences for C-1

G

general purpose registers 1-6
general wait 5-40
generating virtual interrupts 5-28
granularity, timer 5-71

H

handling interrupts 1-6
hardware characteristics 4-54
header
  command control block 4-67
  TOC module B-3

I

identification, virtual machine 2-10
immediate messages 1-17
import modules 5-76
index of supervisor call instructions A-1
inexact result floating-point operation 7-7
initialize
  process 5-11
input query device structure 4-58
input/output
  cancellation 1-16
  code number 4-49
  code number (IOCN) 1-15, 3-7
  device number (IODN) 1-15, 3-7
  execution 1-16
  initiation 1-16
  notification 1-16
  subsystem 1-14, 3-5, 5-21, 5-22, 5-25, 5-32, 5-39
  VRM procedures
    ring queue create 5-92
    ring queue delete 5-94
    ring queue get word 5-95
    ring queue put word 5-96
installing code into the VRM 1-15
Institute of Electrical and Electronic Engineers (IEEE) standard 7-4
instruction
  address register 1-6, 5-138
  condition status 1-6
  interrupt control status 1-6
  priority execution mechanism 2-14
  privileged and unprivileged 1-9
  system control 4-5
internal trace, VRM 5-137
interrupt 2-14
  control status register 5-138
  control status, virtual register 2-8
  handler 1-6, 5-23
  priority mechanism 2-14
  processing cycle 2-24
  program check 2-20
  request buffer 2-8
  virtual machine 2-14

interval timer 5-71
inverted page table 3-20
IPL
  record 4-79
  record format 4-81
  status of virtual machine 1-7
  virtual machine 4-78

K

key sequences C-1

L

load
  byte 5-47
  halfword 5-48
  program status instruction 1-11, 2-25, 4-6
  segment registers 5-49
  word 5-50
locking resources 3-28
logging out due to impending shutdown 4-86
loop 8-56
  spin 5-67
  work 3-16

M

machine
  communications interrupt 2-14
  control procedures
    IPL virtual machine 4-78
    machine identification 4-82
    query virtual machine 4-83, 5-83
    re-IPL VRM 4-85
    terminate virtual machine 5-85
  identification 5-112
  state at IPL 1-7
managing physical resources 3-10
map
  displaying a VRM module 8-57
  system memory 5-113
mapped page ranges 2-28
master dump table 5-140
memory
  management 1-11, 3-20
  map 3-21, 3-23
  units 1-12
  virtual 3-6
memory management
  attach segment 5-43
  convert effective address to real address 5-44
  convert virtual address to real address 5-45
  detach segment 5-46
  load byte 5-47
  load halfword 5-48
  load segment registers 5-49
  load word 5-50
  move buffer 5-51
  pin page range 5-52
  query segment ID 5-53
  restore segment registers 5-54
  save segment registers 5-55
  store byte 5-56
  store halfword 5-57
  store word 5-58
  unpin command control block 5-59
  unpin I/O buffer 5-60
  unpin page range 5-61
minidisk 3-6
  bad blocks 5-104
  command header format 6-25
  creation of 3-31
  manager 6-4
  manager check 5-105
  map 3-32
  maximum number supported 4-34
module control block 8-30
module identifier (MID) 3-7, 3-30, 5-81
module map, displaying 8-57
move buffer 5-51
multiprogramming 1-5
mutual exclusion 3-28

N

naming conventions 3-6
nested locks 3-28
non-volatile random access memory (NVRAM) 4-89
not-a-number, floating-point operation 7-9

O

object module, TOC B-1
off-level processing 5-13
offset
  NVRAM 4-89
  page 5-44
  segment 5-53
open
  minidisk 6-9
operation
  completion information 4-59, 6-8, 6-16
overflow floating-point operation 7-7
overrun
  conditions 5-13
  count 2-19

P

pad4 (numeric key) 8-8
page
  boundary protection 2-27
  of memory 1-12
  offset 5-44
  virtual memory space 3-32
paging subsystem 5-43
parameter
  check 5-24
  stack 5-5
passthrough SVCs 4-7
path 5-38
  control blocks 8-41
  identifier 3-7
  limit 5-23
peek at queue element 5-36
physical resources, management of 3-10
pin page range 5-52
pinned memory 5-23
planar 5-97
pointer
  references 8-12
  registers for C-compatible routines B-8
pool, constant B-1, B-11
post control block segment identifier 2-6
post event control bit 5-37
print a debugger screen 8-13
priority mechanism, interrupt and instruction execution 2-14
privileged instructions 1-9
problem state 2-10
process 1-5
  control block 8-25
  control flow 3-15
  dispatcher 5-137
  identifier 5-10
  management
    change attributes 5-7
    create process 5-10
    initialize process 5-11
    query ID 5-12
    signal process 5-14
  management characteristics 3-13
  priority 2-8, 3-15
  states, VRM 3-14
processor type indicator 2-7
program
  check interrupt 2-14, 2-20
  check status 2-20
  management
    allocate memory 5-75, 5-80
    bind module 5-76
    copy module 5-78
    query module ID 5-81
  status 2-15
  status block locations 2-16
  status blocks 2-15
  status blocks (PSB) 4-70
  status, load 1-11

protection of segments 2-27

Q

query
- a fixed disk for minidisks 6-22
- a minidisk 6-19
- attached path 5-38
- device 1-17
- device identifier 5-115
- device status 5-116
- device structure 3-9, 4-58, 4-60, 5-27
- identifier 5-12
- module ID 5-81
- segment ID 5-53
- virtual machines 4-83, 5-83

queue 5-40
- control blocks 8-39
- element format 5-27
- element formats 5-33
- elements 5-21, 5-25, 5-32, 5-39
- management 3-18
  - attach queue 5-17
  - broadcast virtual interrupt 5-21
  - cancel queue 5-22
  - create queue 5-23
  - dequeue element 5-25
  - destroy queue 5-30
  - detach queue 5-31
  - enqueue element 5-32
  - peek at queue element 5-36
  - post event control bit 5-37
  - query attached path 5-38
  - read queue 5-39
  - virtual interrupts 5-28
  - wait for queue or event 5-40

quiet not-a-number 7-9

R

re-IPL
- CPU key sequence C-1
- virtual machine key sequence C-1
- VRM 4-85

read
- block 5-89
- queue element 5-39
- words 5-90

read-only
- page 2-28
- TOC module data B-2

real
- address conversion 5-44, 5-45
- machine 1-8
- memory map of the VRM 3-21
- time of IPL 2-11

receive
- data from bus memory 5-91
- from semaphore 3-28, 5-65

region mode, setting up 5-117
- device status 5-117

register
- conventions, C language B-8
- general purpose 1-6
- instruction address 1-6
- sets, floating point 2-8
- system control 1-6
- virtual 2-4
- virtual machine control 1-6

relocation dictionary B-4

reserved
- component IDs 5-142
- debugger variables 8-10

resources, locking 3-28

restore segment registers 5-54

ring queue
- create 5-92
- delete 5-94
- get word 5-95
- put word 5-96

S

save segment registers 5-55
schedule I/O 5-13
scratch data B-8
screen, printing from debugger 8-13
second-level interrupt handler control block 8-28
segment
- attach 5-43
- detach 5-46
- identifier 3-22
- identifier, post control block 2-6
- identifiers 1-14
- information from debugger 8-45
- offset 5-53
- register conventions
  - virtual machine 3-24
  - VRM device driver 3-25
  - VRM process 3-25
- register format 3-22
- register number 5-53
- registers 1-14
- sharing 2-29

semaphore 3-6, 3-28
- debugger control blocks 8-23
- management
  - create semaphore 5-63
  - destroy semaphore 5-64
  - receive from semaphore 5-65
  - send to semaphore 5-66

send
- command program status block 4-66
- command SVC 4-63
- to semaphore 3-28, 5-66

sequence
- key C-1
- number, VRM and virtual machine 2-7

serial number, software readable 4-82
serialization 3-28
set
- device timer 5-70
- interval timer 5-71

shutdown, impending 4-86
signal 5-14
- completion of an event 5-67
- mask 5-14

signalling not-a-number 7-9
single-step 8-74
SLIH
- control blocks 8-28

specific wait 5-40
spin loops 5-67
stack
- frame B-8
- overflow B-11
- parameter 5-5
- pointer B-8

start
- DMA transfer 5-97
- I/O 1-16, 4-67

static breakpoints 8-8
status
- blocks, program 2-15
- flags for execution level interrupts 2-18
- of virtual machine at IPL 1-7
- program 2-15
- word, machine communications 2-22

store
- byte 5-56
- halfword 5-57
- word 5-58

string
- data 8-9
- storage B-5

structure
- input query device 4-58

subroutine linkage B-8
subroutines, C language D-1
subsystem, input/output 3-5
supervisor call handler 3-5
supervisor call instructions 4-7
- allocate floating-point register sets 4-10
- attach device 4-47
- cancel input/output 4-48
- change segment size 4-24
- clear segment register 4-25
- copy segment 4-26
- create segment 4-27
- debug a virtual machine 4-77, 8-8
- define code 4-49

define device 4-51
destroy segment 4-29
detach device 4-57
discard page range 4-30
dispatch 4-11
free floating-point register sets 4-12
index of A-1
IPL virtual machine 4-78
load segment register 4-32
machine identification 4-82
map page range 4-34
no operation 4-13
passthrough SVCs 4-7
pin page range 4-37
post 4-14
protect pages 4-38
purge page range 4-39
purge segments 4-41, 4-62
query device 4-58
query floating-point register sets 4-15
query page protect 4-43
query virtual machine 4-83
re-IPL VRM 4-85
read from NVRAM 4-89
return 4-16
return from interrupt 4-17
ring queue get word 4-61
send address message 4-73
send command 4-63, 6-4
send immediate message 4-74
set interrupt 4-18
set message receive 4-75
set timer 4-19
soft interrupt 4-21
start input/output 4-67
terminate virtual machine 4-86
terminate virtual terminal 4-86
unmap page range 4-44
unpin page range 4-45
update VRM 4-87
virtual machine wait 4-22
write data to NVRAM 4-90
SVC code 3-5, 4-7
synchronization 3-6
system
abend routine 5-137
calls B-8
control instructions 4-5
control registers 1-6, 5-138
DMA device 5-97

T

table-of-contents object module format B-1
table, page 3-20
target queue 5-32
terminate
process 5-14
virtual machine 4-86, 5-85
time
interval granularity 5-70
of day 3-29
of day, extended 2-11
timer 3-6, 3-29, 5-14
granularity 5-71
information 8-43
interrupts 3-16
management
cancel interval timer 5-68
control device timer 5-69
set device timer 5-70
set interval timer 5-71
wait for interval timer 5-73
request control block 8-24
source register 4-20
source, virtual machine 2-11
status register, virtual machine 2-11
unit 3-29
to ID 5-17
TOC object module
definition B-2
ESD entries B-5
format B-1
header B-3
loader table section B-5
read-only section B-5
read/write section B-5
RLD entries B-6
string storage B-5
trace

buffer synch flag 2-7
internal VRM 5-137
point 8-18
process 5-123, 5-133, 5-135
process, VRM 5-119
translation
control word (TCW) 3-26, 5-97
lookaside buffer (TLB) 8-79
trap-disabled floating-point exceptions 7-10
trap-enabled floating-point exceptions 7-10
trap, debugger 8-8

U

underflow floating-point operation 7-7
uninitialized state 3-15
unique serial number 4-82
units of virtual memory 1-12
unpin
command control block 5-59
I/O buffer 5-60
page range 5-61
unprivileged instructions 1-9
unsolicited interrupts 5-21
update
NVRAM key sequence C-3
VRM 4-87
utilization information 8-38

V

variables
external D-21
reserved debugger 8-10
virtual
address 1-14
address conversion 5-45
devices 4-51
interrupt control status (VICS) register 2-8
interrupts 5-21, 5-22, 5-25, 5-28, 5-32, 5-39
broadcast 5-21
machine 1-8
communications 1-17
control registers 2-4
information 8-44
interface (VMI) 1-3, 2-3, 3-3, 3-6
interface, advantages of 1-8
interrupts 2-14
sequence number 2-7
memory 3-6
memory addressing 1-14
memory map of VRM 3-23
page information 8-46
page space 3-32
register 1-10
registers 2-4
resource manager (VRM) 1-4, 3-3, 3-14
VRM
event subroutines 5-128
process states 3-14

W

wait
for interval timer 5-73
for queue or event 5-40
general 5-40
specific 5-40
warning 2-27
words
read 5-90
write 5-101
work loop 3-16
write
block 5-100
verify 6-7, 6-15
words 5-101

IBM RT PC Programming Family

Reader's Comment Form

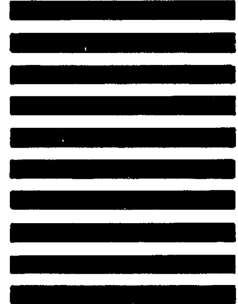
IBM RT PC Virtual Resource Manager Programming Reference

SC23-0816

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

IBM RT PC VRM Programming Reference	SC23-0816
Book Title	Order No.

Book Evaluation Form

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y N Is the purpose of this book clear?

Y N Is the table of contents helpful?

Y N Is the index complete?

Y N Are the chapter titles and other headings meaningful?

Y N Is the information organized appropriately?

Y N Is the information accurate?

Y N Is the information complete?

Y N Is only necessary information included?

Y N Does the book refer you to the appropriate places for more information?

Y N Are terms defined clearly?

Y N Are terms used consistently?

Y N Are the abbreviations and acronyms understandable?

Y N Are the examples clear?

Y N Are examples provided where they are needed?

Y N Are the illustrations clear?

Y N Is the format of the book (shape, size, color) effective?

Other Comments

What could we do to make this book or the entire set of books for this system easier to use?

Optional Information

Your name

Company name

Street address

City, State, ZIP

No postage necessary if mailed in the U.S.A.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape Fold and tape

Cut or Fold Along Line

Tape Please Do Not Staple Tape

© IBM Corp. 1987
All rights reserved.

International Business
Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Printed in the
United States of America

SC23-0816-0