RT–11 System Internals Manual

Order Number AA–PD6NA–TC

August 1991

This manual describes the internals of the RT–11 operating system. It is most useful to system programmers, but it provides valuable background information to application programmers as well.

Revision/Update Information:	This is a new manual for programmers; it is a complete revision of the information previously located in Chapters 1 through 6 of the *RT–11 Software Support Manual*.
Operating System:	RT–11 Version 5.6

Digital Equipment Corporation
Maynard, Massachusetts

First Printing, August 1991

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

Any software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license. No responsibility is assumed for the use or reliability of software or equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227–7013.

© Digital Equipment Corporation 1991
All rights reserved. Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation: CTS–300, DDCMP, DECnet, DECUS, DECwriter, DIBOL, MASSBUS, MicroPDP–11, Micro/RSX, PDP, Professional, Q-bus, RSTS, RSX, RT–11, RTEM–11, UNIBUS, VMS, VT, and the DIGITAL Logo.

S1782

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

Preface xv

Chapter 1 System Components and Memory Layouts

1.1 Static Components 1–1
1.1.1 Trap Vectors 1–2
1.1.2 System Communication Area 1–2
1.1.2.1 User Error Byte 1–6
1.1.2.2 Job Status Word (JSW) 1–8
1.1.3 Interrupt Vectors 1–10
1.1.4 Background Job 1–12
1.1.4.1 RUN Command 1–12
1.1.4.2 R Command 1–15
1.1.4.3 VRUN Command 1–15
1.1.4.4 V Command 1–15
1.1.4.5 .CHAIN Request 1–15
1.1.5 Resident Monitor (RMON) 1–17
1.1.6 System Device Handler 1–18
1.1.7 I/O Page 1–19
1.2 Dynamic Components 1–20
1.2.1 Device Handlers and Free Space 1–21
1.2.2 Foreground and System Jobs 1–22
1.2.2.1 Differences Between Foreground and Background Jobs 1–23
1.2.2.2 FRUN Command 1–24
1.2.2.3 Starting Foreground and System Jobs 1–26
1.2.2.4 Foreground Stack 1–27
1.2.2.5 Foreground Impure Area 1–28
1.2.3 User Service Routine (USR) 1–29
1.2.3.1 Structure 1–29
1.2.3.2 Execution 1–31
1.2.3.3 USR Swapping with Unmapped Systems 1–32
1.2.4 Keyboard Monitor (KMON) 1–36
1.2.4.1 DCL Command Processing 1–36
1.2.4.2 CCL Command Processing 1–37
1.2.4.3 Adding New Commands Through UCL 1–38
1.2.4.4 User Commands First (UCF) Feature 1–39
1.2.4.5 Passing Commands Through the Chain Area 1–40
1.2.4.6 Character Case Handling 1–40
1.3 Determining Components Size 1–41

1.3.1 Size of the USR 1–41
1.3.2 Size of KMON 1–41
1.3.3 Size of RMON 1–41
1.3.4 Size of Device Handlers 1–41

Chapter 2 Resident Monitor (RMON)

2.1 Terminal Service 2–1
2.1.1 Output Ring Buffer 2–1
2.1.1.1 Storing a Character in the Output Ring Buffer 2–2
2.1.1.2 Removing a Character from the Output Ring Buffer 2–3
2.1.2 Input Ring Buffer 2–3
2.1.2.1 Storing a Character in the Input Ring Buffer 2–4
2.1.2.2 Removing a Character from the Input Ring Buffer 2–5
2.1.3 High-Speed Ring Buffer 2–5
2.1.4 Terminal I/O Limitations 2–6
2.1.5 Control Functions 2–6
2.1.5.1 CTRL/C 2–6
2.1.5.2 CTRL/O 2–7
2.1.5.3 CTRL/S and CTRL/Q 2–7
2.1.5.4 CTRL/B, CTRL/F, and CTRL/X 2–7
2.1.6 SET Options Status Word, .TCFDF 2–8
2.2 Clock Support and Timer Service 2–9
2.2.1 SB Systems Without Timer Service 2–9
2.2.2 Systems with Timer Service 2–9
2.3 Queued I/O System 2–11
2.3.1 I/O Channel Format 2–11
2.3.2 I/O Queue (.QELDF) 2–13
2.3.3 Completion Queue 2–18
2.3.3.1 .SYNCH Considerations 2–20
2.3.4 Flow of Events in I/O Processing 2–20
2.3.4.1 Issuing the Request 2–21
2.3.4.2 Queuing the Request 2–21
2.3.4.3 Performing the I/O Transfer 2–23
2.3.4.4 Completing the I/O Request 2–23
2.4 Scheduling in Multi-Job Systems 2–24
2.4.1 User and System State 2–25
2.4.1.1 Switching to System State Asynchronously 2–26
2.4.1.2 Switching to System State Synchronously 2–28
2.4.1.3 Returning to User State 2–29
2.4.2 Context Switching in Multi-Job Systems 2–30
2.4.3 Blocking Conditions 2–31
2.4.3.1 How the Monitor Blocks a Job 2–32
2.4.3.2 $SYSWT Monitor Routine 2–33
2.4.3.3 How the Monitor Unblocks a Job 2–35

2.4.4 Scheduler Operations 2–35
2.4.4.1 How the Monitor Requests a Scheduling Pass 2–35
2.4.4.2 Characteristics of a Runnable Job 2–35
2.4.4.3 $RQTSW Monitor Routine 2–35
2.4.4.4 How the Scheduler Works 2–36
2.4.5 Implications for Completion Routines 2–36
2.5 System Jobs 2–37
2.5.1 Characteristics 2–37
2.5.2 Logical Names 2–37
2.5.3 Job Number 2–37
2.5.4 Priority 2–37
2.5.5 Design Considerations 2–38
2.5.5.1 Scheduling Considerations 2–38
2.5.5.2 Space Considerations 2–38
2.5.6 Programmed Requests 2–39
2.5.7 Message Handling 2–39
2.5.8 Monitor Commands 2–40
2.5.8.1 SRUN and FRUN Commands 2–40
2.5.8.2 LOAD and UNLOAD Commands 2–40
2.5.8.3 SUSPEND and RESUME Commands 2–41
2.5.8.4 SHOW JOBS Command 2–41
2.5.9 Interchanging Between System Jobs 2–41
2.6 Data Structures 2–41
2.6.1 Fixed Offsets 2–41
2.6.1.1 Configuration Word ($CNFG1) 2–47
2.6.1.2 Low-Memory Protection Bitmap ($LOWMA) 2–48
2.6.1.3 DCL and IND Indirect File Status Word ($STATW) 2–50
2.6.1.4 Second Configuration Word ($CNFG2) 2–50
2.6.1.5 System Generation Features Word ($SYSGE) 2–52
2.6.1.6 Transparent Spooler (SPOOL) Status Word ($SPSTA) 2–53
2.6.1.7 IND Control Status Byte ($INDST) 2–54
2.6.1.8 Pointer to the End of the Impure Area ($IMPLO) 2–55
2.6.1.9 Default Editor ($PROGD) 2–55
2.6.1.10 Default FORTRAN Compiler ($PROGF) 2–55
2.6.1.11 Job Slots on the System ($JOBS) 2–55
2.6.1.12 Third Configuration Word ($CNFG3) 2–56
2.6.1.13 KMON/UCF Interface Word (CLI.FL/CLI.TY) 2–57
2.6.2 Impure Area 2–57
2.6.3 Device Tables 2–63
2.6.3.1 $PNAME Table 2–64
2.6.3.2 $STAT Table 2–65
2.6.3.3 $DVREC Table 2–65
2.6.3.4 $ENTRY Table 2–65
2.6.3.5 $DVSIZ Table 2–65
2.6.3.6 $HSIZE Table 2–66

2.6.3.7 $UNAM1 And $UNAM2 Tables 2–66
2.6.3.8 $OWNER Table 2–66
2.6.3.9 $PNAM2 Table 2–67
2.6.3.10 Adding a Device to the Tables 2–67

Chapter 3 Memory Mapping

3.1 Default System Memory Layouts 3–2
3.1.1 Mapped System Memory Layout 3–2
3.2 Default Job Mapping 3–5
3.2.1 Privileged Mapping Environment 3–7
3.2.1.1 Privileged Background Job 3–8
3.2.1.2 Privileged Foreground or System Job 3–8
3.2.2 Virtual Mapping Environment 3–8
3.2.2.1 Selecting Virtual Mapping 3–11
3.2.2.2 A Virtual Background Job 3–11
3.2.2.3 A Virtual Foreground or System Job 3–12
3.2.3 Completely Virtual Mapping Environment 3–13
3.2.3.1 V/VRUN Keyboard Command 3–14
3.2.3.2 Nonseparated I-D Space Completely Virtual Environment 3–15
3.2.3.2.1 Automatic Job Running 3–15
3.2.3.2.2 Stopping Automatic Job Running 3–15
3.2.3.3 Separated I-D Space Completely Virtual Environment 3–16
3.2.3.4 Chaining in the Completely Virtual Environment 3–17
3.2.3.5 Completely Virtual Background Job 3–18
3.2.3.6 Completely Virtual Foreground/System Job 3–18
3.3 Kernel, User, and Supervisor Processor Modes 3–23
3.4 How Programs Control Mapping 3–29
3.4.1 Physical Address Regions 3–29
3.4.1.1 Local Regions 3–30
3.4.1.2 Global Regions 3–30
3.4.1.3 Handler Global Regions 3–31
3.4.2 Virtual Address Windows 3–31
3.4.2.1 The Static Window 3–32
3.4.2.2 Dynamic Windows 3–33
3.4.3 Program's Logical Address Space (PLAS) 3–33
3.4.4 Mapping and Context Switches with Virtual and Privileged Jobs 3–33
3.5 Introduction to the Extended Memory Programmed Requests 3–35
3.6 Extended Memory Data Structures 3–36
3.6.1 Region Definition Block 3–37
3.6.1.1 Region Status Word (R.GSTS) 3–38
3.6.1.2 .RDBDF Macro 3–39
3.6.1.3 .RDBBK Macro 3–40
3.6.2 Region Control Block 3–40

3.6.3 Global Region Control Block 3–41
3.6.3.1 Global Region Control Block Status Byte (GR.STA) 3–42
3.6.4 Window Definition Block 3–42
3.6.4.1 Window Status Word (W.NSTS) 3–45
3.6.4.2 .WDBDF Macro 3–46
3.6.4.3 .WDBBK Macro 3–47
3.6.5 Window Control Block 3–48
3.6.6 Mapping Context Area (MCA) Region 3–50
3.6.7 I/O Queue Element 3–54
3.6.8 Free Memory List 3–54
3.7 Flow of Control Within Each Programmed Request 3–54
3.7.1 Defining the Memory Mapping Context: .CMAP, .GCMAP, .MSDS 3–54
3.7.2 Creating a Local Region: .CRRG 3–56
3.7.3 Creating and Attaching to a Global Region 3–56
3.7.4 Attaching to an Existing Global Region 3–57
3.7.5 Detaching from a Global Region 3–57
3.7.6 Creating a Window: .CRAW 3–58
3.7.7 Mapping a window to a Region: .MAP 3–59
3.7.8 Getting the Mapping Status: .GMCX 3–60
3.7.9 Unmapping a Window: .UNMAP 3–61
3.7.10 Eliminating a Local Region: .ELRG 3–62
3.7.11 Eliminating a Global Region 3–62
3.7.12 Automatic Global Region Elimination 3–63
3.7.13 Eliminating a Window: .ELAW 3–63
3.8 Typical Extended Memory Applications 3–64
3.8.1 Completely Virtual Environment (VRUN) 3–64
3.8.2 Minimizing Low-Memory Usage 3–64
3.8.3 Large Buffers or Arrays in Extended Memory 3–66
3.8.4 Multi-User Program 3–67
3.8.5 Work Space in Extended Memory 3–68
3.8.5.1 Enabling the XM Feature of the .SETTOP Programmed Request 3–68
3.8.5.2 Program and Virtual High Limits and the Next Free Address 3–69
3.8.5.3 Non-XM .SETTOP 3–69
3.8.5.4 XM .SETTOP 3–71
3.8.5.5 XM .SETTOP and Privileged Jobs 3–72
3.8.5.6 XM .SETTOP and Virtual Jobs 3–74
3.8.5.7 .SETTOP and Completely Virtual Jobs 3–76
3.8.5.8 Summary of .SETTOP Action 3–78
3.8.6 Directly Modifying Address Page Registers (APRs) 3–83
3.9 Hardware Concepts 3–84
3.9.1 Virtual and Physical Addresses with Extended Memory Hardware 3–84
3.9.2 Circumventing the 32K-Word Address Limitation 3–84
3.9.3 Concept of Pages 3–85
3.9.4 Relocation 3–86

3.9.5 Active Page Register (APR) 3–87
3.9.5.1 Page Address Register (PAR) 3–88
3.9.5.2 Page Descriptor Register (PDR) 3–89
3.9.6 Converting a 16-Bit Address to an 18- Or 22-Bit Address 3–92
3.9.7 Status Registers 3–94
3.10 Restrictions and Design Implications 3–94
3.10.1 PAR1 Restriction 3–94
3.10.2 PAR2 Restriction 3–95
3.10.3 Programmed Requests 3–95
3.10.4 Synchronous System Traps 3–95
3.10.4.1 TRAP, BPT, And IOT Instructions 3–96
3.10.4.2 Traps to 4 and 10, and FPU Traps 3–97
3.10.4.3 Memory Management Faults 3–97
3.10.4.4 Memory Parity Errors 3–97
3.11 Debugging an Extended Memory Application 3–98
3.12 Extended Memory Example Program 3–98
3.13 Procedure to Create and Map a Global Region 3–98

Chapter 4 Multiterminal Feature

4.1 Components of a Multiterminal System 4–1
4.2 Hardware Background Information 4–2
4.3 What is the Console Terminal? 4–4
4.4 Connecting Handlers to Terminal Lines 4–5
4.4.1 Monitor Support 4–7
4.4.1.1 Terminal Hooks Data Structure, THOOKS 4–7
4.4.1.2 Terminal Output Enable Routine, MTOENB 4–8
4.4.1.3 Terminal Line Break Routine, MTYBRK 4–9
4.4.1.4 Terminal Modem Control Routine, MTYCTL 4–9
4.4.1.5 Terminal Modem Status Routine, MTYSTA 4–10
4.4.2 Connecting a Serial Interface Printer Handler (LS) 4–11
4.4.3 Connecting a Serial Communications Handler (XL) 4–12
4.5 Using Two or More Terminals 4–12
4.5.1 A Separate Terminal for Each Job 4–12
4.5.2 Multiterminal Applications 4–12
4.6 Introduction to Multiterminal Programmed Requests 4–13
4.7 Multiterminal Data Structures 4–14
4.7.1 Terminal Control Block (TCB) 4–14
4.7.1.1 Format 4–14
4.7.1.2 Patching a TCB 4–24
4.7.2 Asynchronous Terminal Status (AST) Word 4–25
4.8 Using the Multiterminal Programmed Requests 4–26
4.8.1 Attaching a Terminal: .MTATCH 4–26
4.8.2 Getting Terminal Status: .MTGET 4–27
4.8.3 Setting Terminal Characteristics: .MTSET 4–27

4.8.4 Getting Characters: .MTIN 4–28
4.8.5 Printing Characters: .MTOUT 4–28
4.8.6 Printing a Line: .MTPRNT 4–29
4.8.7 Resetting CTRL/O: .MTRCTO 4–29
4.8.8 Getting System Status: .MTSTAT 4–29
4.8.9 Detaching a Terminal: .MTDTCH 4–30
4.9 The Console as a Special Case 4–30
4.10 Interrupt Service 4–30
4.10.1 Local Terminals 4–31
4.10.2 Remote Terminals 4–31
4.11 Polling Routines 4–32
4.11.1 Time-Out Routine for DL Terminals 4–32
4.11.2 DZ Remote Line Polling Routine 4–32
4.12 Restrictions 4–33
4.13 Debugging a Multiterminal Application 4–33
4.14 Multiterminal Example Program 4–33
4.15 Using Two or More Terminals Without the Multiterminal Feature 4–38
4.15.1 A Video Console Terminal and a Hardcopy Printing Terminal 4–38
4.15.1.1 The Video Terminal Is the Boot-Time Console 4–38
4.15.1.2 The Hardcopy Terminal Is the Boot-Time Console 4–39
4.15.2 Switching the Console Terminal 4–40

Chapter 5 Interrupt Service Routines

5.1 Noninterrupt Programmed I/O 5–1
5.2 Interrupt-Driven I/O 5–2
5.2.1 How an Interrupt Works 5–3
5.2.2 Device and Processor Priorities 5–3
5.2.3 Processor Status (PS) Word 5–4
5.3 In-line Interrupt Service Routines Versus Device Handlers 5–4
5.4 How to Plan an Interrupt Service Routine 5–6
5.4.1 Get to Know Your Device 5–8
5.4.2 Study the Structure of an Interrupt Service Routine 5–10
5.4.3 Study the Skeleton Interrupt Service Routine 5–10
5.4.4 Think About the Requirements of Your Program 5–10
5.4.5 Prepare a Flowchart of Your Program 5–10
5.4.6 Write the Code 5–10
5.4.7 Test and Debug the Program 5–11
5.5 Structure of an Interrupt Service Routine 5–11
5.5.1 Protecting Vectors: .PROTECT 5–11
5.5.2 Setting up the Interrupt Vector 5–11
5.5.3 Stopping Cleanly: .DEVICE 5–13
5.5.4 Lowering Processor Priority: .INTEN 5–13
5.5.5 Issuing Programmed Requests: .SYNCH 5–14
5.5.6 Running at Fork Level: .FORK 5–15

5.5.7 Summary of .INTEN, .FORK, and .SYNCH Action 5–17
5.5.8 Exiting from Interrupt Service 5–17
5.6 Skeleton Outline of an Interrupt Service Routine 5–17
5.7 Interrupt Service Routines in Mapped Systems 5–20

Appendix A The System Definition Library (SYSTEM.MLB)

A.1 Introduction A–1
A.2 Macros That Define EMT Request Block Structures A–2
A.3 Macros That Define Data Areas A–5
A.4 Miscellaneous Macros A–5
A.5 Macros That Are Available in SYSTEM.MLB A–6

Appendix B Software Simulation Of The Console Terminal Hardware

B.1 Introduction B–1
B.2 Single Terminal Software Simulation B–1
B.2.1 Data Structures B–1
B.2.2 Setting Up the Terminal Input Software Simulation B–2
B.2.2.1 Providing an Interrupt Source B–3
B.2.2.2 Operation B–3
B.2.3 Setting Up the Terminal Output Software Simulation B–4
B.2.3.1 Operation B–6
B.3 Multiterminal Software Simulation B–7
B.3.1 Data Structures B–7
B.3.2 Setting Up the Multiterminal Software Simulation B–8
B.3.2.1 Identify the Console Terminal B–10
B.3.2.2 Operation B–11

Index

Figures

1–1 Trap Vector Area 1–3
1–2 System Communication Area 1–4
1–3 Interrupt Vector Area 1–13
1–4 Background Job 1–14
1–5 RUN Command 1–16
1–6 Resident Monitor (RMON) 1–17
1–7 System Device Handler 1–19
1–8 I/O Page 1–20
1–9 SB System with Two Loaded Handlers 1–22
1–10 SB System with One Handler Unloaded 1–23
1–11 SB System with Both Handlers Unloaded 1–24
1–12 Foreground Job 1–25

1–13 FRUN Command 1–27
1–14 FB System 1–28
1–15 USR 1–30
1–16 Keyboard Monitor (KMON) 1–37
2–1 Output Ring Buffer 2–2
2–2 Storing Characters in the Output Ring Buffer 2–3
2–3 Input Ring Buffer 2–4
2–4 Storing Characters in the Input Ring Buffer 2–5
2–5 Components of the Queued I/O System 2–12
2–6 I/O Queue with Three Available Elements 2–15
2–7 I/O Queue with Two Available Elements 2–16
2–8 I/O Queue with One Available Element 2–17
2–9 I/O Queue when One Element Is Returned 2–17
2–10 I/O Queue when Two Elements are Returned 2–18
2–11 Device Handler Queue when a New Element is Added 2–19
2–12 Device Handler/RMON Relationship 2–23
2–13 Interrupts and Execution States 2–27
2–14 $SYSWT Monitor Routine 2–34
2–15 $OWNER Entry 2–66
3–1 Default Mapping at Bootstrap Time 3–3
3–2 XB and XM System Memory Layout 3–4
3–3 Privileged Background Job 3–9
3–4 Privileged Foreground or System Job 3–10
3–5 Virtual Background Job 3–12
3–6 Virtual Background Job Mapping into the Static Region 3–13
3–7 Virtual Foreground or System Job 3–14
3–8 Completely Virtual Background Job 3–19
3–9 Completely Virtual Background Job with Separated I & D Address Spaces 3–20
3–10 Completely Virtual Background Job with Extended Memory Overlays 3–21
3–11 Completely Virtual Background Job with Separated I & D Address Spaces and Extended Memory Overlays 3–22
3–12 Completely Virtual Foreground/System Job 3–24
3–13 Completely Virtual Foreground/System Job With Separated I & D Address Spaces . . 3–25
3–14 Completely Virtual Foreground/System Job With Extended Memory Overlays 3–26
3–15 Completely Virtual Foreground/System Job With Separated I & D Address Spaces and Extended Memory Overlays 3–27
3–16 Processor Status Word and Active Page Registers 3–28
3–17 Mapping the Same Virtual Addresses to Different Physical Locations 3–28
3–18 Virtual Address Space and Three Windows 3–32
3–19 Region Definition Block 3–37
3–20 Region Control Block 3–40
3–21 Global Region Control Block 3–41
3–22 Window Definition Block 3–43
3–23 .WDBBK Macro Example 3–48
3–24 Window Control Block 3–49

3–25 Virtual Background Job with Extended Memory Overlays 3–65
3–26 Virtual Background Job with an Array in Extended Memory 3–66
3–27 Multi-User Virtual Background Program 3–67
3–28 Program and Virtual High Limits, and the Next Free Address 3–70
3–29 Gaps in Virtual Address Space 3–73
3–30 Privileged Background Job with .SETTOP 3–74
3–31 Virtual Background Job with .SETTOP 3–76
3–32 Virtual Foreground or System Job with .SETTOP 3–77
3–33 Background .SETTOP Summary 3–79
3–34 Foreground .SETTOP Summary 3–81
3–35 Completely Virtual Job .SETTOP Summary 3–82
3–36 Virtual and Physical Addresses with Extended Memory Hardware 3–85
3–37 Program Segments Sharing Virtual Address Space 3–86
3–38 4K-Word Pages 3–87
3–39 Smaller Pages 3–88
3–40 Relocation by Program 3–89
3–41 Relocation by Page 3–90
3–42 Active Page Register (APR) 3–90
3–43 Correspondence Between Pages and Active Page Registers 3–91
3–44 Page Address Register (PAR) 3–91
3–45 Page Descriptor Register (PDR) 3–91
3–46 Virtual Address 3–93
3–47 MMU Address Conversion (Detail) 3–93
3–48 Creating and Mapping a Global Region 3–99
4–1 Interfaces and Physical and Logical Unit Numbers 4–3
4–2 Multiterminal Handler Hooks Connection 4–6
4–3 Format of the Terminal Control Block (TCB) 4–15
4–4 Multiterminal Example Program 4–33
4–5 Patch for Procedure 4 4–39
5–1 RT–11 Priority Structure 5–3
5–2 Processor Status (PS) Word 5–5
5–3 In-line Interrupt Service Routines and Device Handlers 5–7
5–4 Summary of Registers in Interrupt Service Routine Macro Calls 5–18
5–5 Skeleton Interrupt Service Routine 5–19
5–6 Kernel and Privileged Mapping 5–21
5–7 Interrupt Service Routine Mapping Error 5–22
5–8 PAR1 Restriction for Interrupt Service Routines 5–23

Tables

1–1 Trap Vectors 1–2
1–2 System Communication Area (.SYCDF) 1–5
1–3 User Error Byte ($USRRB), .UEBDF 1–7
1–4 Job Status Word ($JSW) (.JSWDF) 1–8
1–5 Interrupt Vectors 1–10
1–6 Monitor P-sects 1–18
2–1 SET Options Status Word ($TTCNF) 2–8
2–2 Timer Queue Element Format (.QTIDF) 2–10
2–3 I/O Channel Description (.CHNDF) 2–13
2–4 Channel Status Word (CSW), .CSWDF 2–13
2–5 I/O Queue Element 2–14
2–6 Completion Queue Element Format (.QCMDF) 2–20
2–7 Synch Queue Element (.QSYDF) 2–20
2–8 Values of the interrupt Level Counter (INTLVL) 2–26
2–9 Job's Stack After $INTEN 2–28
2–10 Job's Stack After $ENSYS 2–29
2–11 Blocking Conditions 2–31
2–12 RMON Fixed Offsets 2–42
2–13 Configuration Word (.CFIDF), Offset 300 2–47
2–14 Low-Memory Bitmap 2–48
2–15 DCL and IND File Status Word (.STWDF), Offset 366 2–50
2–16 Extension Configuration Word (.CF2DF), Offset 370 2–51
2–17 Determining Processor Bus Structure 2–52
2–18 System Generation Features Word (.SGNDF), Offset 372 2–53
2–19 Transparent Spooler Status Word (.SPLDF), Offset 414 2–53
2–20 Spool Operation Type Field (SP$OPR) 2–54
2–21 Spool Flag Page Support Field (SP$NFL) 2–54
2–22 IND Control Status Byte (.INDDF), Offset 417 2–54
2–23 Default Editor (.PGMDF), Offset 452 2–55
2–24 Default FORTRAN Compiler, Offset 453 2–55
2–25 Job Slots on the System, Offset 455 2–56
2–26 Third Configuration Word (.CF3DF), Offset 466 2–56
2–27 CLI.FL (.CLIDF) 2–57
2–28 CLI.TY (.CLIDF) 2–57
2–29 Impure Area (.IMPDF) 2–58
2–30 Job State Word Bits, Offset 0 (.ISTDF) 2–61
2–31 Job Blocking Bits, Offset 36 (.IBKDF) 2–62
2–32 Change Mapping Context (I.CMAP) Word Bits (.CMPDF) 2–63
3–1 Memory Mapping Terms 3–1
3–2 Initial Contents of Kernel and User APRs 3–2
3–3 Characteristics of Privileged Jobs 3–5
3–4 Characteristics of Virtual Jobs 3–6
3–5 Characteristics of Completely Virtual Jobs 3–6

3–6 Summary of Activities for a Program in an Extended Memory System 3–35
3–7 Region Definition Block (.RDBDF) 3–38
3–8 Region Status Word (.RDBDF) 3–38
3–9 Region Control Block (.RCBDF) 3–41
3–10 Global Region Control Block (.GRBDF) 3–42
3–11 Global Region Control Block Status Byte, GR.STA 3–42
3–12 Window Definition Block (.WDBDF) 3–43
3–13 Correspondence Between Active Page Registers and Virtual Addresses 3–44
3–14 Window Status Word, W.NSTS 3–45
3–15 Window Control Block (.WCBDF) 3–50
3–16 XM .SETTOP and the Completely Virtual Job Environment 3–78
3–17 Background .SETTOP Summary 3–80
3–18 Summary of Foreground Job High Limit After .SETTOP 3–82
3–19 Completely Virtual Job .SETTOP Summary 3–82
3–20 Synchronous System Traps and Their Vectors 3–96
4–1 Terminal Hooks Data Structure, THOOKS 4–8
4–2 Summary of Activities for a Program in a Multiterminal System 4–13
4–3 Contents of the Terminal Control Block (TCB), .TCBDF 4–16
4–4 Terminal Configuration Word, T.CNFG (.TCFDF) 4–19
4–5 Second Terminal Configuration Word, T.CNF2 (.TC2DF) 4–22
4–6 Terminal Status Word, T.STAT (.TSTDF) 4–24
4–7 Asynchronous Terminal Status (AST) Word (.TASDF) 4–25
5–1 Synch Block (.QSYDF) 5–15
5–2 Fork Block 5–16
5–3 Summary of Interrupt Service Routine Macro Calls 5–17
A–1 SYSTEM.MLB Macros and Their Related Programmed Requests A–6

Preface

Document Structure

This manual is divided into five chapters and two appendixes:

- Chapter 1, System Components and Memory Layouts, presents a somewhat idealized view of the operating system and introduces the various components without providing great detail.
- Chapter 2, Resident Monitor (RMON), describes the functions of the Resident Monitor (RMON) that are generally common to all RT–11 systems.
- Chapter 3, Memory Mapping, provides the information you need to understand and manipulate memory in a mapped monitor system.
- Chapter 4, Multiterminal Feature, describes the RT–11 multiterminal feature, providing background information on the hardware and describing the data structures of a multiterminal system.
- Chapter 5, Interrupt Service Routines, describes the ways a program (as opposed to a device handler) can transfer data between memory and a peripheral device.
- Appendix A, The System Definition Library (SYSTEM.MLB), describes the macro library file SYSTEM.MLB. SYSTEM.MLB contains macros that define symbolic names, values, and offsets for the data structures of the RT–11 system.
- Appendix B, Software Simulation of the Console Terminal Hardware, describes an interface into the terminal handler that lets you simulate the console hardware under all monitors.

Audience

This manual is most useful to system programmers, but it provides valuable background information to application programmers as well.

Conventions

The following conventions are used in this manual.

Convention	Meaning

Black print	In examples, black print indicates output lines or prompting characters that the system displays. For example:
	`.BACKUP/INITIALIZE DL0:F*.FOR DU1:WRK`
	`Mount output volume in DU1:; continue? Y`

Red print	In examples, red print indicates user input.
Braces ({ })	In command syntax examples, braces enclose options that are mutually exclusive. You can choose only one option from the group of options that appear in braces.
Brackets ([ ])	Square brackets in a format line represent optional parameters, qualifiers, or values, unless otherwise specified.
Lowercase characters	In command syntax examples, lowercase characters represent elements of a command for which you supply a value. For example: `DELETE filespec`
UPPERCASE characters	In command syntax examples, uppercase characters represent elements of a command that should be entered exactly as given.
RET	RET in examples represents the RETURN key. Unless the manual indicates otherwise, terminate all commands or command strings by pressing RET.
CTRL/*x*	CTRL/*x* indicates a control key sequence. While pressing the key labeled Ctrl, press another key. For example: CTRL/C

Associated Documents

Basic Books

- *Introduction to RT–11*
- *Guide to RT–11 Documentation*
- *PDP–11 Keypad Editor User's Guide*
- *PDP–11 Keypad Editor Reference Card*
- *RT–11 Commands Manual*
- *RT–11 Mini-Reference Manual*
- *RT–11 Master Index*
- *RT–11 System Message Manual*
- *RT–11 System Release Notes*

Installation Specific Books

- *RT–11 Automatic Installation Guide*
- *RT–11 Installation Guide*
- *RT–11 System Generation Guide*

Programmer Oriented Books

- *RT–11 IND Control Files Manual*
- *RT–11 System Utilities Manual*
- *RT–11 System Macro Library Manual*
- *RT–11 System Subroutine Library Manual*
- *RT–11 Device Handlers Manual*
- *RT–11 Volume and File Formats Manual*
- *DBG–11 Symbolic Debugger User's Guide*

Chapter 1
System Components and Memory Layouts

This chapter presents a somewhat idealized view of the operating system and introduces the various components without providing great detail. You should look further in this manual and in the *RT–11 Device Handlers Manual* and *RT–11 Volume and File Formats Manual* for more information on particular components or concepts that are introduced here. Consult the *RT–11 Master Index* for page references.

RT–11 distributes a library of system data structures, SYSTEM.MLB, described in Appendix A. Throughout this chapter (and the rest of the manual), the symbols that define data structures and the elements of those structures are as defined in SYSTEM.MLB.

This chapter introduces the components of the RT–11 system that can be memory resident. It provides maps of physical and virtual memory that show where the components are located, and it indicates how their positions can change dynamically. This chapter covers components divided into two groups: static components, which have a relatively fixed position in memory, and dynamic components, whose locations are changeable.

The location of system components in physical memory is determined by hardware and the monitor. In mapped systems, components reside at virtual addresses. A virtual address does not necessarily (or probably) correspond to the same physical address. In mapped systems, a component that is described as residing at location 40, for example, probably does not reside at physical location 40, but rather at a virtual location to which it is mapped. The location of components in mapped systems is described more fully in Chapter 3.

The components are arranged to leave the most space available for user programs and to be flexible. Flexibility is obtained by positioning the components after determining the total amount of memory at bootstrap time. Normally, you do not have to take any special steps to move a program that runs under a particular monitor from one PDP–11 computer to another.

1.1 Static Components

The static components have fixed locations in memory. Their actual addresses vary from one PDP–11 computer to the next. Unmapped addresses correspond directly to physical memory addresses. Mapped (virtual) addresses do not necessarily correspond directly to physical memory addresses.

The static components or areas are as follows:

- Trap vectors
- System communication area

- Interrupt vectors
- Background job
- Resident Monitor (RMON)
- System device handler
- I/O page

1.1.1 Trap Vectors

Table 1–1 shows the memory locations from 0 to 36, an area that contains the trap vectors. A plus sign (+) marks the locations that are reserved for use by RT–11. You should not attempt to modify these locations; a bitmap protects them each time you load a program. An asterisk (*) marks the locations that your programs can use. Figure 1–1 is a summary of the trap vector area information.

Table 1–1: Trap Vectors

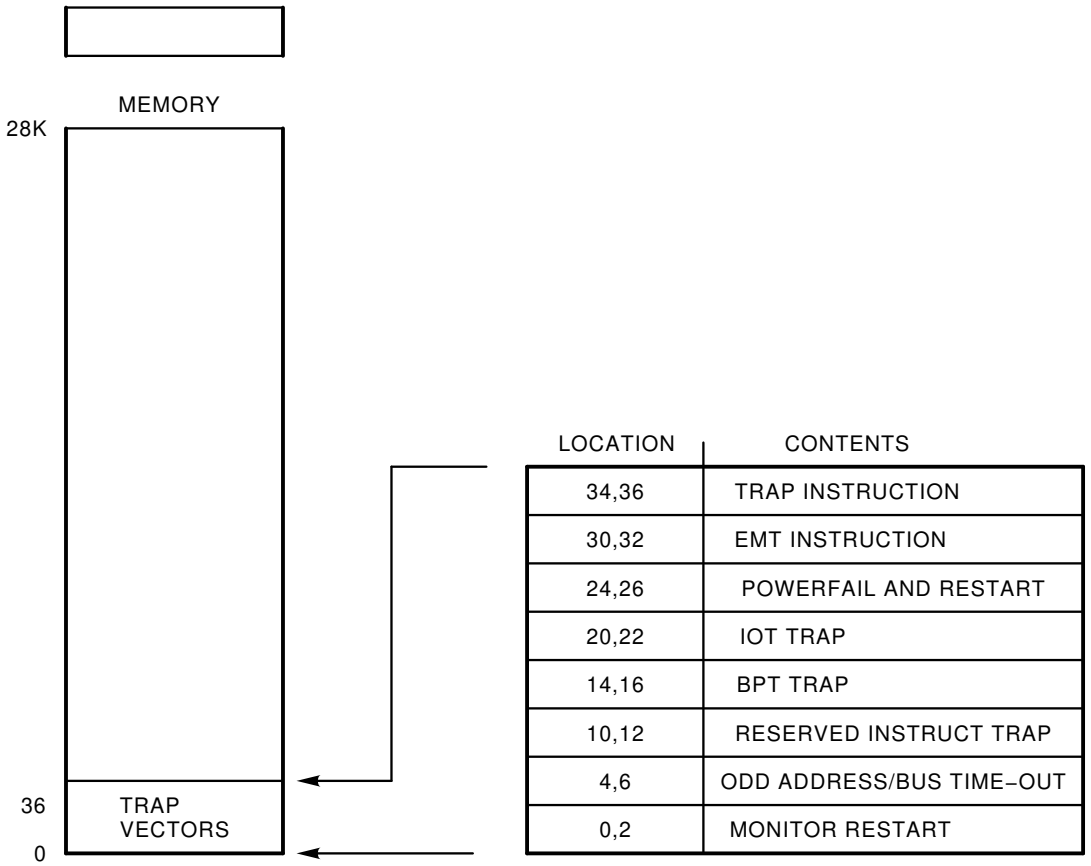
Location	Contents
0,2+	Monitor restart, executes the .EXIT request and returns control to the monitor (has additional uses in mapped systems).
4,6+	Odd address and bus time-out trap; RT–11 sets this to point to its internal trap handler.
10,12+	Reserved instruction trap; RT–11 sets this to point to its internal trap handler.
14,16*	BPT (breakpoint trap), T-bit trap (used by debugging utility programs).
20,22*	IOT, input/output trap.
24,26*	Powerfail and restart trap. Your programs can use this location unless you included support for powerfail restart through system generation. If your system includes the powerfail restart feature, locations 24 and 26 are reserved for use by RT–11.
30,32+	EMT, emulator trap; RT–11 uses this for programmed requests.
34,36*	TRAP instruction. Note that you cannot use the TRAP instruction in assembly language subroutines linked with FORTRAN, DIBOL, or BASIC programs; these languages use the TRAP instruction for internal error reporting.

1.1.2 System Communication Area

The memory locations from 40 through 57 are called the **system communication area**. This area holds information about the job currently executing, as well as certain information normally used only by the monitor.

The diagram in Figure 1–2 is a summary of the system communication area information. Table 1–2 describes the contents of each location. The symbols in the table are from SYSTEM.MLB.

Figure 1–1: Trap Vector Area



LOCATION	CONTENTS
34,36	TRAP INSTRUCTION
30,32	EMT INSTRUCTION
24,26	POWERFAIL AND RESTART
20,22	IOT TRAP
14,16	BPT TRAP
10,12	RESERVED INSTRUCT TRAP
4,6	ODD ADDRESS/BUS TIME–OUT
0,2	MONITOR RESTART

Figure 1–2: System Communication Area

Table 1–2: System Communication Area (.SYCDF)

Location	Symbol	Contents
40,41	$USRPC	Start address of job. When you link a file to create an RT–11 executable image, the linker sets the word at address 40 in the program's file to the starting address of the program. This word is loaded into memory location 40 at run time. When a foreground job executes, the FRUN processor relocates this word to contain the actual starting address of the program.
42,43	$USRSP	Initial value of stack pointer. If the user program does not set this value with an .ASECT directive, the value defaults to 1000 or to the top of the program's absolute section, whichever is larger. You can use the linker /B:n option to set the initial value of the background job's stack pointer. If a foreground program does not specify a stack pointer in this word (by using an .ASECT directive), the FRUN processor allocates a default stack of 128_{10} bytes immediately below the program, and the initial stack pointer value is 1000, relative to the base of the foreground job.
44,45	$JSW	Job Status Word (JSW). This is a flag word for the monitor. The monitor maintains some of the bits itself, and your program can set or clear others. See Section 1.1.2.2 for more information on the JSW.
46,47	$UFLOA	USR load address. This word is normally 0, but you can set it in the file or at run time to any valid word address in your program. If this word is 0, the USR loads in its default location through an address contained in offset 266 ($USRLC of .FIXDF) of RMON. If this word is not 0, the USR loads at the address it specifies, unless the USR is set NOSWAP. This location is cleared by an exit to KMON (via .EXIT, CTRL/C, or fatal error).
50,51	$USRTO	High-memory address. In this word the monitor maintains the highest address your program can use. The linker sets this word initially to the high-limit value. You can modify it by using the .SETTOP programmed request. Your program must never modify this word directly. In mapped systems, locations 50 and 51 in the file contain the address that is the top of the root section plus the low memory (/O) overlays. In memory, locations 50 and 51 contain the same value unless the program issues a .SETTOP. In this case, these locations contain the highest available virtual address (see Section 3.8.5.6).

Table 1–2 (Cont.): System Communication Area (.SYCDF)

Location	Symbol	Contents
52	$ERRBY	EMT error code. If a monitor request results in an error, the code number of the error is always returned in byte 52 in memory, and the carry bit is set. Each monitor call has its own set of possible errors. Byte 52 in the job's file has a different meaning (see *RT–11 Volume and File Formats Manual*). $ERRBY is defined in the .ERRDF macro in SYSTEM.MLB. **NOTE** Always address location 52 as a byte, never as a word, since byte 53 has a separate function.
53	$USRRB	User program error code (USRRB). If a user program encounters errors during execution, it indicates the error by using this byte in memory. See Section 1.1.2.1 for more information about this byte. See *RT–11 Volume and File Formats Manual* for its meaning in the job's file.
54,55	$SYPTR	Address of the beginning of RMON. RT–11 always loads the monitor into the highest available memory locations of low (rather than extended) memory; this word in memory points to its first location. Never alter this word—doing so causes RT–11 to malfunction.
56	$TTFIL	Fill character (7-bit ASCII). Some very old terminals require fill (null) characters after printing certain characters. Byte 56 in memory should contain the ASCII 7-bit representation of the character after which fills are required. See *RT–11 Volume and File Formats Manual* for the meaning of this bit in the job's file.
57	$TTNFI	Fill count. This byte in memory specifies the number of fill characters that are required. The number of characters is determined by hardware. If bytes 57 and 56 are 0, no fill is required. See *RT–11 Volume and File Formats Manual* for the meaning of this byte in the job's file.

)

1.1.2.1 User Error Byte

KMON examines the *user error byte* when a program terminates. If your program has reported a significant error in this byte, KMON can abort any indirect command files in use. This prevents spurious results from occurring if subsequent commands in the indirect file depend on the successful completion of all prior commands.

A program can exit in one of the following states:

- Success
- Warning
- Error

- Fatal
- Unconditionally fatal error

The program status is **success** when the execution of the program is free of errors.

The **warning** status indicates that warning messages occurred, but the program ran to completion.

The **error** status indicates that a user error occurred and the program did not run to completion. This level is also used by RT–11 system utility programs when they produce an output file even though it may contain errors. For example, a compiler can use the error level to indicate that an object file was produced, but the source program contains errors. Under these conditions, execution of the object file will not be successful if the module containing the error is encountered.

The **fatal** status indicates that the program did not produce any usable output, and any command or operation depending on this program output may not execute properly. This type of error can result when a resource needed by the program to complete execution is not available—for example, insufficient memory space to assemble or compile an application program.

The **unconditionally fatal** status indicates that not only has an operation completely failed, but that the integrity of the monitor itself is questionable.

Utility programs and KMON always set the user error byte to reflect the result of each monitor command you issue. By default, indirect command files abort when there has been any monitor command error. However, by issuing the SET ERROR NONE command, you guarantee that indirect command files will continue to execute unless they encounter an unconditionally fatal error. Only unconditionally fatal errors that indicate problems within KMON itself abort indirect files at the SET ERROR NONE level. Table 1–3 shows the bits of byte 53, their status, and the status code printed by the RT–11 system utility program messages.

Table 1–3: User Error Byte ($USRRB), .UEBDF

Bit	Mask	Symbol	Status	RT–11 Message
0	1	SUCCS$	Success	?PROG-I-text, or none
1	2	WARN$	Warning	?PROG-W-text
2	4	ERROR$	Error	?PROG-E-text
3	10	FATAL$	Fatal	?PROG-F-text
4	20	UNCON$	Unconditional	?PROG-U-text

Bits 5 through 7 of the user error byte are reserved for Digital's future use; do not use them in your programs. Programs should never clear byte 53 and should set it only through a BISB instruction, as the following example shows. If more than one bit is set at any given time, the highest bit is the one that RT–11 recognizes.

```
        .LIBRARY "SRC:SYSTEM"
        .MCALL   .UEBDF .SYCDF
        .UDBDF
        .SYCDF
        .
        .
        .
ERROR:  BISB    #ERROR$,@#$USRRB        ;SET ERROR STATUS
        CLR     R0                      ;HARD EXIT
        .EXIT
```

Note that this byte is meaningful only for KMON and for background jobs. It was designed to be used by system utility programs and language processors, which run as background jobs. A foreground job can set it, but that action has no effect on the system.

1.1.2.2 Job Status Word (JSW)

Bytes 44 and 45 make up the *Job Status Word* (JSW). Table 1–4 shows the meanings of the bits in this word. The bits marked with a dollar sign ($) can be set by the monitor. The bits marked with an asterisk (*) can be set by a user program during execution. Bits marked with a plus sign (+) are set at load time from the program image. Note that some bits can be set at both load and run time. Unused bits are reserved for future use by Digital.

Table 1–4: Job Status Word ($JSW) (.JSWDF)

Bit	Symbol	Meaning When Set
15		Reserved.
14+*	TTLC$	Lowercase bit. Disables automatic conversion of typed lowercase to uppercase characters. EDIT sets it when you type the EL command.
13+*	RSTRT$	Reenter bit. Indicates that a program can be restarted from the terminal when you type the REENTER command.
12+*	TTSPC$	Special mode terminal bit. Indicates that the job is in a special keyboard mode of input. Refer to the explanation of the .TTYIN and .TTINR programmed requests in the *RT–11 System Macro Library Manual* for details.
11+*	CHNIF$	Pass line to KMON bit. Indicates, when a program exits, that the program is passing a command line to KMON. This action causes any open indirect file to abort. The command line should be stored in the CHAIN information area, locations 500 through 776. R0 must be cleared before exiting. Refer to the example program for .EXIT in the *RT–11 System Macro Library Manual*. This bit is ignored with foreground or system jobs.

Table 1–4 (Cont.): Job Status Word ($JSW) (.JSWDF)

Bit	Symbol	Meaning When Set
10$+	VIRT$	Virtual image bit (mapped systems only). Indicates that the job to be loaded is a virtual job. You must set this bit yourself in the executable file before you attempt to run the program. Do this at assembly time by using an .ASECT directive and modifying the $JSW, or before run time by patching this location in the file. See Chapter 3 for more information on virtual jobs.
9+*	OVLY$	Overlay bit. This bit is set by the linker if the user program uses the linker overlay feature. You can set OVLY$ in the file if you want to have channel 17 left open on the program's file (even if the program is not overlaid.)
8$+	CHAIN$	CHAIN bit. This bit can be used in two ways. If it is set in a job's save image, the monitor always loads words 500 through 776 from the save file when the job is started. (These words are normally used to pass parameters from one job to another across a .CHAIN or to pass a copy of the command string if the program was invoked by RUN or CCL.) The monitor sets this bit when loading the job only if the job was actually entered with a .CHAIN.
7$	VBGEX$	VBGEXE is executing in the background (mapped systems only). Bit is set by VBGEXE and should not be changed by a job. Futher, the VBGEX$ bit in the $JSX word (the $JSW extension) has a different meaning.
6+*	TCBIT$	Inhibit terminal wait bit. Inhibits the job from entering a console terminal wait state. For more information, refer to the sections concerning .TTYIN, .TTINR, .TTYOUT, and .TTOUTR in the *RT–11 System Macro Library Manual*.
5+*	SPXIT$	Special chain exit bit. If set when a program exits, text in the chain area, locations 510 to 777, is passed to KMON and appended to the command buffer. R0 must be cleared before exiting. This does not abort an open indirect file. Refer to bit 11, above. If you pass multiple command lines, any line containing the @ indirect file command must be the last line of the series.
4+*	EDIT$	Disable single-line editor bit. Setting this bit disables all single-line editor functions.

Table 1–4 (Cont.): Job Status Word ($JSW) (.JSWDF)

Bit	Symbol	Meaning When Set

3+*	GTLIN$	Nonterminating .GTLIN bit. When GTLIN$ of the $JSW is set and your program encounters a CTRL/C in an indirect command file, the .GTLIN request collects subsequent lines from the terminal. If you then clear GTLIN$, the next line collected by the .GTLIN request is the CTRL/C in the indirect command file; this causes the program to terminate. Further input will come from the indirect command file if there are any more lines in it. The LINK, DUP, SIPP, SLP, QUEMAN, SRCCOM, and LIBR utilities make use of this feature. To activate it in an indirect file, put an up arrow (^) followed by a C on a line by itself in the file. This causes the utilities to accept the response from the terminal instead of taking it directly from the file.
		
		The following indirect file shows how to obtain a response from the terminal:
		
		`RUN LINK`
		`TEST,TEST=MOD1,LIB/I`
		`^C`
		
		All further input to the linker will come from the terminal, as a result of the CTRL/C in the indirect command file.

2$*	VRUNV$	Job was run by the V or VRUN command (mapped systems only). If set, a .CHAIN request will run the chained-to program in the completely virtual environment and leave VRUNV$ set.
0-1		Reserved.

1.1.3 Interrupt Vectors

Table 1–5 shows the locations in the low-memory area that are reserved for interrupt vectors. (RT–11 does not support all the devices in the table.) Figure 1–3 shows how the interrupt vector area relates to the rest of memory.

Table 1–5: Interrupt Vectors

Location	Contents
60,62	DL11: Console terminal input
64,66	DL11: Console terminal output
70,72	PC11: Paper tape reader
74,76	PC11: Paper tape punch
100,102	KW11–L: Line clock
104,106	KW11–P: Programmable clock
110,112	Reserved

Table 1–5 (Cont.): Interrupt Vectors

Location	Contents
114,116	Memory system errors: parity, cache, and uncorrectable ECC errors
120,122	XY11: X/Y Plotter
124,126	DR11-B: DMA interface
130,132	AD01: Analog to digital subsystem
134,136	AFC11: Analog input subsystem
140,142	AA11: Digital to analog subsystem
144,146	AA11: (requires two vectors)
150,152	MSCP device number 1
154,156	MSCP device number 0
160,162	RL11/RLV11: RL01/RL02 Disk cartridge
164,166	Reserved
170,172	LP/LS/LV11 Line printer number 1
174,176	LP/LS/LV11 Line printer number 2
200,202	LP/LS/LV11 Line printer number 0 (includes LA180 parallel interface)
204,206	RH11,RH70: RS03/RS04 fixed-head disk RF11: fixed-head disk
210,212	RK611/RK711: RK06/RK07 disk cartridge
214,216	TC11: DECtape
220,222	RK11/RKV11: RK05 disk cartridge
224,226	RH11/RH70: TU16, TE16, TU45 magtape TM11: TU10/TE10 magtape TS03: magtape TS11: magtape first controller (others float) TS05/TSV05: magtape
230,232	CD11/CM11/CR11: card reader
234,236	UDC11: Digital control subsystem
240,242	PIRQ (programmed interrupt request)
244,246	FPP or FIS floating-point exception
250,252	KT11: memory management fault
254,256	RP11: RP02/03 disk RH11/RH70: RP04/05/06/RM02/03 Disk

Table 1–5 (Cont.): Interrupt Vectors

Location	Contents
260,262	TMSCP Unit 0 (bootable unit); TA11: cassette tape
264,266	RX11/RXV11/RX211/RX2V1: RX01, RX02 diskette
270,272	LP/LS/LV11 line printer number 3
274,276	LP/LS/LV11 line printer number 4
300,302	Start of the floating vector area
320,322	VT11/VS60 Graphics terminal (requires three vectors)
324,326	VT11/VS60
330,332	VT11/VS60

1.1.4 Background Job

The **background job** in the single-job and multi-job systems is essentially identical for the purpose of this discussion. Figure 1–4 shows the general structure of an unmapped background job, as well as its relative location in memory. Chapter 3 describes and illustrates background job loading in a mapped system.

There are five ways in which RT–11 can load a background job: RUN, R, VRUN, V, and .CHAIN. They are described in the following sections.

1.1.4.1 RUN Command

One way to load a job is to use the *RUN* command. The RUN command is the same as the GET and START commands combined. First, if the SAV file is not on the system device, RUN (or GET) loads the handler for the proper device. When this occurs, KMON and the USR, which normally occupy the space above the background job and below RMON, relocate themselves, if necessary. For more information on the USR and KMON, see later sections of this chapter. For information on virtual and completely virtual background job loading, see Chapter 3.

The space available for background job loading consists of the background job area, the space occupied by KMON, and (in unmapped systems) the space occupied by the USR (unless the USR is set to NOSWAP). In mapped systems, the USR is always resident. If the job needs more space, an error message prints and then control returns to KMON.

With mapped systems, you can often run a background job in the completely virtual environment when there is insufficient memory to run it otherwise. See Chapter 3 for information about the SET RUN VBGEXE command, and other information about the completely virtual environment.

Figure 1–3: Interrupt Vector Area

Once the job passes the size tests, RUN loads from the file those memory locations (0 through 476) that are not protected. KMON reads block 0 of the .SAV file into an internal USR buffer. It extracts information from locations 40–64 and 360–377 (the CCB bitmap, described further in this section). Using the protection bitmap (called $LOWMA), which resides in RMON, KMON checks each word in block 0 of the file. It does not load locations that are protected, such as location 54 and the device interrupt vectors. It loads unprotected locations into memory from the USR buffer. Next, KMON sets $USRTO (location 50) to the top of the user program.

By default, the RUN command causes RT–11 to load main memory locations 500 through 776 with a copy of the command string following RUN ddn:filname. This enables a program to make use of the command string that was used to invoke it.

If required, RT–11 can load locations 500 through 776 from the save image of the program rather than with a copy of the command string. To do this with the RUN

Figure 1–4: Background Job

command, you must set CHAIN$ (bit 8) in $JSW (offset 40) in block 0 of the save image. CHAIN$ must be set in the file, not in memory, after the program is loaded.

For example, the command:

```
RUN SY:FOO INPUT OUTPUT
```

returns the string INPUT OUTPUT in locations 500 through 776 if CHAIN$ in $JSW in block 0 of SY:FOO.SAV is clear. If CHAIN$ is set, RT–11 loads locations 500 through 776 from the save image.

To load locations 1000 and up, RUN examines the *core control block*, called the CCB, which starts at location 360 in the job file and goes through location 377. The CCB is restricted for use by the system. The Linker stores the program memory usage bits in these eight words, which are called a bitmap. Each bit represents one 256-word block of memory and is set if the program occupies any part of that block of memory.

Bit 7 of byte 360 corresponds to locations 0 through 777; bit 6 of byte 360 corresponds to locations 1000 through 1777, and so on. The monitor uses this information when it loads the program. If KMON is in memory space that the program needs to use, KMON puts the block of the .SAV file into a USR buffer and then moves it to the file SWAP.SYS.

Finally, when it is time to begin execution of the program, KMON transfers control to RMON. RMON reads the parts of the program, if any, that are stored in SWAP.SYS into memory, where they overlay KMON and possibly the USR. The monitor keeps track of the fact that KMON (and perhaps the USR) are swapped out, and execution of the program begins. Figure 1–5 summarizes how the RUN command loads a job image into memory.

When the program terminates, RMON reads KMON and the USR back into memory from the monitor .SYS file. The memory area up to the bottom of KMON contains the background job image. If the job overlaid KMON and SET EXIT SWAP is in effect, the remainder of the job image is first written out to SWAP.SYS. This procedure allows the EXAMINE and DEPOSIT commands to operate on the job image on disk, even though KMON has written over the job's locations in memory, and the RESTART command can restart the program. If SET EXIT NOSWAP is in effect, program termination is faster, but you cannot use the EXAMINE, DEPOSIT, or RESTART commands.

1.1.4.2 R Command

The **R command** is identical to the RUN command except the default location for the utility being run with the R command is SY.

1.1.4.3 VRUN Command

The **VRUN command** runs a job in the completely virtual environment under mapped monitors. The default location for the utility being run is DK. The completely virtual environment is described in Chapter 3.

1.1.4.4 V Command

The **V command** is identical to the VRUN command except the default location for the utility being run with the V command is SY.

1.1.4.5 .CHAIN Request

The fifth way to load a job is to chain to it from another job. The first job issues the *.CHAIN* programmed request to do this. The second job can use information in memory locations 500 through 776 that was placed there by the first job. Consequently, the only difference between loading a job with the RUN command and starting a job by chaining to it is that chaining does not load memory locations 500 through 776 from the second file unless you set the chain bit in the $JSW of the second file at assembly time.

Figure 1–5: RUN Command

Note that chaining to a FORTRAN job does not preserve channel information from the previous job. This is because FORTRAN itself closes the channels and discards the impure area.

1.1.5 Resident Monitor (RMON)

The **Resident Monitor** (RMON) is the RT–11 monitor component that is always resident in memory. When you bootstrap an RT–11 system, the bootstrap routine determines how much main memory is available. RMON loads at the highest possible low memory address, just below the system device handler. It does not move during system operation.

RMON contains routines to handle the programmed requests in RT–11. It also contains the background job's impure area, the error processor, timer routines, console terminal service routines, USR swap routines, and other monitor functions. Figure 1–6 shows a summary of the contents of RMON. See Chapter 2 for more information.

Link maps of the distributed RT–11 monitors are part of the distribution kit. They exist as files named RTxx.MAP, where *xx* is the monitor name, such as RTXM.MAP. Table 1–6 lists the p-sects that make up RMON and KMON.

Figure 1–6: Resident Monitor (RMON)

Table 1–6: Monitor P-sects

P-sect Name	Contents
RT11	KMON
RMNUSR	USR buffer and code
RTDATA	RMON fixed offsets and database
OWNER$	$OWNER table
UNAM1$	$UNAM1 table
UNAM2$	$UNAM2 table
PNAME$	$PNAME table
ENTRY$	$ENTRY table
STAT$	$STAT table
DVREC$	$DVREC table
HSIZE$	$HSIZE table
PNAM2	$PNAM2 table
DVSIZ$	$DVSIZ table
DVINT$	$DVINT table
MTTY$	Multi-terminal terminal control blocks
RMON	Resident Monitor
USRRMN	RMON code in the USR source
XMSUBS	Extended Memory routines
MTEMT$	Multi-terminal programmed requests
MTINT$	Multi-terminal interrupt service
STACK$	RMON stacks
PATCH$	Patch space
OVLYnn	KMON overlays containing command processors
$LAST$	Last monitor .PSECT

1.1.6 System Device Handler

The **system device handler** is the handler for the device from which the system was bootstrapped. *RT–11 Device Handlers Manual* describes the structure of a system device handler in detail.

At bootstrap time, the monitor loads any system support handlers and then binds together with the system device handler file found on the system volume. The system device handler is loaded into memory immediately below any system support handler or the I/O page. RMON is loaded below the system device handler. Once it is read

into memory, the system device handler remains resident and does not change its location. Figure 1–7 shows where the system device handler resides in memory.

Figure 1–7: System Device Handler

1.1.7 I/O Page

The highest 4K words[1] of addressing space in PDP–11 computers are reserved for device control, status, and data buffer registers. This area is called the **I/O page**. In addition to the device registers, it may also contain the Processor Status word and, for some processors, the system's general registers (R0 through R5), the stack pointer (R6), and the program counter (R7). Locations in the I/O page are directly addressable by application programs and system software, but since they are bus

[1] An LSI-11 with MSV-11DD and memory jumper and the SBC–11/21 have a 2K-word I/O page and 30K words of regular memory. Throughout this manual, however, a 4K-word I/O page is assumed.

addresses and not memory locations, they cannot be used to store code and data. Figure 1–8 shows where the I/O page is addressed in relation to the rest of the system components. You can find more information on the I/O page and the device registers for your own processor and peripherals in the *PDP–11 Processor Handbook*, the *PDP–11 Peripherals Handbook*, the *Microcomputer Processor Handbook*, the *Memories and Peripherals Handbook*, and in most hardware manuals.

Figure 1–8: I/O Page

1.2 Dynamic Components

Dynamic components do not always load into fixed places in memory. Once loaded, the USR and KMON can continue to shift location based on the state of the rest of the system. The dynamic components and areas are as follows:

- Device handlers (device drivers) and free space
- Foreground and system jobs
- User Service Routine (USR)
- Keyboard Monitor (KMON)

As you read about the rest of the dynamic components, you will also learn how the system manages free space in memory. You have already seen how the system device handler and RMON load at the highest possible addresses, and how the background job begins loading at location 1000 and up. The strategy behind the way the system manages free memory is that it attempts to make the most space available for foreground and background application jobs.

1.2.1 Device Handlers and Free Space

Device handlers (drivers) are routines that provide the interface to the computer's hardware devices. The handlers **drive**, or **service**, peripheral devices and take care of moving data between memory and devices. *RT–11 Device Handlers Manual* describes device handlers in greater detail.

RT–11 uses a dynamic scheme to provide memory space for loaded handlers, foreground jobs, system jobs, indirect file and command line expansion. Memory is allocated in the region above KMON/USR and below RMON. If there is not enough memory in this region (initially, after the system is bootstrapped, there is none), memory is taken from the background region by ''sliding down'' the KMON and USR the required number of words.

When memory allocated in this manner is released, the memory area is returned to a singly-linked free memory list, the head of which is located in RMON. Any contiguous blocks are concatenated into a single larger block. A block found to be contiguous with the KMON/USR is reclaimed by ''sliding up'' the KMON/USR, thus removing the block from the list.

Figure 1–9 shows an SB system with a small application job and two loaded device handlers. When you issue the LOAD monitor command, the handler loads into the memory area just above the USR and KMON. The USR and KMON slide down in memory to provide the handlers with enough space, leaving less space for the user program.

Once handlers are brought into memory, they do not move up or down, as the USR and KMON do. Figure 1–10 shows the system after the monitor UNLOAD command has removed one handler from memory. In the figure, the free space above handler #2 has not been reclaimed and is available for later use. A handler that is the same size as the empty space, or smaller, can be loaded there without causing any other components to move.

Figure 1–9: SB System with Two Loaded Handlers

Figure 1–11 shows the system after the second handler was unloaded. This time there is free space directly above the USR (the space formerly occupied by the two handlers), so the USR and KMON slide up into it, making more space available for the user program.

1.2.2 Foreground and System Jobs

Foreground and system jobs are described in detail in the the *Introduction to RT–11*. In a multi-job system, **foreground jobs** and **system jobs** are essentially identical. The RT–11 system jobs in the multi-job environment are the error logger (ERRLOG), the on-line index utility (INDEXX), the keypad editor (KEX), the file queuing program (QUEUE), the transparent spooler program (SPOOL), and the virtual terminal communication program (VTCOM). For more information on the foreground and system job loading environment, see Chapter 3.

Figure 1–10: SB System with One Handler Unloaded

Figure 1–12 shows the general structure of an unmapped foreground job, as well as its relative location in memory. Handlers loaded after the foreground job are placed below it in memory, and above the USR. (See Chapter 2.)

1.2.2.1 Differences Between Foreground and Background Jobs

There are some significant differences between foreground and background jobs.

1. The impure area (described in Chapter 2) for the foreground job is located immediately below the job area itself. For a background job, the impure area is always in RMON.
2. Another major difference is that a foreground job cannot dynamically change its memory allocation: the job is a fixed size. You can only change the low memory

Figure 1–11: SB System with Both Handlers Unloaded

allocation at FRUN time by using the /BUFFER:n option. See Chapter 3 for further information on memory allocation.

3. You must load all the handlers a foreground job needs before the job attempts to use them. A background job, on the other hand, can use the .FETCH programmed request to load a handler when it is needed.

4. For FB systems only, if the USR is swapped out and the foreground job needs it, the foreground job must allocate 2K words of program space for the USR to swap over. (See Section 1.2.3 for more information on the USR.)

1.2.2.2 FRUN Command

The **FRUN command** loads the foreground program into memory and starts execution. The **SRUN command**, which performs the same functions for system jobs, is essentially identical. You can also use FRUN or SRUN to start a virtual .SAV job, since these jobs do not require relocation. (See Chapter 3 for more information

Figure 1–12: Foreground Job

on virtual jobs.) Before you start a job with FRUN, you must load all the handlers the job requires. You can use the FRUN/PAUSE option, load the handlers and then resume the foreground job. In any case, the handlers need to be loaded only before the job actually uses them.

FRUN first opens the .REL file or virtual .SAV file, reads its first block (locations 0 through 776), and determines how much memory the job requires. The job's total memory requirement is equal to the sum of the program itself (as indicated by $USRTO (location 50) in block 0 of the file), the size of the impure area, the extra space allocated with the FRUN/BUFFER:n command, and the extra space (if any) allocated with the LINK/FOREGROUND:stacksize command. If you do not allocate extra stack space, the default stack size is used. If there is not enough memory available to run the job, an error message prints and the monitor dot prints on the terminal.

Once FRUN gets the memory space the job needs, it sets up the job's impure area. FRUN also sets up the job context on the foreground job's stack for FB systems, or in the job's impure area for mapped systems. So, when you first load a foreground job, it appears to be context-switched out. (See Chapter 2 for more information on context switching and other monitor functions.)

Next, FRUN loads the foreground main program into memory and relocates addresses in the root to reflect the current load address. Virtual .SAV files do not require relocation. If the job is overlaid, there is one more step before execution can begin. FRUN reads and relocates just the root of an overlaid program. Then it reads the overlay relocation information into a buffer. One by one, each overlay segment is then read into memory, relocated, and written back to disk. Finally, FRUN starts job execution. Therefore, you cannot run more than one copy of a disk-overlaid .REL system job from the same file. Figure 1–13 shows a summary of how the FRUN command loads a foreground job image into memory.

1.2.2.3 Starting Foreground and System Jobs

The *Introduction to RT–11* provides a lot of information on starting foreground and system jobs; you should read that part before using either. In brief, Figure 1–14 illustrates the procedure Digital recommends for starting up a system that has both system jobs and a foreground job. In the example in Figure 1–14, the two handlers that the QUEUE program needs are loaded first, since the error logger and the QUEUE program are both intended to run as long as the system runs. (The QUEUE program needs handlers for the device to which it will copy files, as well as handlers for the devices on which those files are currently stored. The error logger needs no specific handler; it logs errors from any handler that calls it.) The SRUN command is used next to start the more important of the two system jobs (the error logger). Then the second system job (QUEUE) is started, also with SRUN. This ordering of system jobs gives the error logger higher priority by default than the QUEUE program. (Note that if it is not convenient for you to load the higher priority system job first, you can assign priorities to the system jobs with the SRUN/LEVEL:n command.) Lastly, the foreground job, which requires no other handler, is started with the FRUN command. In Figure 1–14, the foreground job, which always has the highest priority, is loaded last, because it will only run for a short time before it is stopped, unloaded, and replaced by a different foreground job. After you stop a job by typing two CTRL/Cs or the ABORT command, you must use the monitor commands to unload it and replace it with another. RT–11 does not provide a way for one foreground job to automatically start another.

Multiple copies of a system job can be run at the same time by assigning, using the /NAME: option, a different logical name to each job. That is described for KEX in the *Introduction to RT–11*.

Figure 1–13: FRUN Command

1.2.2.4 Foreground Stack

The foreground job's stack is located immediately above the impure area. Its default size is 128_{10} bytes. You can change the size of the stack at link time by using the /FOREGROUND:stacksize option.

You can also change the location of the foreground stack. To do this, use the /STACK:n option at link time, and specify either an octal value for the stack pointer

Figure 1–14: FB System

or a global symbol name. If you change the stack location, you are responsible for allocating space for the stack in your program.

Be careful not to let the stack overflow during execution. Since RT–11 neither checks for this error nor makes any attempt to correct it, the most likely result is that your program or the impure area will be corrupted.

1.2.2.5 Foreground Impure Area

The memory locations just below the foreground job area contain job-dependent information. This area is called the **impure area**, and its contents are maintained by RMON. Chapter 2 lists the information contained in this area.

1.2.3 User Service Routine (USR)

The **User Service Routine** (USR) is the part of the RT–11 operating system that provides support for the RT–11 file structure. It contains instructions to:

- Fetch device handlers
- Get the status of device handlers
- Get and set information about files
- Open existing files
- Create new files
- Add queue elements
- Protect and unprotect files
- Delete and rename files
- Close files

In addition, the USR contains the Command String Interpreter (CSI) which interprets device, file, and option specifications. The default memory location for the USR is directly above the background area, or directly below the system jobs, foreground job, and loaded device handlers, if there are any. You can change this default location by setting an address in $UFLOA (offset 46) in low memory.

The USR is always resident in memory under all mapped monitors.

Under an unmapped monitor, the USR does not always have to be resident in memory. In fact, in an unmapped system, the USR is designed to be swappable in order to make as much space as possible available for user jobs. Generally, the USR is then only read into memory when file-oriented operations are required.

1.2.3.1 Structure

The USR consists of two basic parts: the buffer area and the permanent code area. The first section, which is two blocks long, contains code when the USR is brought into memory. This area also serves as the buffer in which the USR stores a device directory segment. The second section contains permanent code. Figure 1–15 shows an overview of the USR's structure and its memory location in a single-job system.

The first routine in the USR buffer section consists of initialization code to relocate pointers in the USR and KMON. This relocation code becomes active the first time the USR is entered after it is brought into memory. It relocates internal pointers in the USR that point to RMON and to other important locations within the USR. If the USR was called from KMON, it also relocates pointers to RMON within KMON.

In unmapped monitors, the next section of code handles the .QSET programmed request, followed by a small amount of scratch space, takes up the remainder of the two-block buffer area. In mapped monitors, the .QSET request is handled in the buffer area.

Figure 1–15: USR

Following the buffer area is the USR's permanent code which starts at offset 2000 from the beginning of the USR. The permanent code consists of routines that process the following programmed requests:

.CLOSE	.DELETE	.DSTATUS	.ENTER
.FETCH	.FPROT	.GFDAT	.GFINF
.GFSTAT	.LOOKUP	.RELEAS	.RENAME
.SFDAT	.SFINF	.SFSTAT	

The Command String Interpreter (CSI) occupies the end of the USR, where the .GTLIN, .CSIGEN and .CSISPC programmed requests are processed.

1.2.3.2 Execution

The general flow of execution in the USR is straightforward. When a fresh copy of the USR is brought into memory, its buffer area contains the code described in the previous section. When a program issues a USR programmed request, the first code to execute is the relocation code. This code then calls the routine to process the particular request that was issued. If the USR stays in memory, subsequent USR requests go directly to the routines that process them. The initialization code is not called again.

Usually, a USR request requires a device directory segment. If the correct segment is already in the USR buffer, the USR does not read in a fresh copy of that segment. If the correct segment is not in memory, or if the USR has no segment at all, the USR reads the directory segment into its buffer. When it does this, the USR stores two words of information in the RMON fixed offset area. $BLKEY, at offset 256, contains the number of the directory segment currently in the USR buffer. $CHKEY, at offset 260, contains the device's unit number in the high byte, and an index into the monitor device tables in the low byte.

It can be useful to you to know under what circumstances the USR reads in a new directory segment. The following conditions cause the USR to read in a new directory segment:

1. Anything that causes the USR to swap out. When a fresh copy of the USR is brought into memory, it will have no directory segment in its buffer and will be forced to read one from a device.
2. Executing code in the buffer area. Since the code to process some programmed requests is located in the USR buffer area, attempting to process one of those requests always causes a fresh copy of the USR to be brought into memory. The following requests cause this to happen:
   - .QSET (unmapped monitors only)
   - .EXIT (if your program was loaded over any part of KMON)
3. Issuing an .ENTER programmed request. This always causes the USR to read a fresh directory segment.
4. Issuing a .LOOKUP programmed request with a different device or file specification from the previous .LOOKUP. Note that doing a .LOOKUP with the same device specification as the previous .LOOKUP does not necessarily cause the USR to read in a fresh copy of the same directory segment. This is why

you cannot remove a volume from a given device unit, replace it with another volume, and expect the USR to have the new volume's directory segment in memory. However, in this situation, you can force the USR to read a directory segment from the new volume by locking the USR to gain exclusive use of it, storing a value of 0 in $BLKEY (RMON fixed offset 256), and then issuing a .LOOKUP programmed request with the same arguments as the previous .LOOKUP. Clearing $BLKEY causes the USR to ''forget'' the current directory segment and read a fresh one from the new volume.

1.2.3.3 USR Swapping with Unmapped Systems

The USR can be swapped out of memory with only unmapped (SB or FB) systems. Therefore, this entire section (up to Section 1.2.4) applies only to the SB and FB systems. A copy of the USR is always resident in memory in mapped systems.

Because the USR does not always have to be resident in memory for unmapped systems, you have a variety of options to consider when you design an application program. You can keep the USR in memory at all times (the simplest case), or you can arrange to have the USR swap into memory only when your program needs it. The latter procedure permits your program to use an extra 2K words of memory when the USR is swapped out. The guidelines that follow can help you design programs that handle the USR efficiently.

NOTE

In general, the burden of USR swapping should be undertaken by the program, not by the operator who runs it. SET USR NOSWAP is useful to override the default action of programs outside an operator's control (such as FORTRAN), but its use requires operators to understand internal programming details—a requirement that should be avoided if at all possible.

Keeping the USR Resident in an SB System

In an SB system, the normal location for the USR is just below RMON and loaded device handlers (see Figure 1–15). If your program does not need the space the USR occupies, you can force the USR to remain resident while your program is executing by issuing the monitor SET USR NOSWAP command before you run the program. In any case, if the space is not needed, the USR does not swap. Note that the USR can still slide up or down in memory, as Section 1.2.1 describes.

Keeping the USR resident means that 2K words less memory is available to your program. However, the directory operations involved in file opening and closing and in program loading will be faster because this arrangement eliminates swapping and disk I/O. In addition, the program will have a much simpler design. To keep the USR resident, a MACRO program should avoid issuing a .SETTOP request for memory above the base of the USR.

Remember that even though the USR is set to NOSWAP, there are some programmed requests that can cause a fresh copy of the USR to be brought into memory. For an SB system, these requests are .EXIT, and .QSET. If the USR is swappable and if

the background program issues a .SETTOP request for memory above the base of the USR, the USR loads into the area specified by the contents of $UFLOA (location 46) in low memory. If $UFLOA contains 0, as it should when you intend to keep the USR resident, the USR loads in its usual place, below RMON. However, if for any reason you move a different value to location 46 and then execute one of the requests that loads a fresh copy of the USR, the USR will then load into the area you specified. If you execute a program that keeps the USR resident, the monitor ignores the contents of $UFLOA.

Allowing the USR to Swap with an SB MACRO Program

The only reason to allow the USR to swap in an SB system is to gain access to the extra 2K words of memory that swapping makes available. To enable USR swapping, make sure that the SET USR SWAP command is in effect. (This is the default condition.)

A MACRO program gains access to the 2K words of memory because its high limit requires it, or because it does a .SETTOP to an address within the USR area. (Refer to Figure 1–5 for a summary of how the RUN and R commands load programs that overlay the USR area.) When the program issues a programmed request that requires the USR, the part of the program that occupies the USR area is written out to SWAP.SYS, and a fresh copy of the USR is brought into memory from the monitor file on the system volume. $UFLOA (location 46) should contain a value of 0 if you want the USR to swap into memory at its default location. If you want it elsewhere, put the starting address into $UFLOA during your program's initialization routine. When the programmed request completes, the part of the program in SWAP.SYS is copied back into memory, overlaying the USR. This sequence of events occurs for each programmed request that requires the USR, even if your program issues two or more requests in a row.

To make more efficient use of the USR, your program can issue the .LOCK programmed request before any other USR requests. This swaps part of your program out, reads the USR in, and returns to your program. After this, the USR remains in memory at the location you specified in $UFLOA (if any). You can now issue a number of USR programmed requests and avoid the overhead of USR swapping. When your program next needs the 2K words of space, use an .UNLOCK request to release the USR.

When the USR is swappable, it is important that you put it in a safe place in your program. This means that the area the USR will swap over must not contain code or data that will be needed at the same time the USR is in memory. The following is a list of code and data that must not be overlaid by the USR:

- Device block and/or CSI or .GTLIN file description string for the current request
- Active device handlers
- Active completion routines
- Active interrupt service routines
- Active I/O buffers

- Queue elements from .QSET
- I/O channels from .CDFN
- Program stack
- Trap service routines from .SPFA and .TRPSET
- Code executed between the .LOCK and .UNLOCK requests

You can control USR swapping by careful use of the .SETTOP request. A typical practice that many system utility programs use is to issue a .SETTOP request to obtain space up to the base of the USR. The programs then perform all their USR operations. Finally, the programs issue an additional .SETTOP request to obtain as much memory as possible, if necessary.

Another situation to be aware of occurs when a program issues a .SETTOP request for more memory than is available. In this case, the program is given only the amount of memory that is available. After issuing a .SETTOP request, a program must always use the value returned in R0 (or $USRTO (location 50) in low memory) as the true high limit of the program. For example, a program can issue a .SETTOP request for memory above the base of the USR when the USR is set to NOSWAP. However, the value returned to the program as its true high limit is just below the base of the USR.

Keeping the USR Resident in an FB System

As with an SB system, the easier way to deal with the USR in an FB system is to keep it resident. Use the SET USR NOSWAP command. This arrangement is suitable if the background, foreground, and system jobs have enough memory. The USR is brought into memory at its usual place, just below any loaded handlers and below the foreground job and it remains in memory during program execution. Neither job has to allocate program space for the USR, and programs execute faster without the overhead of USR swapping and disk I/O.

The important issue in an FB system with the USR resident is determining which job should have control of the USR. Because only one job can use the USR at a time, both jobs must be aware of sharing this resource. Since a program in an SB system can lock the USR in order to process a number of USR programmed requests, in an FB system, either the background job or the foreground job can lock the USR to gain exclusive use of it.

The .LOCK request gives ownership of the USR to one job. The .UNLOCK request releases the USR, making it available for the other job. The request .TLOCK can determine whether or not the other job has exclusive ownership of the USR. It permits a program to try for a .LOCK, but to continue with execution if the attempt fails.

The LOCK/UNLOCK system permits one job to lock out another for a considerable length of time. During a lockout, interrupt service and completion routines can run, but not mainline code. This could cause serious difficulties in a real-time foreground program. There are some ways to minimize or eliminate this lockout problem:

1. Be sure to separate USR operations from real-time operations.

2. Avoid using devices with slow directory operations, such as magtape.
3. Organize your real-time foreground program so that real-time operations are in interrupt service routines and completion routines and will not be affected if the mainline code is locked out with a pending USR request.

Typically, a real-time foreground job can be organized in three parts: an initialization phase, which opens all required channels and begins real-time operations; a real-time phase, which does interrupt service and I/O operations; and a completion phase, which stops real-time activity and closes the channels. With this arrangement, the background program can perform USR operations during the real-time phase without locking out the foreground. The foreground program can use .LOCK and .UNLOCK to prevent interference from the background job during initialization and completion phases.

Swapping Considerations for Background Jobs

When either the background job or the foreground job needs the extra 2K words of memory that swapping the USR provides, both jobs must be concerned with USR swapping. The general concerns for background jobs are those listed in the previous sections.

The easiest approach for the background job is to swap the USR into its default location, the highest 2K words of program space. If this is not convenient for any reason, the background job can select any other contiguous 2K words of program space. In this case, it must also put the starting address of the USR swap area into $UFLOA (location 46) in the system communication area. This location is context-switched in the FB system, so it always contains the correct value for the job that is currently executing.

The background job must not place any USR-sensitive code or data in the area where the USR will swap. In addition to the list in Section 1.2.3.3, the following items must not be in the USR swap area:

- Memory list from the .CNTXSW request
- Active message buffers
- Code containing the .LOCK or .TLOCK requests

You must also be careful that the background job does not lock the USR for an unreasonable length of time so it can block the foreground job from running. If you lock the USR in a background job, remember to unlock it as well.

Swapping Considerations for Foreground Jobs

If the background job issues a .SETTOP that causes the USR to swap, or if the background job is large enough to force the USR to swap, the foreground job must be concerned with USR swapping. However, while the background job can simply allow the USR to swap into its default position (the highest 2K words of the background job area), the foreground job has no default location for the USR.

It must allocate 2K words within its program bounds in which to swap the USR - space that must not contain any USR-sensitive code or data. The foreground job must also place the starting address of that space in $UFLOA (location 46) in the system communication area. This location is context-switched during normal foreground/background execution, so it always contains the correct swapping address for whichever program is currently executing.

The foreground program could also be concerned with sharing the USR with the background job. The .LOCK/.UNLOCK requests can give the foreground job exclusive ownership of the USR to prevent interference by the background job. The foreground job should avoid keeping the USR permanently locked, which sometimes happens strictly because of a programmer's oversight.

1.2.4 Keyboard Monitor (KMON)

The **Keyboard Monitor** (KMON) is the part of the RT–11 system that provides the communication link between you at the console terminal and the rest of the RT–11 system. KMON commands permit you to assign logical names to devices, load device handlers, run programs, control foreground/background operations, control system jobs, invoke indirect command files, and examine or modify memory locations. KMON is brought into memory when the background job completes. When KMON is in memory, the USR is also present directly above it.

KMON consists of a root segment and a number of overlays that contain the command processors. KMON runs as an ordinary background job, in user mode. The root segment is contained in the p-sect RT11. See Table 1–6 for a summary of all monitor p-sects.

The various command line interpreters (DCL, CCL, UCL, UCF) can be enabled or disabled by using the SET CLI command, as described in the *RT–11 Commands Manual*.

1.2.4.1 DCL Command Processing

When KMON interprets a command that you type at the terminal, it expands the command text into an internal indirect file. For example, the command COPY MYFILE DL:MYFILE expands internally into:

```
R PIP RET
DL:MYFILE=DK:MYFILE RET
^C
```

KMON stores this internal indirect file in the *command expansion buffer area*. KMON creates space in memory for this buffer area immediately above the USR. When KMON and the USR slide up or down in memory, the command buffer spaces move with them. Figure 1–16 shows KMON in memory.

The *Introduction to RT–11* gives an overview of KMON command processing. The *Introduction to RT–11* and *RT–11 System Generation Guide* describe how to remove individual commands or groups of commands from a system you create through the

Figure 1–16: Keyboard Monitor (KMON)

system generation process. If you are interested in modifying KMON itself to change the monitor command set, see the *Introduction to RT–11* for information.

1.2.4.2 CCL Command Processing

If KMON does not recognize the first word of an input line as a valid DCL command, it tries to treat the input line as a CCL command by searching for a program of that name on SY and running the program. If a program is found, KMON passes the remainder of the command line to the program in the CSI input buffer as a CSI command string, followed by a ^C. The general format of a CCL command is:

command <sp> field1<sp>field2

or

command <sp> csistring

If the first form is used, KMON converts it to the second form by reversing the fields and inserting an equal sign:

command <sp> field2=field1

For example, you might type:

```
PIP A=B
```

or

```
PIP B A
```

Both forms are equivalent to typing:

```
.R SY:PIP
*A=B
*^C
```

If the first word on the line is more than six characters, characters after the first six are ignored. *field1* and *field2* can contain multiple file names, separated by commas. If you have an application program on SY named EVALUA.SAV to evaluate certain collected data and print a report, you could type:

```
EVALUATE DU3:DATA16.DAT,DU1:DATA03.DAT LP:
```

This is equivalent to:

```
R SY:EVALUA
LP:=DU3:DATA16.DAT,DU1:DATA03.DAT
^C
```

1.2.4.3 Adding New Commands Through UCL

Using UCL is described in *Introduction to RT–11*; you should look there for information on defining UCL commands.

In parsing a command, KMON first checks to see if the first word of the line is a valid DCL command. If not, KMON tries, using the CCL conventions outlined above, to find and run a program of that name. If that also fails, KMON looks for the program UCL.SAV on SY and runs it if present. KMON passes to UCL the entire command line (including the first word) as ASCII text in the chain area starting at location 512. Location 510 contains the number of bytes in the command line. Locations 500 through 507 of the chain area are not used by UCL.

UCL interprets and expands the command line and performs any operations required by the command. UCL can reformat and pass the command to another program by doing a .CHAIN, or UCL can create a new command line and pass the new command to KMON by doing a normal or special chain exit. For example, you could type:

```
BUILD MYPROG
```

UCL.SAV might expand the command into the following series of commands:

```
R MACRO
MYPROG,LP:/C=MYPROG
^C
R LINK
MYPROG=MYPROG
^C
RUN MYPROG
```

These commands could then be passed back to KMON by doing a normal chain exit or a special chain exit. Refer to the following section and sections 1.1.2.2 and 1.1.4.5 for information about normal and special chain exits.

1.2.4.4 User Commands First (UCF) Feature

Using UCF is described in *Introduction to RT–11*; you should look there for information on defining UCF commands.

KMON supports the User Commands First (UCF) feature. UCF allows you to create a KMON preprocessor that will intercept all command line input after KMON has tried indirect file (@) syntax and before DCL parsing. This preprocessor can be used to determine the way KMON processes valid DCL commands and to process commands that KMON would otherwise refuse.

The distributed file UCL.SAV (the User Command Linkage utility) can be copied and renamed to create a functioning UCF utility as explained in the *Introduction to RT–11*.

Once you have created the UCF utility, you make it available to KMON by issuing a SET command (SET CLI *condition*) as described in the *RT–11 Commands Manual* or by performing the following customization. The SET command affects only the memory image monitor; the customization patch alters the file copy. Digital recommends you include the SET CLI *condition* command in a start-up command file to 'automatically' set the command line interpreter as you want, rather than perform this patch.

Besides making UCF available to KMON, the customization patch can also selectively inhibit command processors (DCL, CCL, and UCL) from processing KMON commands. Great care must be taken when inhibiting command line processors; if all are inhibited, only the R and RUN commands are recognized. Think again about using the SET CLI *condition* command.

In the customization, *monitr*.SYS is the name of the monitor file that you want to modify. The value of the lower four bits of the value represented in *xxxxxx* specifies the command line interpreters: 1 for UCF, 2 for DCL, 4 for CCL, and 10 for UCL. The distributed monitors support DCL, CCL, and UCL command interpreters; the value represented by *xxxxxx* is 177416.

To enable the UCF command line interpreter, set bit 0 (value 001) by specifying 177417 for *nnnnnn*. To selectively inhibit distributed command line interpreters DCL, CCL, or UCL, clear ,respectively, bits 1, 2, or 3 in *xxxxxx* by entering the corrected values for *nnnnnn*.

```
.R SIPP  RET
*   monitr.SYS  RET
Base?        0   ret
Offset? CLIFLG  RET

Base        Offset          Old        New
 000000      CLIFLG       xxxxxx        nnnnnn RET
 000000      CLIFLG+2     xxxxxx         CTRL/Y  RET
*
.
```

KMON/UCF Communication

KMON and UCF communicate by means of a 1-word interface in RMON, defined in .CLDDF and .CLIDF in SYSTEM.MLB. The low byte in this word—CLI.FL—

indicates which command processors can be run. The high byte—CLI.TY—indicates which command processor is running. This interface is described in Section 2.6.1.13.

KMON collects a command from the command line and checks for indirect file command syntax. If indirect file command syntax is not found, KMON checks UCF.KM (bit 7) in CLI.FL.

If UCF.KM is set, KMON clears the bit and goes through the normal parsing sequence (DCL, CCL, UCL). If UCF.KM is clear, KMON checks UCF.ON (bit 0).

If UCF.ON is set, KMON places the command line in the chain area (locations 510 through 776) and runs UCF.SAV.

If UCF processes a command line and returns it to KMON, UCF sets UCF.KM in CLI.FL and performs a special chain exit. If UCF executes the command line, it does not set UCF.KM and does not perform a special chain on exit.

1.2.4.5 Passing Commands Through the Chain Area

The RUN command and CCL commands cause KMON to load the chain area with a copy of all parameters that appear in the command line (that is, all data following the file name and separator). KMON stores these parameters exactly as they appear in the command line. This feature lets KMON use the chain area to pass unaltered parameters to the program that is being invoked. This is useful for programs that require command input that does not contain a standard RT–11 file specification, such as SETUP commands.

All constructions are valid. KMON stores the unchanged parameters in the chain area beginning at location 512.

A byte count of the contents in the chain area is stored at location 510. The byte count includes all null bytes. A value of 1 at location 510 informs a program that there is no data following the file name.

For example, the following CCL command includes a 2-word string as a parameter.

```
.ECHO HELLOOO THERE
```

KMON stores the string in the chain area, beginning at location 512, exactly as it appears in the command line.

To obtain unaltered parameters from the chain area, a program uses the GTLIN SYSLIB routine as described in the *RT–11 System Subroutine Library Manual*.

The .GTLIN programmed request cannot be used to obtain unaltered parameters from the chain area.

1.2.4.6 Character Case Handling

The case of characters stored in the KMON input buffer depends on the way TTLC$ (bit 14) in the $JSW is set at the time the characters are typed. KMON does not automatically convert lowercase characters to uppercase from terminal input or command files. If you wish all characters to be upper case, use the SHIFT/LOCK keys or the CAPS/LOCK key.

A program using .GTLIN with the lowercase bit set receives the commands in lowercase if the commands were entered in lowercase.

Since the case of a character depends on the contents of the job status word at the time the character is entered, the case of characters that are entered by means of the type-ahead feature are unpredictable.

For compatibility with earlier versions of RT–11, the USR puts all characters indicating options in uppercase. Even so, Digital recommends that all programs test and convert option characters before processing them. The recommended method for testing and converting lowercase characters to uppercase is:

```
        CMPB    @SP,#'A+40
        BLT     X
        CMPB    @SP,#'Z+40
        BGT     X
        BICB    #40,@SP
X:
```

1.3 Determining Components Size

Whether you are using a distributed or customized monitor, if you are using a distributed system and you need to know the sizes of the components, you should follow the guidelines in the next few sections.

1.3.1 Size of the USR

For unmapped systems, the size of the USR is always 2K words. For mapped systems, the USR, which is always resident, is somewhat larger. Your running program can determine the exact size of the USR by examining RMON fixed offset 374, $USRAR, which contains the size of the USR in bytes. You can also determine the size of the USR by issuing the monitor commands SET USR NOSWAP and SHOW MEMORY.

1.3.2 Size of KMON

The size of KMON is the same as the size of the p-sect RT11. Examine the link map that resulted from the system generation for your system to obtain this value.

1.3.3 Size of RMON

To determine the size of RMON, issue the SHOW MEMORY monitor command. This command prints the base address of RMON and its size in decimal words.

1.3.4 Size of Device Handlers

The size of each device handler, in bytes, is contained in H.SIZ (location 52) of the handler's .SYS file. You can also obtain this value by issuing a .DSTATUS programmed request on the device from a running program or by issuing the SHOW MEMORY monitor command, which reports the sizes of all loaded device handlers.

Chapter 2
Resident Monitor (RMON)

The main purpose of the *Resident Monitor* (RMON) is to provide services to running programs and to the Keyboard Monitor. The services include fielding traps and interrupts, providing the programmed requests, and acting as the central manager of the device-independent I/O system. In a multi-job system, the monitor also arbitrates the demands of up to eight jobs for processor time.

This chapter describes the functions of RMON that are generally common to all RT–11 systems. It provides information on the monitor's terminal service for a single console terminal. (See Chapter 4 for information on multiterminal systems.) It also describes how clock interrupts are handled and explains how timer support is implemented. The queued I/O system is discussed, scheduling for multi-job systems is described, and the system job feature is introduced. Lastly, information on RMON's data structures is provided.

2.1 Terminal Service

RT–11 provides terminal service through RMON. Terminal service is always resident, and it is part of RMON itself. Because of the way RT–11 implements terminal service, no handler is involved in the interaction between you at the terminal and the running system. It is designed to be a good interface between a person and the system, rather than an interface between a peripheral device and the system.

As part of the resident terminal service, RMON provides special programmed requests for terminal I/O. Because it uses ring buffers to implement the terminal service, RMON provides support for line-by-line editing. The terminal input interrupts are always enabled, which means that you can get the system's attention at any time by pressing CTRL/C, CTRL/B, CTRL/F, and so on. You can also type ahead to the system without losing characters.

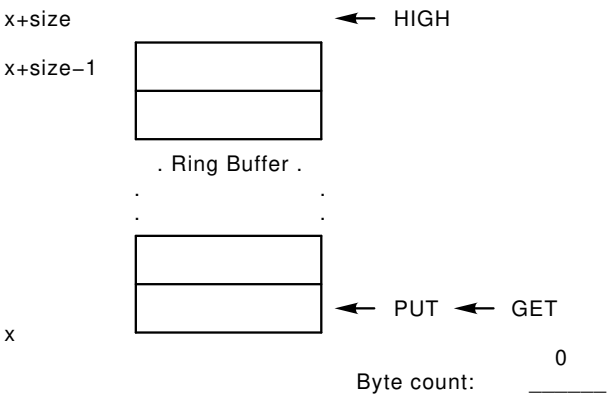
The ring buffers are the heart of the terminal service implementation. In single-job systems, one input ring buffer and one output ring buffer are located in the impure area of the single background job. In multi-job systems, each job has its own set of ring buffers located in its impure area. The ring buffers store text in a buffer zone between you at the terminal and a running program in memory. The default size of the input ring buffer is 134_{10} bytes; the default size of the output ring buffer is 40_{10} bytes.

2.1.1 Output Ring Buffer

An output ring buffer consists of the buffer area, three pointers, and a byte count. The buffer, or ring, itself is a block of bytes reserved for storing characters. Two of the three pointers store and retrieve characters. The PUT pointer marks the location where the next character will be stored and is used by the programmed requests that

fill the buffer, such as .TTYOUT, .TTOUTR, and .PRINT. The GET pointer marks the next character to be retrieved and is used by the output interrupt service routine that sends characters to the terminal. The third pointer, HIGH, points to the first memory location past the buffer. Lastly, the monitor maintains a byte count for the number of characters currently in the buffer. Figure 2–1 shows an output ring buffer in memory just after the system was bootstrapped.

Figure 2–1: Output Ring Buffer



2.1.1.1 Storing a Character in the Output Ring Buffer

The output ring buffer is filled by characters that are passed by .TTYOUT, .TTOUTR, and .PRINT. Characters that echo what you type on the terminal are also stored here, including sets of backslashes to enclose text you rub out with the DELETE key on a hard copy terminal. To store a character in the output ring buffer, the monitor first compares the buffer size to the byte count to check for room. If there is no room, the character cannot be stored. This condition is sufficient to block a job if the job is doing output. (If the output is the result of echoing, it is simply discarded.) If there is enough room, the monitor checks to see if the PUT pointer is equal to the HIGH pointer. This check ensures that the PUT pointer is pointing to a location that is within the buffer. If the PUT and HIGH pointers are the same, the monitor subtracts the size of the buffer from the current PUT pointer to obtain the new PUT pointer. By doing this, the monitor ‘‘wraps’’ around the ring to move from the highest address in the buffer to the lowest one.

Next, the monitor moves a byte into the buffer and it increments both the PUT pointer and the byte count. Figure 2–2 shows how characters are stored in the output ring buffer.

Figure 2–2: Storing Characters in the Output Ring Buffer

2.1.1.2 Removing a Character from the Output Ring Buffer

The terminal output interrupt service routine removes characters from the output ring buffer. If the character count is 0, the routine terminates. The routine checks to see if the GET pointer is equal to the HIGH pointer. If it is, this means it is time to ''wrap'' around the ring to move from the highest address in the buffer to the lowest one. The wrap routine subtracts the size of the buffer from the current GET pointer to obtain the new value of the GET pointer. This check ensures that the GET pointer is pointing to a location that is within the buffer.

Next, the output interrupt service routine removes one character through the GET pointer and prepares to send it to the terminal. It increments the GET pointer and decrements the byte count.

2.1.2 Input Ring Buffer

The input ring buffer is similar to the output ring buffer except that in addition to the GET, PUT, and HIGH pointers, it has a LOW pointer that points to the first byte of the buffer. This pointer is useful when the pointers are moving backward through the buffer as a result of CTRL/U or DELETE. It indicates when to ''wrap'' the buffer in the reverse direction, from the lowest address to the highest.

The monitor also keeps a count of the number of lines that are stored in the input ring buffer. A *line* is any sequence of characters terminated by line feed, CTRL/Z, or CTRL/C. (Each time you issue a carriage return at the terminal, RT–11 stores two

characters in the input ring buffer: a carriage return and a line feed.) In normal mode, the monitor does not pass input characters to a program until an entire line is present. This is why you can use DELETE to rub out a character and CTRL/U to remove an entire line when you are typing at the terminal. Since the monitor provides for line-by-line editing, application programs need not have this overhead themselves.

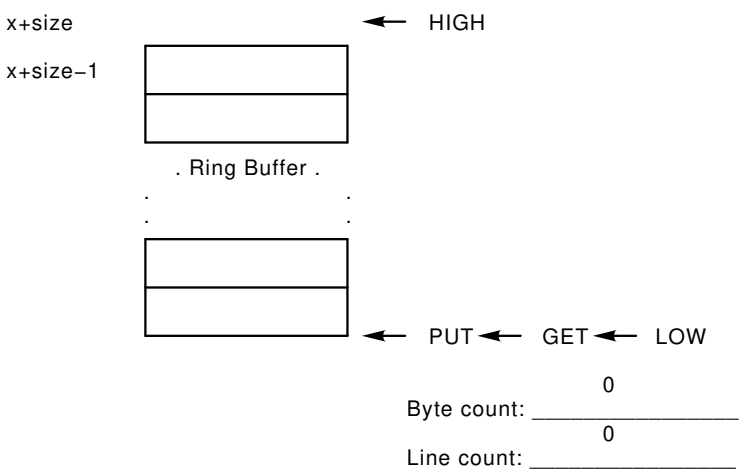
In *special mode*, however, the monitor passes bytes to a program exactly as they are typed on the terminal. In the latter case, the program itself must be able to interpret editing characters, such as DELETE and CTRL/U.

NOTE

Special mode does not provide the complete transparency required to handle devices other than terminals—such as communication lines – through RMON's terminal service. You can achieve transparency through the multiterminal feature of RT–11 by using the ''read pass-all'' and ''write pass-all'' modes. These are described in Chapter 4.

Figure 2–3 shows the input ring buffer just after the system was bootstrapped.

Figure 2–3: Input Ring Buffer



2.1.2.1 Storing a Character in the Input Ring Buffer

When you type characters at the terminal, the keyboard interrupt service routine stores them in the input ring buffer. First, the routine checks to see if there is room in the buffer. If there is no room, it rings the terminal bell (by putting a bell character in the output ring buffer). If there is room, the routine increments the byte count, increments the PUT pointer, wrapping it if necessary, and stores the

byte in the ring buffer. It also increments the line counter if the character typed is a valid line terminator. Figure 2–4 shows how characters are stored in the input ring buffer.

Figure 2–4: Storing Characters in the Input Ring Buffer

2.1.2.2 Removing a Character from the Input Ring Buffer

The monitor removes characters from the input ring buffer when it processes the .TTYIN, .TTINR, .GTLIN, .CSIGEN, and .CSISPC programmed requests. First it increments the GET pointer, wrapping around the ring if necessary. Then it gets a byte from the buffer and decrements the byte count. It decreases the line count as well if the character is a valid line terminator.

2.1.3 High-Speed Ring Buffer

RT–11 provides an optional, additional high-speed ring buffer that you can specify during system generation. This adds an extra input ring buffer to RMON; it adds an extra output ring buffer only if your system has multiple DL interfaces.

When the high-speed ring buffer is present, all character processing and interpretation is performed at fork level. The high-speed buffer is used to pass characters from interrupt level to fork level. The advantage of having the high-speed

buffer is that it allows the monitor to handle short bursts of characters coming in at a very high rate. This is useful for systems with intelligent terminals that report their status by sending a burst of information to the host computer. It is also useful for connecting one computer to another over a serial line.

The disadvantage to using the high-speed ring buffer is that a .FORK call is required for each burst of characters, and, thus, overall terminal service may be slower.

2.1.4 Terminal I/O Limitations

Terminal input and output limitations are completely separate; you use different methods to change each of them.

RT–11 accepts terminal input in either of two forms: a line at a time, or a character at a time. In line mode, characters you type at the terminal are stored in the input ring buffer until you type a valid line terminator such as carriage return, line feed, CTRL/Z, or CTRL/C. Only then does a running program receive the line of data. The factor limiting the length of the input line is the size of the input ring buffer. (The setting of the terminal right margin bears no relation to the length of the input line.) The default length is 134_{10} bytes, but you can change this through the system generation process. Any attempt to insert characters beyond this limit causes the terminal bell to ring, and the extra characters are lost. The Command String Interpreter can accept only 80 characters per line. Most utility programs (PIP for example) use the CSI to obtain lines of data from the terminal, limiting the functional input line length to 80 characters.

In character mode, a running program receives each character immediately after you type it at the terminal. In this mode, you can enter any number of characters without using a line terminator.

The length of terminal output lines is not related to the size of the output ring buffer; instead, it is related to the setting of the terminal right margin. Use the SET TT: WIDTH=*n* command to adjust the right margin. (See the *RT–11 Commands Manual* for details on SET TT: WIDTH and SET TT: CRLF.)

2.1.5 Control Functions

A special aspect of RT–11's terminal service is its response to control characters that you type at the terminal. The monitor handles each character differently, depending on the special function of each one. The following sections describe the different processes involved for the various control characters.

2.1.5.1 CTRL/C

When you type one CTRL/C at the terminal, the terminal interrupt service routine puts it into the input ring buffer, just as it would any other character. The monitor treats it as a line terminator and passes it to the running program.

However, if you type two CTRL/Cs in a row, the monitor processes them entirely differently. Instead of passing them directly to the program, the monitor aborts the running job. A program can use the .SCCA programmed request to intercept CTRL/C and prevent the abort (see the *RT–11 System Macro Library Manual* for a description of .SCCA).

2.1.5.2 CTRL/O

When the terminal interrupt service routine detects a CTRL/O, it never places the character in the input ring buffer, even if it is in special mode. The monitor simply toggles a flag in the impure area. (This flag is the sign bit of the output ring buffer byte count.)

The first time you type CTRL/O, the monitor echoes it, then clears the output ring buffer byte count. It empties the ring by setting the GET and PUT pointers equal to each other, and output from a running program is thrown away. This can unblock a job waiting for room in the output buffer. The next time you type CTRL/O or your job issues the .RCTRLO programmed request, normal output resumes.

2.1.5.3 CTRL/S and CTRL/Q

RT–11 implements terminal synchronization through the characters CTRL/S and CTRL/Q. CTRL/S, or XOFF, is a signal that stops a host computer from transmitting data to a terminal. The CTRL/Q, or XON, signal causes the computer to resume the transmission. Although XOFF has many uses, RT–11 supports only the two most common.

In a typical situation, you may be doing program development using a video terminal. When you use the TYPE monitor command to review a file, the text scrolls past faster than you can read. You can press the Hold-Screen key or type CTRL/S to stop the display so that you can read it, and then again press Hold-Screen or type CTRL/Q to resume the scrolling. You initiate the XOFF yourself, in this case.

In another situation, the computer may send characters to a terminal faster than the terminal can display them. So, the terminal itself sends the XOFF signal to the computer, empties its internal silo, and sends XON when it is ready to accept more data. This procedure is transparent to you.

A flag in RMON, called XEDOFF, indicates the XOFF/XON status. Typing CTRL/S sets the flag; typing CTRL/Q clears it. When XEDOFF is set, the monitor disables terminal output interrupts and stops emptying the output ring buffer. See the *RT–11 Commands Manual* for a description of the SET TT: NOPAGE command, which disables CTRL/S and CTRL/Q processing.

2.1.5.4 CTRL/B, CTRL/F, and CTRL/X

These control characters have meaning in only multi-job systems. In multi-job systems, CTRL/B and CTRL/F direct terminal I/O to the correct job. CTRL/X summons the system job prompt; you supply the system job name in response to that prompt. (See *Introduction to RT–11* for more information on communicating with system jobs.) The CTRL/B, CTRL/F, and CTRL/X characters are not put into the input ring buffer. Instead, they are recognized by the input interrupt service routine (unless SET TT: NOFB is in effect, in which case the characters have no special meaning) and the monitor switches the set of ring buffers it is using.

The interrupt service routine uses two control words, TTOUSR and TTIUSR, to point to the impure area of the correct job. The job's identification is stored in a special buffer in the impure area. The foreground job ID is the job name followed by a right bracket (>); the background job ID is B>; the ID for a system job is its job name.

When terminal I/O is directed to a different job, the new job's identification prints on the terminal.

2.1.6 SET Options Status Word, .TCFDF

The word $TTCNF in RMON is a status word that indicates which terminal SET options are in effect. For multiterminal systems, each terminal control block has a status word similar to $TTCNF. $TTCNF reflects the status of the SCOPE, PAGE, FB, FORM, CRLF, and TAB options. Table 2–1 shows the meanings of the bits. Unused bits are reserved for future use by Digital.

Table 2–1: SET Options Status Word ($TTCNF)

Bit	Symbol	Meaning When Set
0	HWTAB$	SET TT: TAB option is in effect.
1	CRLF$	SET TT: CRLF option is in effect.
2	FORM$	SET TT: FORM option is in effect.
3	FBTTY$	SET TT: FB option is in effect.
4-6		Reserved
7	PAGE$	SET TT: PAGE option is in effect.
8-14		Reserved.
15	BKSP$	SET TT: SCOPE option is in effect.

To get the status word and current width of the terminal (in systems without the multiterminal special feature), use the following lines of code:

```
        .LIBRARY "SRC:SYSTEM"
        .MCALL  .FIXDF  .TTCDF  .SYCDF
        .FIXDF                          ;RMON fixed area layout
        .SYCDF                          ;SYSCOM area
        .TTCDF                          ;Terminal Config area

        MOV     @#$SYPTR,R5             ;Get address of fixed area
        MOV     $TCFIG(R5),R5           ;Get address Term Config area
        MOV     $TTCNF(R5),STATUS       ;get current term config bits
        MOVB    $TTWID(R5),WIDTH        ;and current width setting
        .
        .
        .
STATUS: .BLKW   1
WIDTH:  .BLKB   1
        .END
```

Use the following additional line to obtain the value of the current carriage or cursor position (a value of 0 means the cursor or carriage is at the left margin):

```
        MOVB    -1(Rn),POSIT
```

2.2 Clock Support and Timer Service

You do not need a system clock in order to run RT–11 on a PDP–11 computer. However, if your computer does have a clock, RT–11 provides basic support for keeping time of day or time of year, depending on the processors. As distributed, RT–11 provides timer service with all multi-job and mapped single-job systems. Timer service for SB requires a system generation.

2.2.1 SB Systems Without Timer Service

As distributed, SB systems (without the timer feature) provide basic support for a system clock. Essentially, RT–11 keeps track of the time of day, but does not provide a means to implement mark time or timed wait requests.

The bootstrap routine looks for a clock on the system. If it finds one, it sets CLOCK$ (bit 100000) in the RMON configuration word ($CNFG1) at fixed offset 300. If the clock has a CSR (Control and Status Register), the bootstrap turns the clock on. If the clock does not have a CSR (as is the case with some LSI–11 and PDP–11/23 computers), no executing routine can turn the clock on or off; there may be a switch for the clock on the front panel.

RMON maintains the time of day in a two-word counter. The counter is called $TIME (high-order word) and $TIME+2 (low-order word). RT–11 stores time of day as the number of ticks since midnight if you set the time with the monitor TIME command. If you do not set the time, RT–11 stores the number of ticks since the system was last bootstrapped.

RT–11 supports KW11–L and similar line frequency clocks, and KW11-P programmable clocks. (Support for the programmable clock is a feature that you select through system generation.) The default interrupt frequency for the clocks is the same as the line frequency. That is, the clock interrupts 60 times per second with 60 Hz power, and 50 times per second with 50 Hz power. Each time the clock interrupts, it adds one tick to the two-word time of day counter.

In a simple system with a clock and no timer service, you can use the monitor TIME command to set the time of day or get the current time. A running program can use the .GTIM programmed request to obtain the current time, and .SDTTM to set it.

2.2.2 Systems with Timer Service

Timer service is supported in all multi-job systems and mapped single-job systems. It is a system generation special feature for SB systems. Timer service provides three extra programmed requests: the mark time request (.MRKT), the cancel mark time request (.CMKT), and the timed wait request (.TWAIT). In addition, another system generation special feature provides device time-out support through the time-out macro (.TIMIO) and the cancel time-out macro (.CTIMIO), which are described fully in *RT–11 Device Handlers Manual*.

To implement timer services, RT–11 uses a timer queue, which is a linked list of queue elements, sorted in order of expiration time. The element that expires soonest is at the head of the queue. The .MRKT, .TWAIT, and .TIMIO requests use the timer

queue. They schedule completion routines to be executed after a certain time interval elapses.

The monitor uses the timer queue internally to implement the .TWAIT programmed request, which causes the job that issues it to be suspended. The monitor places a timer request in the timer queue with the .RSUM programmed request code as its completion routine. The job waits until the specified time interval has elapsed. Execution resumes when the monitor itself issues the .RSUM request as a completion routine.

Table 2–2 shows the timer queue element. It includes the symbolic names and offsets as well as the contents of each word in the data structure. Note that time is stored as a two-word number; the number of ticks until the timed wait expires.

Table 2–2: Timer Queue Element Format (.QTIDF)

Name	Offset	Contents
C.HOT	0	High-order time
C.LOT	2	Low-order time
C.LINK	4	Link to next queue element; 0 if none
C.JNUM	6	Owner's job number
C.SEQ	10	Owner's sequence number ID
C.SYS	12	-1 if system timer element; -3 if .TWAIT element in mapped system
C.COMP	14	Address of completion routine

To store the time of day in all systems with timer support, RT–11 uses a two-word pseudoclock called PSCLOK (low-order word) and PSCLKH (high-order word). In this pseudoclock RMON stores the time, in ticks, that has elapsed since the system was bootstrapped. Each clock interrupt adds one tick to the counter. Two other words, $TIME and $TIME+2, contain a constant that, when added to the value of the pseudoclock, yields the current time of day.

The monitor uses the pseudoclock to implement timer requests. When a new queue element is put on the queue, the monitor adds the low-order word of the pseudoclock to the two-word time value in the queue element and it stores the resulting value, a modified time, in the queue element time words. Whenever the pseudoclock carries into the high-order word (approximately every 18 minutes), the monitor subtracts 1 from the high-order word of time in each pending timer queue element. The element expires when the high-order time word is 0 and the low-order time word is less than or equal to the pseudoclock low-order word. This method of storing time information means that handling timer requests requires only test and compare instructions, which execute rapidly, and a pass over the queue roughly every 18 minutes to correct the time words.

Every time the system clock interrupts, the monitor increments the pseudoclock. It then checks the first element in the timer queue. If the high-order word of the

timer element is 0 and the low-order word is greater than the low-order word of the pseudoclock, the element has expired. The monitor removes it from the timer queue and processes it as a completion routine for the correct job. The monitor continues to check the timer queue until it finds an element that has not yet expired or the queue is empty.

There are several uses for system timer elements. If C.SYS is –1, the element is being used by .TIMIO for device time-out support, or by RMON for multiterminal device time-out. If C.SYS is –3, the element is being used to implement a .TWAIT request in a mapped system. For .MRKT and other .TWAIT requests, C.SYS is 0.

In mapped systems, completion routines that have –1 in C.SYS are run in kernel mode and the queue element is discarded. That is, the queue element is not linked into the list of available elements. If C.SYS is –3, the completion routine is still run in kernel mode. However, the queue element is linked into the available queue when the completion routine is run. (The timer queue element is used as the completion queue element.) In all other cases, the queue element is linked into the available queue and completion routines run in user mode. (Chapter 3 provides more information on extended memory systems.)

2.3 Queued I/O System

RT–11 performs I/O transfers through a queued I/O system. A job can thus have multiple I/O requests outstanding at a given time—that is, it can issue an I/O request and still continue processing.

RT–11 implements queued I/O through the queue elements, the device handlers, and the routines in RMON. Once a device handler is in memory and the job has opened a channel, any .READ or .WRITE requests for the corresponding peripheral device are interpreted by the monitor and translated into a call to the handler. Figure 2–5 illustrates the relationship between these components.

2.3.1 I/O Channel Format

Table 2–3 shows the format of an I/O channel. Since each channel uses five words, the size of the monitor's channel area is five times the number of channels. RT–11 allocates 16 channels for each job. The channel area is 80_{10} words long. One channel area for each job is located in the job's impure area. The .CDFN programmed request can provide more channels. Table 2–4 shows the significant bits in the Channel Status Word.

Figure 2–5: Components of the Queued I/O System

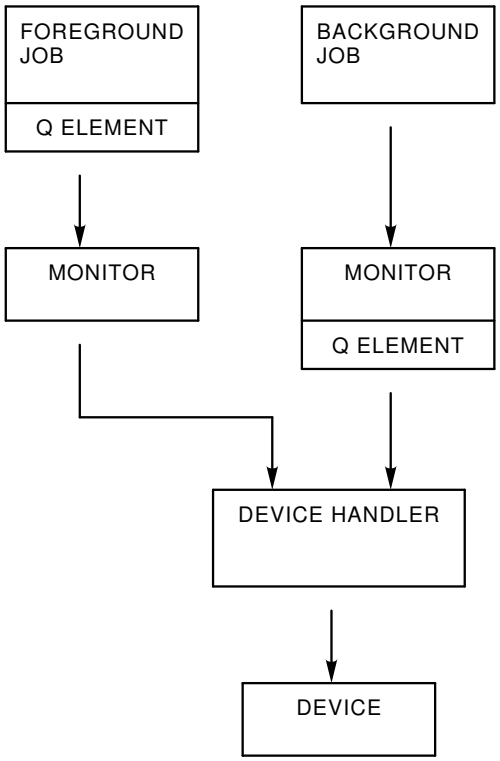


Table 2–3: I/O Channel Description (.CHNDF)

Name	Offset	Contents
C.CSW	0	Channel Status Word (See Table 2–4)
C.SBLK	2	Starting block number of this file (0 if non-file-structured)
C.LENG	4	Length of file (if opened by .LOOKUP) Size of empty area (if opened by .ENTER)
C.USED	6	Highest block written
C.DEVQ	10	I/O count (pending requests)
C.UNIT	11	Device unit number
C.SIZ	12	Symbol whose value is the size of block in bytes

Table 2–4: Channel Status Word (CSW), .CSWDF

Bit	Symbol	Meaning
0	HDERR$	Hard error bit. 0 = No error. 1 = Hard error.
1-5	INDX$M	Index into the $PNAME table and other device tables.
6	RENAM$	RENAME flag. 0 = No RENAME is in progress. 1 = A RENAME operation is in progress.
7	DWRIT$	0 = The file was opened with a .LOOKUP. The monitor does not modify the directory when the file is closed. 1 = The file was opened with an .ENTER. The monitor modifies the directory when the file is closed.
8-12	DBLK$M	The number of the directory segment containing this entry.
13	EOF$	End-of-file (EOF) bit. 0 = No end-of-file. 1 = End-of-file was found on this channel.
14	RONLY$	Write not allowed.
15	ACTIV$	0 = The channel is free. 1 = The channel is active.

2.3.2 I/O Queue (.QELDF)

The RT–11 I/O queue system consists of a linked list of queue elements for each resident device handler and a queue of available elements for each job. I/O queue elements are seven words long for unmapped systems, and 10_{10} words long for mapped systems. Each job has one queue element in its impure area. One queue element is sufficient for a job that uses wait-mode I/O.

Table 2–5 shows the format of an I/O queue element. It includes the symbolic names and offsets, as well as the contents of each word in the data structure.

Table 2–5: I/O Queue Element

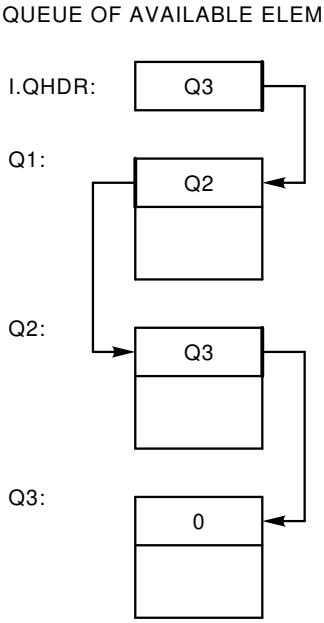
Name	Offset	Contents
Q.LINK	0	Link to next queue element; 0 if none
Q.CSW	2	Pointer to Channel Status Word in I/O channel
Q.BLKN	4	Physical block number
	6	Special function support; contents determined by device-unit support: If 8-unit handler: Q.FUNC Function code (8 bits) If Extended Device-Unit handler: Q.FUNC 4 low bits for special function Q.2UNI 3 bits (MSB) of unit number Q.TYPE 1 bit when set means special function
	7	Unit and job number: Q.UNIT 3 bits (LSB) of unit number Q.JNUM 4 bits job number; high bit reserved and always 0
Q.BUFF	10	User buffer address (mapped through PAR1 with Q.PAR or Q.MEM value, if mapped monitor)
Q.WCNT	12	Word count: if < 0, operation is WRITE if = 0, operation is SEEK if > 0, operation is READ The true word count is the absolute value of this word.
Q.COMP	14	Completion routine code: if 0, this is wait-mode I/O if 1, just queue the request and return if even, this is a completion routine address
Q.ELGH	16	Size of I/O queue element if not mapped
Q.PAR	16	UMR relocation constant (mapped monitors only)
Q.MEM	20	PAR1 displacement bias for MMU (mapped monitors only)
	22	Reserved (mapped monitors only)
Q.ELGH	24	Size of I/O queue element if mapped

If your program uses asynchronous I/O, you must allocate more queue elements for it by using the .QSET programmed request. Otherwise, if the program initiates an I/O transfer and no queue element is available, RT–11 must wait for a free element

before it can queue up the new request. Obviously, this slows processing. The number of queue elements is always sufficient when you allocate *n* new elements, where *n* is the total number of pending requests that can be outstanding at one time for a particular program. This produces a total of *n+1* available elements, since the original single queue element is added to the list of available elements.

The list header, called I.QHDR, is a linked list of free queue elements. It contains a pointer to an available queue element. If I.QHDR is 0, no elements are currently available. Figure 2–6 shows an I/O queue with three queue elements, all of which are available. In this diagram, I.QHDR points to element 1. The first word in each queue element is a pointer to the next element in the queue. Thus, element 1 is linked to element 2, element 2 is linked to element 3, and element 3 is the last element in the linked list; its link word is 0.

Figure 2–6: I/O Queue with Three Available Elements

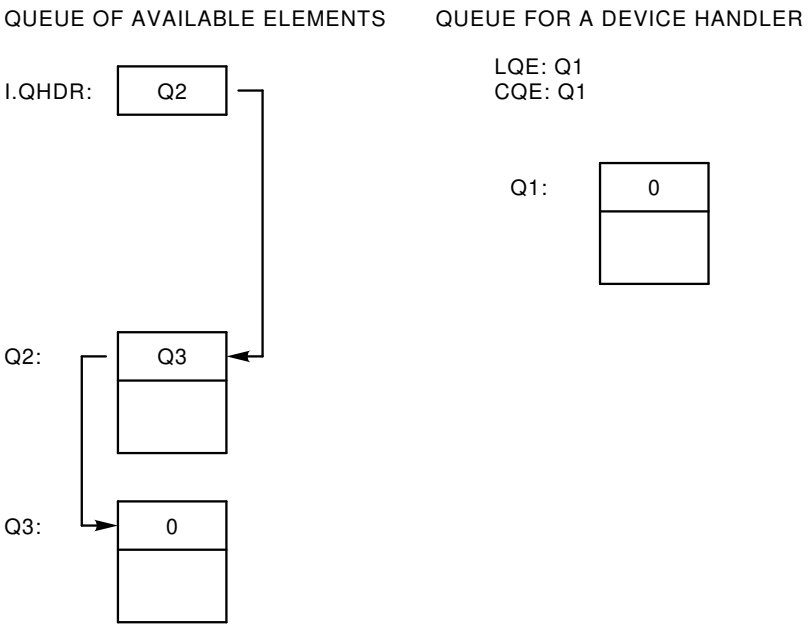


When a program initiates a request for an I/O operation, the monitor allocates a queue element for the request by removing it from the list of available elements. The monitor then links the element into the I/O queue for the appropriate device handler. This is accomplished by using two words in the handler header—ddLQE and ddCQE.

The fourth word of the handler is a pointer to the last element in its queue. This pointer is called *ddLQE*, where *dd* is the two-character physical device name. The fifth word of the handler, called *ddCQE*, is a pointer to the current queue element.

Figure 2–7 shows the status of the queue elements when one I/O request is pending. The monitor removes the first queue element from the available list and puts it on the device handler's queue.

Figure 2–7: I/O Queue with Two Available Elements



When a program requests a second I/O transfer for the same handler before the first transfer completes, the monitor removes another queue element from the available list and adds it to the queue for that handler. Figure 2–8 illustrates this.

When the transfer currently in progress completes, the monitor returns queue element 1 to the available list and initiates the transfer indicated by queue element 2. Figure 2–9 illustrates the queue status when one element is returned.

When the I/O operation indicated by queue element 2 finishes, the monitor returns that element to the available list, as Figure 2–10 indicates. Note that the elements are now linked in a different order from that shown previously in Figure 2–6.

In single-job systems, the monitor always puts the new queue element at the end of the device queue. By using ddLQE it can do this quickly. In multi-job systems, the device queue is sorted in order by job number, with the queue elements belonging to

Figure 2–8: I/O Queue with One Available Element

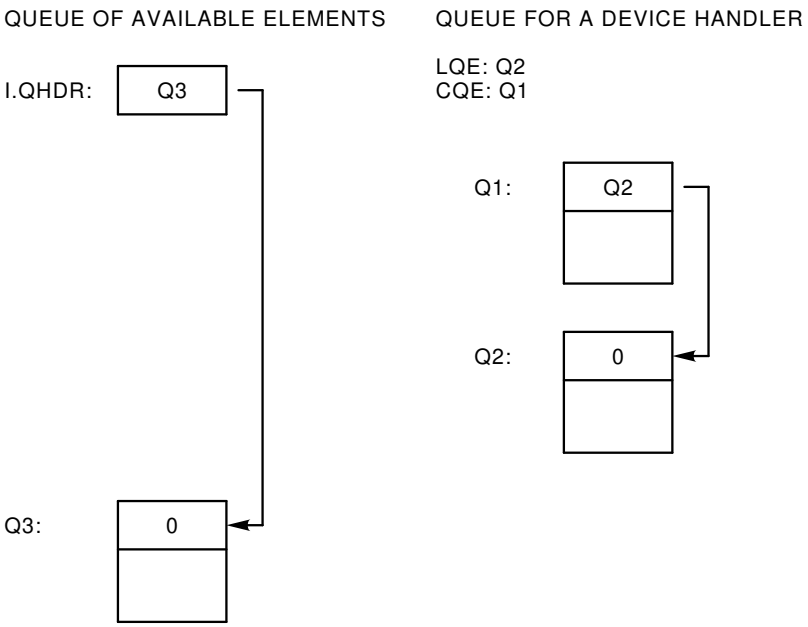
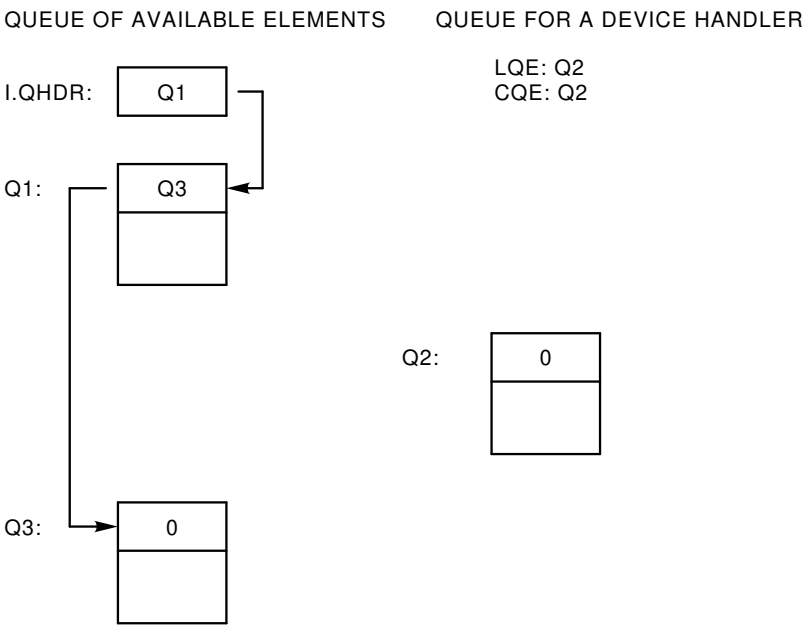
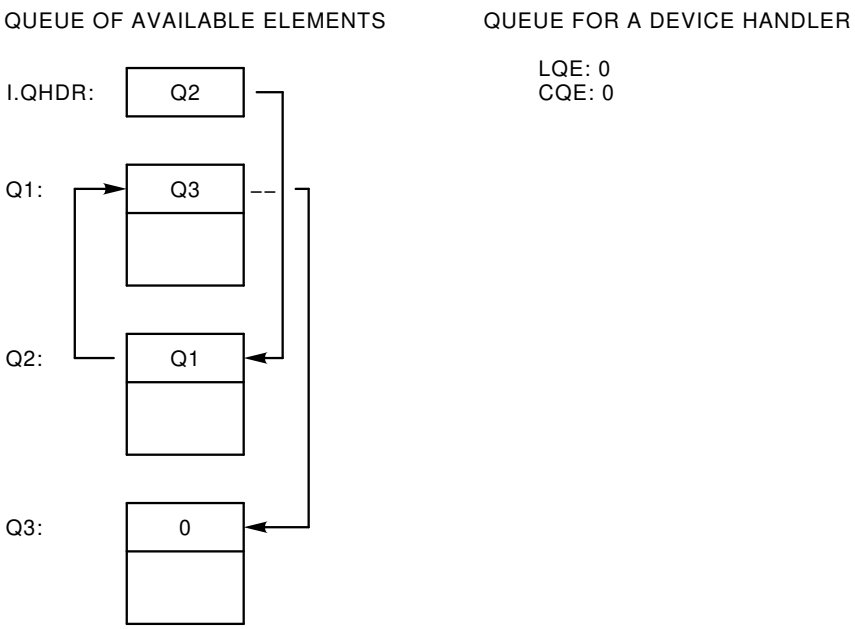


Figure 2–9: I/O Queue when One Element Is Returned



the highest job number appearing at the beginning of the queue and those belonging to the lowest job number at the end. The monitor puts the new element in the queue

Figure 2–10: I/O Queue when Two Elements are Returned



at the end of the list within a specific job group. Thus, if two requests are queued waiting for a particular handler, the request with the higher job number is honored first. At no time though, does the monitor abort an I/O transfer already in progress to start a higher priority request. The operation in progress always completes before the monitor initiates another transfer.

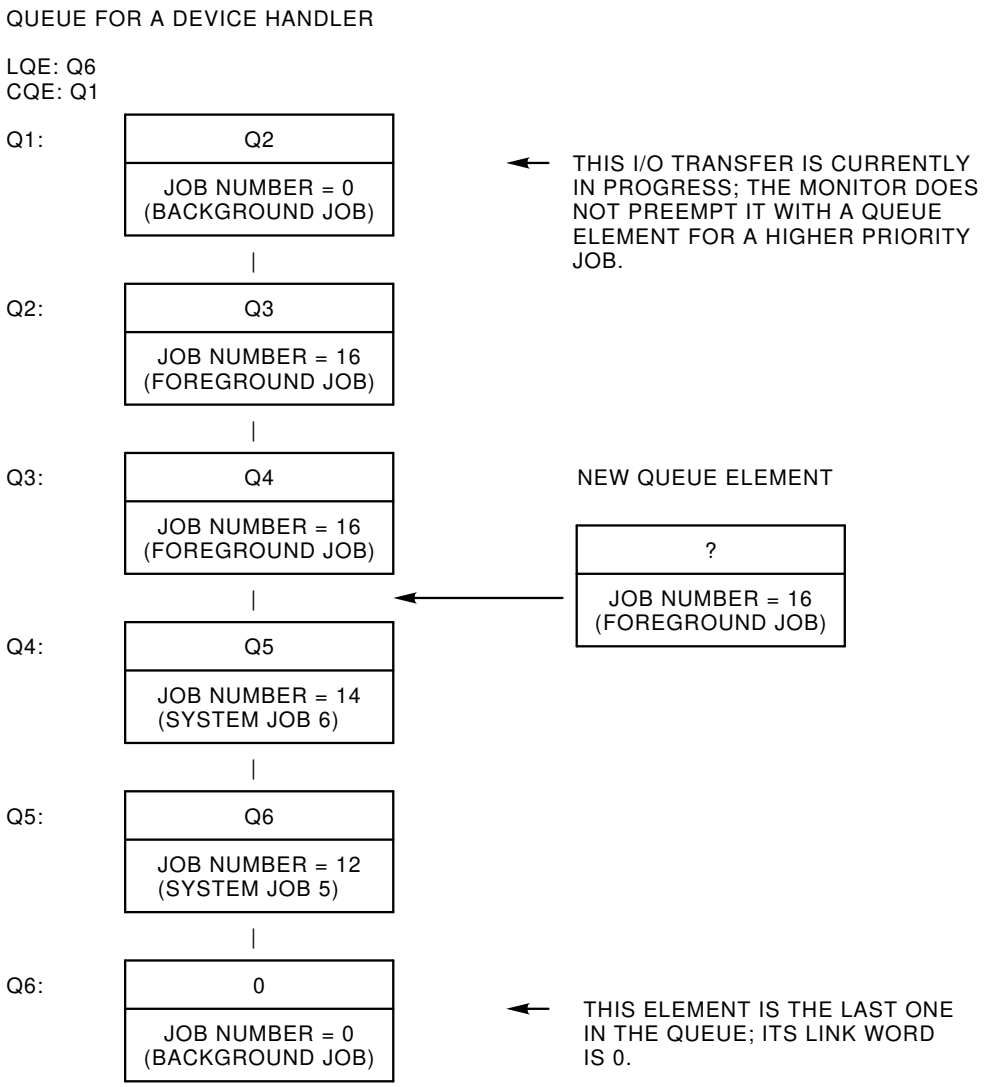
Figure 2–11 illustrates a large queue for a device handler. The monitor adds the new element, an I/O request from the foreground job, to the queue at the end of the list of other foreground job elements. Note that the monitor does not preempt the current queue element, even though it is a request from the background job.

2.3.3 Completion Queue

The monitor maintains a completion queue for a job (for each job in multi-job systems), using it to serialize completion routines for the job. The head of the completion queue is called I.CMPL and it is located at offset 6 from the start of the impure area. I.CMPE, at offset 4, points to the end of the completion queue. By using I.CMPE, the monitor can quickly add a new completion queue element to the end of the queue.

A completion routine is a section of code in a program that begins to execute as soon as an asynchronous event occurs. For example, the .READC programmed request starts an I/O transfer and provides the address in the program at which execution is to begin when the I/O transfer completes. See the *RT–11 System Macro Library Manual* for a thorough description of completion routines.

Figure 2–11: Device Handler Queue when a New Element is Added



When an I/O transfer completes, the monitor checks C.COMP at offset 14_8 from the start of the I/O queue element. If the value is greater than 1, it specifies a completion routine address. The monitor then transforms the I/O queue element into a completion queue element and places it on the completion queue for the job whose job number appeared in Q.JNUM at offset 7 from the start of the I/O queue element.

Table 2–6 shows a completion queue element. It includes the symbolic names and offsets, as well as the contents of each word in the data structure.

Table 2–6: Completion Queue Element Format (.QCMDF)

Name	Offset	Contents
QC.LNK	0	Link to next queue element; 0 if none
	2-6	Reserved
QC.CSW	10	Channel Status Word
QC.OFT	12	Offset from start of channel area to this channel
QC.CMP	14	Completion routine address

2.3.3.1 .SYNCH Considerations

The .SYNCH request also makes use of the completion queue, but it does not use an I/O queue element. When you issue a .SYNCH call, you supply as an argument the address of a seven-word area in your program, called the synch block. The synch block contains, among other things, the address of the routine to be executed. Table 2–7 shows a synch block, or synch queue element. When the monitor interprets your .SYNCH request, there is no current I/O queue element for it to modify. So, it uses your seven-word area as a completion queue element. The monitor puts the synch block at the head of the appropriate job's completion queue.

Table 2–7: Synch Queue Element (.QSYDF)

Name	Offset	Contents
QS.LNK	0	Link to next queue element; 0 if none
QS.JOB	2	Job number
	4-6	Reserved
QS.ID	10	Synch ID
QS.SYN	12	-1 (cue that this is a synch element)
QS.CMP	14	Synch routine address

2.3.4 Flow of Events in I/O Processing

As the central manager of the device-independent I/O system, RMON supervises the I/O procedure, using a queue element as the communication link between a device handler and a program that requests an I/O transfer. The following sections describe the sequence of events that occur in a simple read or write operation.

2.3.4.1 Issuing the Request

Before a program can request an I/O transfer, it has to open a new file or find an existing file on a device. This procedure sets up a channel containing five words of information about the location and length of the file. A channel number is associated with the five-word block so that you can refer to the block later by specifying this number in a single byte. The monitor uses the channel information when it needs to process an I/O request.

A running program initiates an I/O procedure by issuing a request to read from or write to a particular channel. MACRO–11 programs, for example, can use the .READ, .READW, .READC, .WRITE, .WRITW, .WRITC, and .SPFUN programmed requests. Programs written in other languages use similar statements to read and write data.

When the I/O request executes, the monitor uses the channel number the request specifies to find the corresponding device handler. Then the monitor calls its queue manager routine, which allocates a queue element from the list of available elements and fills in the necessary information.

When a queue element is not available in a single-job system, the monitor executes in a tight loop, waiting for a queue element to appear in the list of available elements. This condition is satisfied when a device interrupts and the handler issues the .DRFIN macro, which indicates that an I/O transfer is complete, and the monitor returns the queue element for that transfer to the available list.

When a queue element is not available in a multi-job system, the job requests a scheduling pass, starting with the job whose priority is immediately below that of the current job. When the original job gets a chance to run again, it first checks the available list for a free queue element. If no element is available, it requests another scheduling pass. In FB systems, there is no blocking bit associated with queue element availability. Therefore, the job that needs a queue element is not officially blocked, even though it cannot run effectively until it gets a queue element.

2.3.4.2 Queuing the Request

All jobs (system utility programs, application programs, and language processors) and KMON run in *user state*. Each job uses its own stack. In user state, a low-priority job that is running can be replaced by a higher-priority job that is runnable. Similarly, a higher-priority job that is unable to run for any reason can be replaced by a runnable lower-priority job. In single-job systems, there is only one (background) job. When it is unable to run (or when no job is able to run in a multi-job system), the monitor runs the null job, which consists of waiting for I/O to complete.

The monitor switches to *system state* to modify important data structures and to perform operations that do not run entirely within a job. Stack operations and interrupts in system state use the monitor's stack rather than a job's stack. Jobs cannot run when the monitor is in system state, and switching between lower- and higher-priority jobs is postponed until the monitor returns to user state. In system state, then, the monitor can safely modify critical data structures without the risk that another job could gain control and corrupt the same data structures. (Section 2.4.1 describes system and user state in greater detail.)

The monitor switches to system state before it puts the new element on the device handler's queue in order to prevent interference from other jobs. However, a device interrupt could remove an element from the queue while the monitor is adding the new element and adjusting the LQE and CQE pointers. To ensure the integrity of the queue, the monitor *holds* the handler while it performs the modification.

Holding a handler prevents any other process or routine from changing the I/O queue. For example, when a device interrupts and an I/O operation completes, the handler issues a .DRFIN call to return to the monitor and remove the current queue element from the I/O queue. Depending on the type of I/O request the program issued, the current element should either go back to the linked list of available queue elements, or it should go onto the completion queue for the appropriate job. However, if the handler is held when it issues the .DRFIN, the monitor does not remove the current queue element from the I/O queue. Instead, it delays this action by setting a flag that it checks later. Similarly, when a job aborts, the abort routine holds a handler while it removes queue elements belonging to the aborted job. This prevents the monitor from starting up the next transfer queued for this device until all elements for the aborted job are gone. After the monitor holds the device handler, it checks to see if the queue is empty.

If the queue is empty, the monitor immediately clears the hold flag for the handler, and then makes the new element both the current and the last element in the queue. It increments both the count of queue elements on this channel (the C.DEVQ byte at offset 10_8 in the channel area) and the total number of I/O requests for this job. Remaining in system state, the monitor jumps to the device handler's I/O initiation section to start up the transfer. When the handler starts the transfer and returns control with a RETURN instruction, execution of the program continues in user state within the queue manager. That is, the monitor is executing ''for the program''.

If the queue is not empty, the monitor continues to hold the handler until it finishes modifying the queue. In single-job systems, the monitor puts the new element at the end of the queue. In multi-job systems, elements in the queue are sorted by job number, as Section 2.3.2 explains. The monitor searches the queue from front to back, and places the new element at the end of the group of elements belonging to this job. It increments both the count of queue elements on this channel (the C.DEVQ byte at offset 10_8 in the channel area) and the total number of I/O requests for this job in I.IOCT. (For an exception to this, see the abort processing information in the *RT–11 Device Handlers Manual*.) Since the device handler is busy, the monitor cannot start up an I/O transfer for this request, so its queue element sits in the queue. The queue manager returns to user state.

Whether or not the queue was empty, the queue manager checks to see if this request is for wait-mode I/O. If so, the program waits for the transfer to complete. If this request is not for wait-mode I/O, execution of the program continues concurrently with the I/O transfer.

2.3.4.3 Performing the I/O Transfer

After the monitor and a device handler have started up an I/O transfer, a peripheral device performs the actual operation and interrupts when it is finished. The interrupt causes control to pass to the device handler's interrupt service section, where the code assesses the results of the I/O operation and restarts it if necessary. When the transfer is done, the handler uses the .DRFIN macro to return to the monitor and remove the current queue element from its I/O queue.

Figure 2–12 summarizes the relationship between the parts of a device handler and RMON. *RT–11 Device Handlers Manual* provides a detailed description of the internal operation of a device handler.

Figure 2–12: Device Handler/RMON Relationship

2.3.4.4 Completing the I/O Request

When a device interrupts, an I/O transfer completes, and the handler issues the .DRFIN call, it is the monitor that must take the appropriate action to complete the I/O procedure. In general, this means that the monitor must remove the current queue element from the handler's I/O queue and put it in the list of available elements or on the completion queue. Another I/O request could cause the monitor to hold the handler while it adds an element to the queue. In this case, the monitor

simply sets a flag, dismisses the interrupt, and returns to the interrupted process, removing the element later.

When the handler is not held, the monitor first decrements the count of queue elements on this channel. When the count reaches 0, it makes runnable a job that is waiting for activity on this channel to complete. The monitor next decrements the total number of I/O requests pending for this job. Again, if this number becomes 0, it makes runnable a job that is waiting for all its I/O to complete. When either count reaches 0, it can cause the scheduler to run.

Next, the monitor removes the queue element from the handler's queue. If there is another element in the handler's queue waiting to be processed, the monitor calls the handler again to start the next operation as soon as the final disposition of the current element is resolved. The monitor raises the priority to 7 for a short time as it links the element into either the list of available elements or the job's completion queue. If the element specifies a completion routine address at Q.COMP (offset 14_8), the monitor transforms the I/O queue element into a completion queue element and puts it at the end of the job's completion queue. Then, the monitor returns control to the process or program that was interrupted.

If the element does not specify a completion routine address, the monitor simply returns the element to the available list. Control returns to the process or program that was interrupted, or the scheduler can run.

2.4 Scheduling in Multi-Job Systems

In a multi-job system, the monitor must arbitrate the demands of up to eight jobs for processor time, in addition to performing all its other functions. Multi-job monitors use a number of special tools to implement support for more than one job. These tools exist in single-job systems, but many of them are considerably simpler. For example, the code for context switching essentially disappears in the single-job monitor.

The *scheduler* is the part of the monitor that determines which job is eligible to run and gives control of the processor to it. The scheduler uses a simple algorithm to determine which job should run. It looks at the jobs in order from highest priority to lowest. If a job exists and is runnable, the monitor restores its context and returns to it. Status bits in a flag word (I.BLOK, at offset 36_8 from the start of the impure area) reflect the *blocking conditions* that can prevent a job from running and thereby give a lower-priority job a chance to execute. *Context switching* is the procedure through which the monitor saves a job's *context* – its machine environment and important job-specific information—and begins execution of another job.

All the processes that are job-dependent are kept separate from those that are monitor functions. The monitor functions are, therefore, re-entrant. Data structures that contain job-specific information are located in the *impure area* for each job, and each job has its own stack. Routines that run in a job-dependent environment, including some parts of the monitor, use the job's stack and run as part of the user job in *user state*. Any routines that run outside a job's context, including interrupts, use the monitor's stack and execute in *system state*. This arrangement allows the

monitor to ‘‘unwind’’ the stack after a series of interrupts without changing jobs or stacks.

Two or more jobs can share a peripheral device, so the queued I/O system (as Section 2.3 explains) must keep track of the priority of the job requesting an I/O transfer and act accordingly. The USR is serially reentrant—that is, it cannot be shared by two jobs; all jobs must take turns using the USR.

Lastly, *monitor routines* check for blocking conditions, change execution state, interlock parts of the monitor to prevent corruption of important data structures, request a scheduling pass, and so on. The following sections describe the components of the monitor and provide an understanding of the scheduling process in a multi-job environment.

2.4.1 User and System State

In order to isolate job-dependent functions from monitor processes, the monitor provides two execution states: user state and system state. All jobs and KMON run in user state. Each job maintains relevant data in its impure area and uses its own stack. Context switching is enabled in user state. That is, a lower-priority job that is running can be replaced by a higher-priority job that is runnable. A higher-priority job that is unable to run for any reason can be replaced by a runnable lower-priority job.

The monitor switches to system state and the system stack for several reasons. Jobs cannot run when the monitor is in system state, and context switching is delayed until the monitor returns to user state. Consequently, the monitor can modify important data structures in system state without interference from other jobs. The monitor uses system state for operations that do not run entirely within a job context. These operations, which must not be interrupted by context switching, include the following:

- Blocking a job
- Starting up an I/O transfer
- Aborting an I/O transfer
- Servicing a timer request
- Executing the .PROTECT programmed request
- Executing the .CHCOPY programmed request
- Interlocking the USR
- Executing any mapping programmed request
- Servicing an interrupt
- Executing device handler code (except for .TIMIO completion routines and .SYNCH routines, which run in user state in a specific job’s context)

Because it is chiefly system or monitor routines that execute in system state, monitor errors are fatal. Traps to 4 (odd address errors, and illegal or nonexistent memory

addressing errors) and traps to 10 (illegal or reserved instruction errors), occurring in system state, halt the system.

2.4.1.1 Switching to System State Asynchronously

The monitor switches from user state to system state asynchronously whenever an interrupt occurs. As a result of the interrupt the monitor may modify important data structures. The switch to system state prevents interference from a context switch while the modifications are in progress. In unmapped systems, the monitor switches from the job's stack to the system stack. In mapped systems, the monitor does not perform the stack switch because the hardware does it automatically. Subsequent interrupts that occur in system state put information on the system stack. Note that these subsequent interrupts do not cause another switch to system state.

Interrupt Level Counter

The monitor recognizes three levels of execution state. It uses a counter called INTLVL to distinguish among the three levels. Every interrupt increments this counter. When INTLVL is -1, execution is in user state. When INTLVL is 0, execution is in system state at level zero. When INTLVL is positive, execution is still in system state, but at a deeper interrupt level. Table 2–8 summarizes the relationship between the number of interrupts pending and the execution state.

Table 2–8: Values of the interrupt Level Counter (INTLVL)

Number of Interrupts	Value of INTLVL	Execution State
0	-1	User State
1	0	System State Level Zero
2 or more	1 or greater	Deeper System State

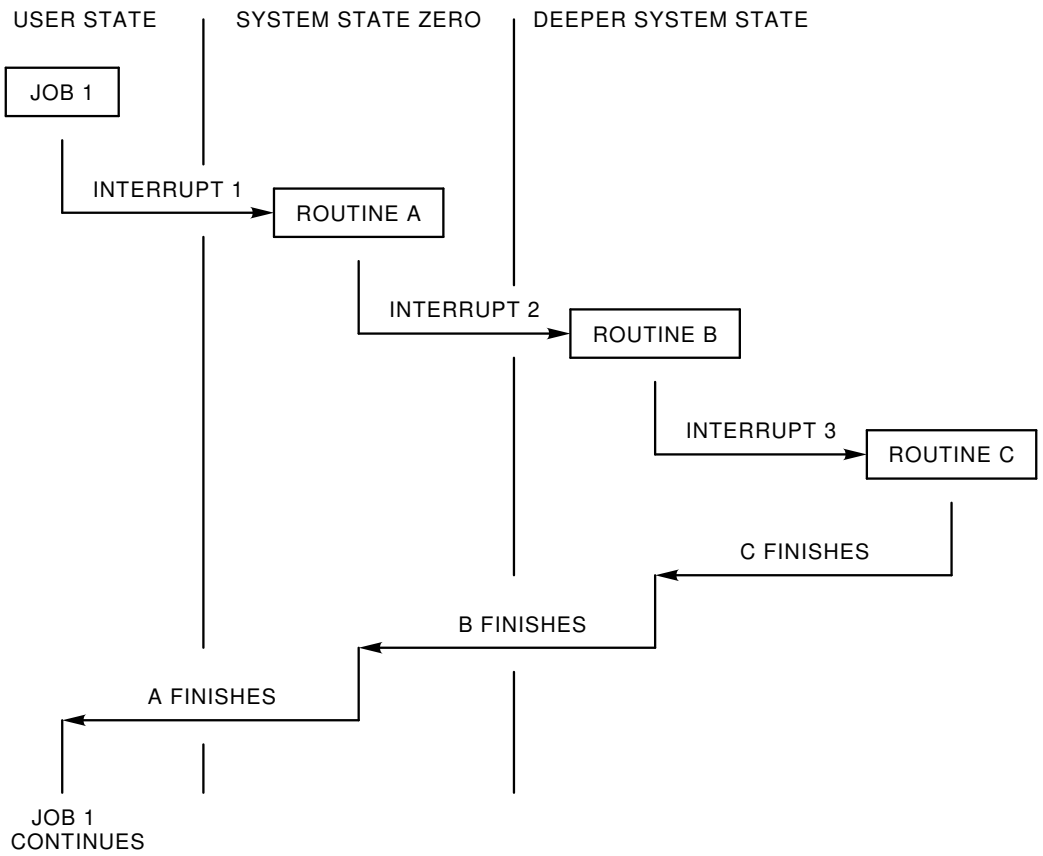
Figure 2–13 shows how interrupts influence the flow of events in a running system.

$INTEN Monitor Routine

When an interrupt occurs, control passes to the routine specified in the interrupt vector, and the current PS and PC are put on the job's stack. In RT–11, both device handlers and in-line interrupt service routines call the monitor at the common interrupt entry point, $INTEN. Device handlers use the .DRAST macro to call the monitor; in-line interrupt service routines use the .INTEN macro.

$INTEN is the monitor routine that performs the switch to system state. The routine assumes that it was called because an interrupt occurred. Therefore, it expects the old PS and PC to be on the job's stack. The priority should be 7, and the interrupt service routine must not have destroyed any registers between the time the interrupt occurred and the time $INTEN was called. Device handlers generally

Figure 2–13: Interrupts and Execution States



call the monitor immediately, before any processing is done. In-line interrupt service routines sometimes perform crucial operations immediately, at priority 7, then call $INTEN to lower processor priority to device priority.

$INTEN assumes it was called with the following instruction sequence, or its equivalent:

```
JSR       R5,@$INTEN
  .WORD ^C<priority*40>&340
```

$INTEN's first action is to save R4 on the job's stack. Since the JSR instruction already saved R5, the job's stack now appears as shown in Table 2–9.

Table 2–9: Job's Stack After $INTEN

Byte Offset	Contents	Agent
0	R4	$INTEN
2	R5	.DRAST macro (JSR R5)
4	PC	Interrupt
6	PS	Interrupt

Next, $INTEN increments the INTLVL counter from -1 to 0. For unmapped systems, it saves the job's stack pointer in a memory location and switches to the system stack. $INTEN then lowers processor priority to device priority, and calls the device handler or interrupt service routine back as a coroutine. The interrupt service routine continues to execute in system state.

2.4.1.2 Switching to System State Synchronously

The monitor switches to system state synchronously—that is, without depending on an interrupt—whenever other monitor routines need to go to system state temporarily to ensure the integrity of a certain operation. In these circumstances, the monitor routines can call the $ENSYS routine to switch to system state.

In special circumstances, a routine in a running privileged job (rather than in the monitor) needs to switch to system state. The routine can do this by artificially mimicking an interrupt and using the .INTEN macro to call the $INTEN monitor routine.

$ENSYS Monitor Routine

The $ENSYS routine is voluntarily and synchronously called by any other monitor routine that needs to switch to system state. $ENSYS mimics an interrupt by altering the job's stack so it duplicates the stack condition immediately after an interrupt. Routines call $ENSYS by using the following instructions:

```
JSR     R5,$ENSYS
.WORD   <return address>-.
.WORD   340
```

The instructions following the call to $ENSYS execute in system state. When the routine that must execute in system state completes, it issues an RTS PC instruction. Control then passes in user state to the routine specified in the calling sequence as *<return address>*.

Table 2–10 shows how $ENSYS manipulates the stack to imitate an interrupt.

Table 2–10: Job's Stack After $ENSYS

Byte Offset	Contents
0	R5
2	Return address
4	0

.INTEN Macro

When a routine in a user privileged job needs to switch to system state, it can use a procedure similar to $ENSYS, which is used solely by monitor routines. Essentially, the routine must push the PS and PC onto the stack, and then call the monitor $INTEN routine with a JSR R5 instruction, which puts R5 on the stack as well.

A device handler or a user program subroutine can use the following instructions to switch to system state:

```
MOV     @SP,-(SP)       ;MAKE ROOM ON THE STACK
CLR     2(SP)           ;FAKE INTERRUPT PS = 0
.MTPS   #340            ;GO TO PRIORITY 7
.INTEN  0,PIC           ;ENTER SYSTEM STATE
```

This routine must be executed with a return address on the top of the stack.

2.4.1.3 Returning to User State

Any routine that is executing in system state issues an RETURN instruction when it completes. The monitor ''unwinds'' its stack from one or more interrupts as each RETURN instruction is issued. As each routine completes, the monitor decrements the INTLVL counter.

When INTLVL is greater than 0, it indicates that the routine that was just interrupted was executing in system state. The monitor defers some special chores until it is just about to return to user state. If it is time to decrement INTLVL after a RETURN instruction, and the value of INTLVL is currently 0, the monitor knows that it is about to drop back to user state. At this time, there are four special considerations for the monitor:

- Is there an outstanding fork routine? (Fork routines run before jobs or their completion routines.)
- Is a scheduling pass required? (As a result of an interrupt, a job that was previously blocked may now be runnable.)
- Are there outstanding clock ticks? (The monitor may need to normalize its time of day counter and check the timer queue.)
- Is there an outstanding floating-point interrupt?

After taking these considerations into account, the monitor is ready to return to user state. It decrements INTLVL to -1 and in unmapped systems, switches to the

appropriate job's stack. It restores R4 and R5, and then executes the RTI instruction to begin execution in user state.

2.4.2 Context Switching in Multi-Job Systems

There is no need for context switching in single-job systems, because there is only one possible job to run. In multi-job systems, context switching occurs as a result of the scheduler's command to run a different job. Its purpose is to restore the context for a job so that it can run. Context switching can occur for one of two reasons:

- The current job becomes blocked and a lower-priority job is runnable.
- A higher-priority job than the current job becomes runnable.

Note that the RT–11 monitor never saves a job's context simply because it switches to system state. For example, if there is only one job running, the monitor does not bother to save or restore its context. A job's context is only significant when there are two or more jobs running. RT–11 avoids the overhead of unnecessary context switching by saving and restoring the context only when it runs a different job. This is a significant saving because there are many situations in which a job is running, an interrupt triggers a switch to system state, and control passes back to the same job once the interrupt is serviced.

When the monitor saves a job's context, it preserves (does not modify) the job-dependent information on the job's stack and in the job's impure area. For mapped monitors, it saves the mapping context information in the Mapping Context Area (MCA) region in extended memory. The following information is preserved and saved in a context switch:

All Monitors

- PS
- PC
- Stack Pointer (saved in the impure area)
- Registers R0 through R5
- BPT vector
- IOT vector
- TRAP vector
- System communication area (locations 4-0-52)
- Location 56 (multiterminal systems only)
- FPP status word and floating-point registers (if floating-point hardware is present)
- All data specified by the program in a .CNTXSW programmed request
- Stack and impure area (preserved)

Mapped Monitors

- Kernel PAR1
- Memory management fault trap vector
- PARs and PDRs
- Region control blocks and window control blocks (preserved)
- Memory management register 3 (MMR3)

See Section 3.4.4 for a description of mapping context switching for virtual and privileged jobs.

When the monitor switches in the new job's context, it tests for a pending completion routine by checking a status bit in I.STATE. If the job's completion queue has a completion queue element on it, the monitor puts a pseudointerrupt on the job's stack to call the completion queue manager when the scheduler actually starts up the job.

2.4.3 Blocking Conditions

A running job is *blocked* if it cannot proceed until some asynchronous event happens. Table 2–11 lists the blocking conditions, the bits in I.BLOK (at impure area fixed offset 36_8) that reflect the conditions, and the events that unblock a job. Unused bits are reserved for future use by Digital.

Note that there is no bit that indicates that a job is waiting for a queue element. This is a special case and the monitor handles it by checking the list of available queue elements. If there are none, it requests a scheduling pass to give a lower-priority job a chance to run. The monitor continues to check the available list until a queue element becomes available.

Job blocking exists in all systems. When a single-job system is blocked, the system idles (runs the null job) until some asynchronous event unblocks the job.

Table 2–11: Blocking Conditions

Blocking Agent	I.BLOK Bit, Name, and Mask	Unblocking Agent
Any request that uses the USR; any monitor command; an exit from a background job.	4 USRWT$ 20	The USR release routine, DEQUSR, when the USR is free and no higher-priority job needs it.
The keyboard monitor SUSPEND command.	6 KSPND$ 100	KMON, when an operator issues the RESUME command.
The .EXIT request; a job that aborts.	8 EXIT$ 400	I/O completion from device handlers, when the job's total I/O count is 0.

Table 2–11 (Cont.): Blocking Conditions

Blocking Agent	I.BLOK Bit, Name, and Mask	Unblocking Agent
Termination of the foreground or system job.	9 NORUN$ 1000	None. Only KMON can clear this bit by removing the job image from memory.
The .SPND or the .TWAIT programmed request.	10 SPND$ 2000	The monitor's .RSUM processor, when the .RSUM request executes or a .TWAIT completion routine runs.
The .READW, .WRITW, .WAIT, .SDATW, .RCVDW, .MWAIT, and wait mode SPFUN programmed requests.	11 CHNWT$ 4000	I/O completion from device handlers, when the I/O count for the specified channel is 0.
The .EXIT programmed request issued from a foreground or system job; the .MTSET request issued for a DZ line; .MTDTCH issued for any terminal but a shared console.	12 TTOEM$ 10000	The monitor's terminal service output routine, when the output ring buffer is empty or CTRL/O is typed.
The .TTYOUT, .PRINT, .MTOUT, and .MTPRNT programmed requests.	13 TTOWT$ 20000	The monitor's terminal output interrupt service routine, when there is room in the output ring buffer.
The .TTYIN request (with JSW bit 6 clear); the .CSIGEN, .MTIN, .CSISPC, and .GTLIN programmed requests.	14 TTIWT$ 40000	The monitor's terminal input interrupt service routine, when a line or character is available.
Any request that needs a queue element when none is available.	none	The monitor's queue element return routine, when a queue element becomes free.

2.4.3.1 How the Monitor Blocks a Job

A job becomes blocked when it encounters any of the circumstances listed in Table 2–11. These circumstances are brought about when one of the three following events occurs:

- The job issues one of the programmed requests listed in Table 2–11.
- The SUSPEND command is issued.
- The job aborts.

Typically, the job, which is running in user state, issues a programmed request, such as .EXIT. The monitor remains in user state while it processes the programmed request. It then checks to see if the job is waiting because of a blocking condition. The .EXIT request, for example, must wait for all the job's I/O requests to complete before it actually terminates the job. Since waiting for all I/O to complete is a blocking condition, the monitor initiates the appropriate test to see if there are outstanding I/O requests and this job is now blocked.

The monitor calls its $SYSWT routine whenever it needs to determine whether or not a job is blocked. The monitor passes to $SYSWT a bit mask for the bit in I.BLOK corresponding to this particular condition. (Table 2–11 lists the bit masks for I.BLOK; bit 8 corresponds to the .EXIT request condition.) It also passes a *decision subroutine*, which is a routine that determines whether or not a job is blocked for a particular reason. There is a unique decision subroutine for each call to $SYSWT, except the waiting for a queue element condition, which has none. The decision subroutine returns with the carry bit set if the job is indeed blocked. Note that a job can be blocked for only one reason at a time.

When control eventually returns to the job, it executes within the monitor in user state at $SYSWT again. (That is, the monitor runs under the auspices of the job, executing code on its behalf.) The blocking condition must be checked once more in order to reblock a job that may have been unblocked to allow a completion routine to run. (Completion routines are part of a job, but they can run even if the main part of the job is blocked. The monitor unblocks the job to run the completion routine, then runs $SYSWT to reblock the job when the completion routine finishes. Section 2.4.5 discusses the implications of completion routines for scheduling.)

2.4.3.2 $SYSWT Monitor Routine

$SYSWT is the monitor routine that decides whether or not a job is blocked. If a job is blocked, $SYSWT sets the appropriate blocking bit. The flowchart in Figure 2–14 shows how $SYSWT works.

First, $SYSWT runs the decision subroutine passed by the monitor to determine whether or not the job is blocked for a specific reason (point *A* in Figure 2–14). If the job is not blocked, control returns to the job and it continues to run (point *B*). In the .EXIT case, for example, a job is not blocked if there is no pending I/O to delay the exit procedure.

If the job is blocked, $SYSWT calls $ENSYS to enter system state (point *C*). Then it sets the appropriate blocking bit. In the .EXIT example, a job is blocked if there are pending I/O requests; $SYSWT sets the EXIT$ bit, bit 8, in I.BLOK.

Next, $SYSWT runs the decision subroutine again. If the job is still blocked, $SYSWT requests a scheduler pass (point *E*). It does this to give a runnable lower-priority job a chance to execute.

If the job is no longer blocked, $SYSWT clears the blocking bit and returns (point *F*). When the monitor switches back to user state, the scheduler runs if a scheduling pass is pending. When control finally returns to this job (the one for which $SYSWT originally ran), the monitor continues execution on the job's behalf at the beginning of the $SYSWT routine (point *A*).

$SYSWT runs the decision subroutine twice because interrupts can occur while $SYSWT is running. Since an interrupt can signal the removal of a blocking condition, the job's status can change even as $SYSWT is trying to determine it.

Figure 2–14: $SYSWT Monitor Routine

An interrupt can occur after the decision subroutine (point *A*) declares a job to be blocked, but before $SYSWT sets the blocking bit. This time interval is shown as ''Window 1'' in Figure 2–14. In this situation $SYSWT sets the blocking bit erroneously. But, when it runs the decision subroutine the second time, it discovers that the job is not blocked anymore. $SYSWT clears the bit and returns to the job (point *F*).

''Window 2'' in Figure 2–14 indicates the second time interval in which an interrupt can occur. The interrupt can remove the blocking condition immediately after $SYSWT correctly sets the blocking bit. In this case, the monitor's UNBLOK routine

clears the blocking bit and requests a scheduling pass because this job became runnable. Control returns to $SYSWT (point *D*), which runs the decision subroutine again. Since the job is no longer blocked, execution leaves $SYSWT (point *F*) and the scheduler runs immediately before the monitor returns to user state.

2.4.3.3 How the Monitor Unblocks a Job

An asynchronous event initiates the monitor's procedure to unblock a job. Table 2–11 lists the significant events that can unblock a job. The completion of all I/O for a specific channel is a significant event, for example, and unblocks a job whose CHNWT$ bit is set.

When an interrupt occurs, control passes to an interrupt service routine. The interrupt routine enters system state by executing the $INTEN monitor routine. Then, the interrupt service routine assesses the meaning of the interrupt and takes appropriate action. In a device handler, for example, an interrupt can indicate that an I/O transfer is complete. The handler returns to the monitor to remove the current element from the I/O queue.

In all cases, the monitor clears the blocking bit and requests a scheduling pass if the significant event removes a blocking condition.

2.4.4 Scheduler Operations

The scheduler runs only if there is an outstanding request for a scheduling pass. The monitor checks a flag byte called INTACT each time it is ready to switch from system to user state. If INTACT is not equal to zero, the scheduler runs. The scheduler exists in all systems. In single-job systems, the scheduler decides only whether to run the single background job, or to run the null job.

2.4.4.1 How the Monitor Requests a Scheduling Pass

The monitor requests a scheduling pass by calling the $RQTSW monitor routine. It does this whenever a job's ability to run changes. (That is, whenever a running job becomes blocked, or whenever a blocked job becomes runnable.)

2.4.4.2 Characteristics of a Runnable Job

A job that does not have any blocking bit set is runnable. However, there is one circumstance in which a job with a blocking bit set can still be runnable. A job's completion routine can run even though the mainline program is blocked. Section 2.4.5 discusses scheduling implications for completion routines.

2.4.4.3 $RQTSW Monitor Routine

The $RQTSW routine posts a request for a scheduling pass for a specific job by placing a value in the flag byte, INTACT. INTACT holds the job number of the highest-priority job that requested a scheduling pass. $RQTSW ignores a scheduling request for a job if its priority is lower than that of the running job. When a job whose priority is higher than that of the running job requests a scheduling pass, $RQTSW saves the job's number in INTACT, which holds the number in the following format:

```
          Job number
INTACT = ---------- + 200
             2
```

2.4.4.4 How the Scheduler Works

The scheduler runs just before the monitor returns to a job. Remember that INTLVL, the interrupt level counter, is 0 when it is time to return to user state.

A scheduling pass needed to make a job runnable happens asynchronously, as a result of an interrupt that removed a blocking condition. A scheduling pass needed to make the current job nonrunnable happens synchronously, after a job issues a programmed request, after the SUSPEND command is issued, or after a job aborts.

The scheduler runs only if INTACT is not equal to 0. When INTACT is 0, it indicates that no job changed its status, and, therefore, the same job that was interrupted should run again. When INTACT is not 0, it contains the number of the highest-priority job that changed its status. The scheduler runs only if the job number in INTACT is greater than the current number of the current job, which is kept in JOBNUM in the monitor.

The scheduler examines jobs in order of descending priority. It starts with the job whose number is in INTACT, which is not necessarily the highest-priority job in the system. As soon as the scheduler finds a runnable job, the monitor switches context and runs the job. If no jobs at all are runnable, the system idles—that is, it runs the null job briefly, then scans all jobs again for runnability.

2.4.5 Implications for Completion Routines

A job's completion routine can run even though the mainline program is blocked. When an asynchronous event occurs, such as the completion of an I/O request, the interrupt service routine enters system state through the $INTEN monitor routine. The device handler's interrupt service routine returns to the monitor when I/O completes, so the monitor can remove the I/O queue element from the device handler's queue. If the I/O request specified a completion routine address, the monitor changes the I/O queue element into a completion queue element and puts it on the job's completion queue. The monitor sets CPEND$ (bit 7) in the job state word (I.STATE, the first word in the job's impure area) to indicate that a completion routine is pending.

As the monitor switches from system to user state, it checks the completion pending bit in I.STATE in the job's impure area. If a routine that just ran in system state queued one or more completion routines for this job and the job is not currently running a completion routine, the monitor clears the blocking bit so the scheduler can run the job. This action permits completion routines to execute even though the mainline program is blocked.

When all the completion routines finish, the mainline program begins to execute. However, since it was recently blocked, the monitor executes for the job at the start of the $SYSWT routine. $SYSWT runs the relevant decision subroutine (the routine for the condition that originally blocked this job) and reblocks the job, if necessary.

2.5 System Jobs

System jobs are described in detail in the *Introduction to RT–11*. The multi-job mapped monitors are distributed with system job support, and through the system generation process, you can create such support for the FB monitor.

RMON in a system job environment is approximately 300_{10} words larger than an equivalent monitor that does not support system jobs.

2.5.1 Characteristics

System jobs are similar to ordinary foreground jobs in that, for both kinds of jobs, object code must be stored in relocatable object file format. In addition, system jobs are subject to the same restrictions as foreground jobs—that is, they use restricted arithmetic with global variables.

2.5.2 Logical Names

You reference a system job by its logical name, which, by default, is its file name. However, you can assign a new name when you start the job by using the SRUN monitor command with the /NAME:logical-job-name option. Logical job names must be unique. They can be alphanumeric and consist of as little as one letter or digit.

The foreground and background jobs have default logical names as well as their actual file names. For the foreground, the default logical name is *F*; for the background, it is *B*. *F* and *B* are permanently assigned; you cannot use them for system jobs. In addition, *EL* is the logical job name permanently assigned to the error logger system job. You can assign another logical name to the foreground job, in addition to *F* by using the FRUN monitor command with the /NAME:logical-job-name option.

The job name is stored in ASCII at offset I.LNAM in the job's impure area.

2.5.3 Job Number

In a monitor without the system job feature, the background job number is 0 and the foreground job number is 2. In an environment that supports system jobs, the background job number is still 0, but the foreground job number is always 16_8. By default, each system job takes the next highest available job number. Job numbers are multiples of 2, and range from 0 to 16_8. For example, the first system job you start with the SRUN command has a job number of 14, the second system job has a job number of 12, and so on.

2.5.4 Priority

A monitor that supports the system job feature provides the same event-driven, static priority scheduler that ordinary multi-job systems use. The monitor services jobs according to their priority. The background job always has priority 0, the lowest priority. The foreground job always has the highest priority, which is 7. You cannot change these assignments.

To assign a priority to a system job you can:

- Use the SRUN command to start the jobs in order of their importance so that the first job you start gets priority 6, the second job gets priority 5, and so on.
- Explicitly specify the priority when you start the system job. Use the SRUN /LEVEL:priority command to do this. You can specify a priority level for each job in the range 1 through 6, as long as another job is not currently assigned to the level you choose.

The job number is equal to the priority times 2.

You can assign a priority only when you start a system job with the SRUN command. The priority levels do not change dynamically, and you cannot change the priority of a job while it is running.

2.5.5 Design Considerations

If you are planning to write or run system jobs, you should keep in mind two major design considerations:

- RT–11 provides an event-driven, static priority scheduler.
- Address space in low memory is at a premium, and certain parts of each job must reside in low (rather than extended) memory.

2.5.5.1 Scheduling Considerations

The RT–11 scheduler arbitrates the demands jobs make for CPU time, awarding the use of system resources to the highest-priority job that is not blocked. Thus, a compute-bound job that is not blocked can lock out all the jobs with a lower priority. On the other hand, an I/O-bound job, such as the RT–11 QUEUE program, is often blocked waiting for I/O transfers to complete. As a result, it does not interfere significantly with lower priority jobs. If you are running a text editor in the background, for example, the fact that the QUEUE program is active is practically transparent to you.

When you design a program to run as a system job, consider carefully how often it will require system resources. Keep in mind the fact that RT–11 does not permit parallel use of the USR by two or more jobs. Write the program in such a way that it does not monopolize the system and lock out other jobs.

2.5.5.2 Space Considerations

An unmapped environment allows the swapping out of the USR and the sliding down of KMON to extend the amount of low memory available to the programs. That provides about 20K words of memory for the foreground job and device handlers. However, low memory is the only memory available to the system.

In a mapped environment, the USR is always resident. Further, the USR cannot slide down in memory below location 40000 (into the area mapped by kernel PAR1). As a result, about 9.5K-words are available for foreground jobs, device handlers, and system jobs in a mapped environment. This smaller amount of low memory may not be a problem because extended memory is available to the system. Only a job's impure area, queue elements, channels, and (for privileged jobs) the interrupt service routines must reside in low memory. And like the USR, those four parts of

a job cannot reside in the PAR1 area. The rest of the job can reside in extended memory.

The mapped monitors provide four ways to make use of extended memory for foreground and system jobs:

- Run your program in the completely virtual environment, as described in Chapter 3.
- Use the .SETTOP feature in your program.
- Segment your program and use the /V linker option to make the overlays resident in extended memory.
- Use the memory management programmed requests in a MACRO program to increase the program's physical address space.

These methods provide the means to execute code in extended memory. They are described in detail in Chapter 3.

2.5.6 Programmed Requests

Two programmed requests—.GTJB and .CHCOPY—have optional arguments that are meaningful only in a multi-job environment with the system job feature. The .GTJB request obtains job status information for any job in the system. You can reference another job by either logical job name or job number. The .CHCOPY request opens a channel for input, logically connecting it to a file that is currently open for another job for input or output. See the *RT–11 System Macro Library Manual* for a detailed explanation of these requests.

2.5.7 Message Handling

RT–11 provides two methods for sending messages between jobs:

- The .SDAT/.RCVD/.MWAIT programmed requests, through which foreground and background jobs can communicate with each other.

  Those requests are described in the *RT–11 System Macro Library Manual* and their use is illustrated in the foreground/background communications example in the *Introduction to RT–11*.

- The MQ pseudohandler, through which system and foreground jobs communicate. MQ is resident in the monitor and not a file on your distribution kit. (The MQ pseudohandler can be removed from the monitor through the SYSGEN procedure, but its functionality is then no longer available.)

  The MQ message handler is described in this section.

The MQ handler is written as an RT–11 pseudohandler. A pseudohandler is a type of handler that does not connect to any real hardware. For most other purposes, the MQ handler performs like the other RT–11 device handlers, except that it communicates with a job, not a device. Essentially, it makes another job appear to be a peripheral device. As a result, you can open a channel to any other job by using a special .LOOKUP programmed request format, described in the *RT–11 System Macro*

Library Manual. You can send a message by issuing a .WRITx request. Then you can receive a message to the job by using a .READx request. The first word of the received data buffer contains a count of the words transferred.

A further difference between other RT–11 device handlers and the MQ handler becomes apparent when a job exits (with the .EXIT programmed request) or when it aborts (because of CTRL/C or a fatal monitor error). The monitor allows outstanding I/O requests that are queued for the job to complete, but discards any messages that are queued for the job by examining the queue for the MQ handler and removing queue elements that send messages to the job. The mapped monitors normally use a special internal macro to transfer message data via the MTPD instruction. This procedure is slow, but safe, since it does not use a PAR to map any buffers. In some cases, you can use a faster, but more restrictive, transfer procedure by setting the conditional assembly symbol MQH$P2 equal to 1. When the MQ handler is assembled, the assembler will generate code which uses kernel PAR2 to map the user buffers. In this case, all the kernel PAR1 restrictions also apply to PAR2. So, the USR, queue elements, channels, and interrupt service routines cannot reside within locations 20000 through 60000 in a system that actually uses the MQ handler. Note that you cannot run completely virtual jobs when a monitor maps MQ through PAR2 (MQH$P2=1).

2.5.8 Monitor Commands

The collection of monitor commands has some special features that reflect the system job environment. This section describes them briefly. See the *RT–11 Commands Manual* for a complete description. Examples of using the following commands are located in the *Introduction to RT–11*.

2.5.8.1 SRUN and FRUN Commands

Use the SRUN command to start execution of a system job. You can also use the FRUN command to begin execution of a system job in the foreground partition.

NOTE

If you use SRUN or FRUN to start a foreground/system job and a job with the same name is already in memory but has finished executing, the monitor unloads the job in memory and brings in a new copy from a peripheral device.

2.5.8.2 LOAD and UNLOAD Commands

Use the LOAD command to bring a device handler into memory and to assign ownership of a peripheral device to a specific job. Different jobs can own different units of a file-structured device. Since a foreground/system job must already be in memory before you can assign a device to it, remember to start the job with FRUN /SRUN before you use the LOAD command. If the job will not run without the handler, use the /PAUSE option with the FRUN or SRUN command. Note that you cannot assign ownership of SY or MQ.

The UNLOAD command removes a device handler, a foreground/system job, or a global region from memory. You should type a colon (:) after the name of the device handler to distinguish it from the name of a foreground/system job or a global region. If a colon is not included, the UNLOAD command attempts to unload a foreground /system job or a global region of the specified name. If none is found, the command then attempts to unload a device handler with that name. For example, RK could be both the name of a foreground/system job, a global region, and the name of a device handler. To specifically remove the device handler, issue:

```
.UNLOAD RK:
```

To unload the foreground/system job or a global region, issue:

```
.UNLOAD RK
```

2.5.8.3 SUSPEND and RESUME Commands

Use the SUSPEND command to stop execution of a foreground/system job.

Use the RESUME command to continue execution of a foreground/system job that was stopped by the SUSPEND command or the /PAUSE option for SRUN or FRUN.

2.5.8.4 SHOW JOBS Command

Use the SHOW JOBS command to display status information about all foreground /system jobs currently in the system.

2.5.9 Interchanging Between System Jobs

Interchanging between system jobs is described, with examples, in the *Introduction to RT–11*.

2.6 Data Structures

The following sections describe some of the RMON data structures.

2.6.1 Fixed Offsets

Some words always have fixed positions relative to the start of RMON. These words are called fixed offsets. In general, they contain either status words or pointers to other significant information. The fixed offset area in RMON is located at the start of the RTDATA p-sect.

To access the fixed offsets from a running program, use the .GVAL programmed request, as follows:

```
.LIBRARY    "SRC:SYSTEM"
.MCALL      .FIXDF
.GVAL       #area,#offset
```

Here, *area* represents a two-word argument block, and *offset* is a byte offset from Table 2–12.

Table 2–12: RMON Fixed Offsets

Offset	Symbol	Byte Length in Octal	Description
0	$RMON	4	Common interrupt entry point; contains the instruction JMP $INTEN. The .INTEN macro uses it.
4	$CSW	240	Background job channel area (16_{10} channels; each is five words long). See Section 2.3.1 for bit definitions.
244	$SYSCH	12	Internal channel used for system functions; KMON uses this channel.
246-254		2	Reserved.
256	$BLKEY	2	Segment number of the directory now in memory. A value of 0 implies that no directory is there.
260	$CHKEY	2	Device index and unit number of the device whose directory is in memory. The low byte contains the device index into the monitor tables; the high byte is the unit number.
262	$DATE	2	Current date value. See .DATE programmed request in *RT–11 System Macro Library Manual* for the date word format. (Defined by .DATDF macro in SYSTEM.MLB)
264	$DFLG	2	''Directory operation in progress'' flag. This is nonzero to inhibit CTRL/C from aborting a job while a directory operation is in progress.
266	$USRLC	2	Address of the normal USR area. This is where the USR resides when it is called into memory by the background job and $UFLOA (location 46) is 0. In other words, the foreground job must provide space for the USR to swap. (Note: If the foreground job calls in the USR and $UFLOA is 0, the foreground job aborts.) See Chapter 1 for information on USR swapping.
270	$QCOMP	2	Address of the I/O exit routine for all devices. The exit routine is an internal queue management routine through which all device handlers exit once the I/O transfer is complete. Any new device handlers you add to RT–11 must also use this exit location; use the .DRFIN macro in your handler to generate the exit code automatically.
272	SPUSR	2	Special device error word. Non-RT–11 file-structured devices, such as magtape, use this word to report errors to the monitor.
274	$SYUNI	2	The high byte contains the unit number of the system device. This is the unit number of the device from which the system was bootstrapped.

Table 2–12 (Cont.): RMON Fixed Offsets

Offset	Symbol	Byte Length in Octal	Description
276	$SYSVE	1	Monitor version number. You can always access the version number in this fixed offset to determine if you are using the most recent version of the software.
277	$SYSUP	1	Monitor release level. This number identifies the release level of the monitor version specified in byte 276.
300	$CNFG1	2	Configuration word. These 16 bits indicate information about either the hardware configuration of the system or a software condition. Another configuration word located at fixed offset 370 contains additional data. See Section 2.6.1.1 for the meaning of each bit.
302		2	Reserved.(Obsolete; always 0.)
304	$TTKS	2	Address of the console keyboard status register. The default value is 177560. See Chapter 4 for details on changing the hardware console interface to another terminal.
306	$TTKB	2	Address of the console keyboard buffer register. The default value is 177562.
310	$TTPS	2	Address of the console printer status register. The default value is 177564.
312	$TTPB	2	Address of the console printer buffer register. The default value is 177566.
314	$MAXBL	2	The maximum file size allowed in a 0 length .ENTER programmed request. The default value is 177777_8 blocks, allowing an essentially unlimited file size. You can change this value from within a running program (although this is not recommended) or by using SIPP to patch this location.
316	$E16LS	2	Offset from the start of RMON to the dispatch table for EMTs 340 through 357. SL and the BATCH processor uses this. (Defined by .E16DF macro in SYSTEM.MLB)
320	$CNTXT	2	A pointer to the impure area for the current executing job. (Defined by .IMPDF macro in SYSTEM.MLB)
322	$JOBNU	2	The executing job's number.
324	$SYNCH	2	Address of monitor routine to handle .SYNCH requests. Your interrupt routines can issue the .SYNCH programmed request, which enters the monitor through this address to synchronize with the job they are servicing.

Table 2–12 (Cont.): RMON Fixed Offsets

Offset	Symbol	Byte Length in Octal	Description
326	$LOWMA	24	Start of the low-memory protection map. This map protects vectors at locations 0 through 476. See Section 2.6.1.2 for more information on the low-memory bitmap.
352	$USRLO	2	A pointer to the current entry point of the USR. This may be 0, if the USR is not in memory; it may be the relocation code in USRBUF, if the USR was just brought into memory; it is the processing code in all other cases.
354		2	Reserved. (Obsolete; always 0.)
356	$ERRCN	2	Low byte is the error count byte for use by system utility programs. The high byte is reserved.
360	$MTPS	2	Entry point of the move to PS routine. The .MTPS macro calls this routine to perform processor independent moves to the Processor Status word.
362	$MFPS	2	Entry point of the move from PS routine. The .MFPS macro calls this routine to do processor independent moves from the Processor Status word.
364	$SYIND	2	Index into the monitor device tables for the system device. See Section 2.6.3 for information on the device tables.
366	$STATW	2	Indirect file and monitor command state word. See Section 2.6.1.3 for the meaning of each bit.
370	$CNFG2	2	Extension configuration word. This is a string of 16 bits indicating the presence of an additional set of hardware options on the system. See Section 2.6.1.4 for the meaning of each bit.
372	$SYSGE	2	System generation features word. The bits in this word indicate the presence or absence of some system generation special features. See Section 2.6.1.5 for the meaning of each bit.
374	$USRAR	2	Size of the USR in bytes. Your program can use this information to dynamically determine the size of the region you need in order to swap the USR. (The USR is always resident in mapped systems.)
376	$ERRLE	1	Error severity at which to abort indirect command files. You can change this level with the SET ERROR command. The default setting is ERROR. See Section 1.1.2.1 for more information.

Table 2–12 (Cont.): RMON Fixed Offsets

Offset	Symbol	Byte Length in Octal	Description
377	$IFMXN	1	Depth of nesting of indirect files. The default nesting level is 3. You can change this value by using SIPP to patch this location. Be sure to refer to offset 377 as a byte, not as a word.
400	$EMTRT	2	Internal offset for use by SL and BATCH only.
402	$FORK	2	Offset to fork processor from the start of RMON. Use the .DREND macro in your device handler to automatically set up a pointer to the fork processor.
404	$PNPTR	2	Offset to the $PNAME table from the start of RMON.
406	$MONAM	4	Two words of Radix–50 containing the name of the current monitor file.
412	$HSUFF	2	One word of Radix–50 containing the suffix used by the current monitor to name device handlers. For unmapped systems, this word is normally blank. For mapped systems, it is normally *X*, right-justified. This word is set up by the bootstrap; you can modify it there (see the *RT–11 Installation Guide* and the *RT–11 System Generation Guide* for details).
414	$SPSTA	2	Status of the transparent spooler. See Section 2.6.1.6 for the meaning of each bit.
416	$EXTIN	1	IND stored error byte. (Defined by .UEBDF macro in SYSTEM.MLB)
417	$INDST	1	IND control status byte. See Section 2.6.1.7 for bit definitions.
420	$MEMSZ	2	Total physical memory available in 32-word blocks.
422	$ELTIM	2	Reserved for error logger.
424	$TCFIG	2	Address of terminal SET option status word. (Defined by .TTCDF macro in SYSTEM.MLB)
426	$INDDV	2	Pointer to INDDEV (the ASCII name of the device from which IND was run). Also pointer to a 2-byte KMON /UCF interface, located in the first word under INDDEV. See Section 2.6.1.13 for details. This should be considered a read-only location. If you want to change the default device from which IND is run, use the ..INDN offset as described in the *RT–11 Installation Guide*.
430	$MEMPT	2	Offset to memory control block pointers. (Defined by .MEMDF macro in SYSTEM.MLB)

Table 2–12 (Cont.): RMON Fixed Offsets

Offset	Symbol	Byte Length in Octal	Description
432	P1$EXT	2	Pointer to $P1EXT routine, described in *RT–11 Device Handlers Manual*. (Defined by .P1XDF macro in SYSTEM.MLB.)
434-444		2	Reserved.
446	$IMPLO	2	Pointer to word following last word in table of pointers to the impure area. See Section 2.6.1.8 for more information.
450	$KMONI	2	Indicates if KMON is the background job. If $KMONI contains nonzero value, then KMON is background job; if $KMONI contains zero, then KMON not background job.
452	$PROGD	1	Indicates default editor for EDIT command. The bit values in $PROGD are defined in Section 2.6.1.9.
453	$PROGF	1	Indicates which FORTRAN compiler is run by the COMPILE/FORTRAN, EXECUTE/FORTRAN, and FORTRAN commands. See Section 2.6.1.10 for bit definitions.
454	$WILDD	1	Indicates if wildcarding (*) is implicit or must be supplied explicitly. A value of one indicates implicit wildcarding; zero indicates explicit wildcarding.
455	$JOBS	1	Indicates the number of job slots available on the system. See Section 2.6.1.11 for bit definitions.
456	$QHOOK	2	Pointer to RMON code hooks for UB system support handler. (Defined by .QHKDF macro in SYSTEM.MLB)
460	$H2UB	2	Pointer to the UB system support handler entry vector.
462	$XOFF	2	Pointer/offset to an XON/XOFF flag. In nonmultiterminal monitors, $XOFF points to a byte that contains the XON/XOFF flag. In multiterminal monitors, $XOFF contains the offset in the TCB to the T.STAT word. In either case, the high bit (bit 7) of the byte contains the flag. If bit 7 is set, the terminal has sent XOFF to the monitor; if clear, XON is in effect.
464	$RTSPC	4	Address of SEC/RETURN instructions.
466	$CNFG3	2	Third configuration word. See Section 2.6.1.12 for bit definitions.
470	$XTTNR	2	Pointer to a routine to be called when there is no room for input characters in the terminal input buffer.

Table 2–12 (Cont.): RMON Fixed Offsets

Offset	Symbol	Byte Length in Octal	Description
472	$THKPT	2	Pointer to the optional multiterminal handler hooks data structure, THOOKS. See Section 4.4.1.1 for the THOOKS data structure.
474		2	Reserved. (Obsolete; always 0.)
476	$XTTPS	2	Terminal service calls @$XTTPS before reading @$TTPS and after changing @$TTPS.
500	$XTTPB	2	Terminal service calls @$TTPB after changing @$TTPB.
502	$SLOT2	1	The value of $SLOT*2.
503		1	Reserved.
504	$SPSIZ	2	Special device file size. Used to return value from a special directory handler to the monitor. The monitor returns this value in R0 for special directory operations.

2.6.1.1 Configuration Word ($CNFG1)

The configuration word, $CNFG1, gives information about either the hardware configuration of the system or a software condition. Table 2–13 lists the bits and their meanings. Unused bits are reserved for future use by Digital.

Table 2–13: Configuration Word (.CFIDF), Offset 300

Bit	Symbol	Meaning
0	FBMON$	0 = SET MODE SJ in effect. 1 = SET MODE NOSJ in effect.
1	SLKMO$	1 = KMON uses SL editor.
2		Reserved. (Obsolete; always 0.)
3	BATCH$	1 = BATCH is in control of the background.
4	SLEDI$	1 = Single-line editor is available to user programs.
5	CLK50$	0 = 60-cycle clock 1 = 50-cycle clock The value can be controlled by SET CLOCK 50/60.
6	HWFPU$	1 = FP11 floating-point hardware exists.
7	FJOB$	0 = No foreground or system job is in memory. 1 = A foreground or system job is in memory.
8		Reserved. (Obsolete; always 0.)

Table 2–13 (Cont.): Configuration Word (.CFIDF), Offset 300

Bit	Symbol	Meaning
9	USR$	1 = USR is permanently resident, by issuing SET USR NOSWAP. (USR is always resident in mapped systems and this bit is always set.)
10	QUEUE$	1 = The QUEUE program is running.
11	LSI11$	1 = The Processor Status word on this system cannot be accessed by means of an address in the I/O page.
12	KT11$	1 = A mapped system is running.
13	LKCS$	1 = The system clock has a status register.
14	KW11P$	1 = A KW11–P clock exists and programs can use it.
15	CLOCK$	1 = There is a system clock (L clock, P clock, or 11/03–11/23 line-frequency clock).

2.6.1.2 Low-Memory Protection Bitmap ($LOWMA)

RT–11 maintains a bitmap that reflects the protection status of low memory, locations 0 through 477. This map is required in order to avoid conflicts in the use of the vectors. In multiuser systems, the .PROTECT programmed request allows a program to gain exclusive control of a vector or a set of vectors. When a vector is protected, RMON updates the bitmap to indicate which words are protected. If a word in low memory is not protected, it is loaded from block 0 of the executable file. If a word in low memory is protected, it is not loaded from block 0 of the file. In addition, if the word is protected by a foreground job, it is not destroyed when you run a new background program.

The bitmap is a 20_{10}-byte table that starts at offset $LOWMA, 326_8 bytes from the beginning of RMON. Table 2–14 lists the offset from RMON and the corresponding locations represented by that byte.

Table 2–14: Low-Memory Bitmap

Offset	Locations	Offset	Locations
326	0–17	340	240–257
327	20–37	341	260–277
330	40–57	342	300–317
331	60–77	343	320–337
332	100–117	344	340–357
333	120–137	345	360–377
334	140–157	346	400–417
335	160–177	347	420–437

Table 2–14 (Cont.): Low-Memory Bitmap

Offset	Locations	Offset	Locations
336	200–217	350	440–457
337	220–237	351	460–477

Each byte in the table reflects the status of eight words of memory. The first byte in the table controls locations 0 through 17, the second byte controls locations 20 through 37, and so on. The bytes are read from left to right. Thus, if locations 0 through 3 are protected, the first byte of the table contains 11000000.

NOTE

Only words are protected, not individual bytes. Thus, protecting word 0 means that bytes 0 and 1 are both protected.

If locations 24 through 27 are protected, the second byte of the table contains 00110000.

The leftmost bit of each byte represents lower memory locations; the rightmost bit represents higher memory locations. For example, to protect locations 300 through 307, the leftmost four bits of the byte at offset 342 must be set to result in a value of 360 for that byte: 11110000.

The single-job monitors do not support the .PROTECT programmed request. If you need to protect vectors in a single-job system, either use SIPP to manually modify the bitmap or dynamically modify the bitmap from within a running program.

For example, the following instructions protect locations 300 through 306 dynamically:

```
MOV       @#$SYPTR,R0
  BISB    #^B11110000,$LOWMA+14(R0)
```

The RT–11 monitor uses the low-memory bitmap to automatically protect some locations in low memory. The locations it protects are as follows:

0–16
24–32
50–66
100–102 (line-frequency clock)
104–106 (if KW11-P selected as system clock)
114–116
244–246
250–252 (for mapped systems only)
The system device handler interrupt vector
Interrupt vectors for loaded device handlers
Vectors for all interfaces supported in a multiterminal system

NOTE

Vectors of device handlers that you load with the LOAD command are protected; vectors of device handlers that you bring into memory with the .FETCH programmed request are not protected.

2.6.1.3 DCL and IND Indirect File Status Word ($STATW)

The DCL and IND indirect file status word, $STATW, returns information about the DCL (indirect command) and IND (indirect control) files. Table 2–15 lists the bits and their meanings. Unused bits are reserved for future use by Digital.

Table 2–15: DCL and IND File Status Word (.STWDF), Offset 366

Bit Mask	Symbol	Meaning
000001- 000002		Reserved.
000004	IFIND$	0 = @ means indirect command file. 1 = @ means indirect control file.
000010	IFDOL$	Dollar sign ($) entered at command line.
000020	IFSPC$	Special chain exit.
000040	IFBEX$	BATCH is forcing exit after error in user job.
000100	IFRVT$	Revert to TTY input from indirect file.
000200	IFGTC$	CTRL/C seen in indirect file while GTLIN$ bit set in $JSW.
000400	IFACT$	Indirect file active.
001000	IFCHA$	Chain to indirect file.
002000	IFEKO$	Don't echo indirect file lines.
004000	IFCTC$	CTRL/C seen in indirect file.
010000	IFDAT$	Data in indirect file buffer above the USR.
020000	IFEOF$	EOF in indirect command file.
040000	IFABT$	Abort indirect command file input.
100000	IFINP$	Input from indirect command file.

2.6.1.4 Second Configuration Word ($CNFG2)

The second configuration word, $CNFG2, indicates the presence of an additional set of hardware options on the system. Table 2–16 lists the bits and their meanings. Unused bits are reserved for future use by Digital.

Table 2–16: Extension Configuration Word (.CF2DF), Offset 370

Bit	Symbol	Meaning
0	CACHE$	1 = Cache memory is present.
1	MPTY$	1 = Parity/ECC memory is present.
2	SWREG$	1 = A readable switch register is present.
3	LIGHT$	1 = A writeable console display register is present.
4	LDREL$	1 = A handler used by LD may have been unloaded.
5	XITSW$	1 = Do not swap user code or exit.
6	BUS$	See below and Table 2–17.
7	CIS$	1 = The Commercial Instruction Set (CIS) option is present.
8	EIS$	1 = The Extended Instruction Set (EIS) option is present.
9-10		Reserved. (Obsolete; always 0.)
11	KXCPU$	See below and Table 2–17.
12	GSCCA$	1 = Global SCCA support in monitor.
13	PROS$	See below and Table 2–17.
14	PDP70$	1 = The processor is a PDP–11/70.
15	PDP60$	1 = The processor is a PDP–11/60.

Determining Processor Bus Structure

Three bit masks in $CNFG2 indicate the type of processor on which RT–11 is running: BUS$, PROS$, and KXCPU$. Programs that previously determined if RT–11 is running on a CTI Bus machine by testing only PROS$, along with other bus determinations, should use the following algorithm. The algorithm establishes the correct bit mask and then tests values within that mask for relevant $CNFG2 values.

Given the following symbol values:

```
BUS$M = 020100  mask for bus bits
BUS$U = 000000  value for UNIBUS
BUS$Q = 000100  value for Q-bus
BUS$C = 020000  value for CTI Bus
BUS$X = 020100  value for other bus or busless system
```

Use the following code fragment:

```
.GVAL   #AREA,#$CNFG2       ; Put value of $CNFG2 in R0
BIC     #^CBUS$M,R0         ; Relevant mask
CMP     #BUS$U,R0           ; Look for UNIBUS
BEQ     UNIBUS              ; If equal, then UNIBUS
CMP     #BUS$Q,R0           ; Look for Q-bus
BEQ     Q-bus               ; If equal, then Q-bus
CMP     #BUS$C,R0           ; Look for CTI Bus
BEQ     CTI Bus             ; If equal, then CTI Bus
CMP     #BUS$X,R0           ; Look for other bus or
                            ;  busless
BEQ     UNKBUS              ; If equal, then KXJ-11 if
                            ;  KXCPU$ is set
                            ; Done
```

Table 2–17: Determining Processor Bus Structure

Symbol	Bit Mask		Meaning
BUS$	000100	0 =	RT–11 is running on UNIBUS or CTI Bus machine.
			If PROS$ = 0, then UNIBUS machine
			If PROS$ = 1, then CTI Bus machine
		1 =	RT–11 is running on Q-bus machine, a machine with an unknown bus (not CTI Bus, Q-bus, or UNIBUS), or a KXJ–11 processor.
			If PROS$ = 0, then Q-bus machine
			If PROS$ = 1, then unknown bus machine; check KXCPU$
KXCPU$	004000	0 =	RT–11 is not running on a KXJ-11 processor; check BUS$ and PROS$.
		1 =	RT–11 is running on a KXJ-11 processor.
PROS$	020000	0 =	RT–11 is running on a Q-bus or UNIBUS machine; check BUS$.
		1 =	RT–11 is running on a CTI Bus, unknown bus, or KXJ–11 bus machine; check BUS$ first, then check KXCPU$.

2.6.1.5 System Generation Features Word ($SYSGE)

The system generation features word, SYSGEN, indicates which major system generation features are present. Table 2–18 lists the meaning of each bit. Unused bits are reserved for future use by Digital. In addition, do not set or clear any bits in this word yourself.

Note that the values of the low byte must correspond to the conditional variables you use when you assemble your device handler files. Attempts to use handlers that are not compatible with the monitor cause the *?KMON-F-Conflicting SYSGEN options* error message to appear.

Table 2–18: System Generation Features Word (.SGNDF), Offset 372

Bit	Symbol	Meaning
0	ERLG$	1 = The error logging feature is present.
1	MMGT$	1 = The memory management feature is present.
2	TIMIT$	1 = The device I/O time-out feature is present.
3	RTEM$	1 = This is an RTEM–11 system.
4-7		Reserved.
8	FPU11$	1 = FPU support selected during system generation.
9	MPTYS$	1 = The memory parity feature is present.
10	TIMER$	1 = The SB mark time feature is present.
11-12		Reserved.
13	MTTY$	1 = The multiterminal feature is present.
14	STASK$	1 = The system job feature is present.
15	TSXP$	Running under an operating system other than RT–11.

2.6.1.6 Transparent Spooler (SPOOL) Status Word ($SPSTA)

The transparent spooler status word, SPSTAT, indicates the status of the transparent spooler (SPOOL). Table 2–19 indicates the meaning of each bit. Unused bits are reserved for future use by Digital.

Table 2–19: Transparent Spooler Status Word (.SPLDF), Offset 414

Bit	Symbol	Meaning
0-2	SP$UNI†	Unit number for SET commands. The meaning of the unit number bits is tied to SP$OPR as shown below.
3-6	SP$OPR	Type of operation to perform. See Table 2–20.
7	SP$ACT	Spooler active.
8-10	SP$NFL	Type of flag page support. See Table 2–21.
11	SP$SHO	1 = Display spooler status.
12	SP$SCN	1 = Print screen (CTI bus-based computers only).

†The meaning of the spooler unit field (SP$UNI) is determined by the bit mask in Table 2–20:

- If SP$OPR is SP$NXT, SP$OFF, SP$ON, or SP$KIL, then SP$UNI is the SP unit number.
- If SP$OPR is SP$ABT, then SP$UNI is the aborting job number.
- If SP$OPR is SP$EXI, then SP$UNI has no meaning.

Table 2–19 (Cont.): Transparent Spooler Status Word (.SPLDF), Offset 414

Bit	Symbol	Meaning
13	SP$GTM	1 = Date and time request for flag pages.
14	SP$IEN	1 = Fake interrupt enable.
15	SP$ERR	1 = Error bit (set by SPOOL).

Table 2–20: Spool Operation Type Field (SP$OPR)

Bit Mask	Symbol	Meaning
0001	SP$NXT	Move to start of next file.
0010	SP$OFF	Set spooler unit off.
0100	SP$ON	Set spooler unit on.
1000	SP$KIL	Remove spooled output from work file.
0110	SP$ABT	Spooler aborted.
1110	SP$EXI	Cause spooler shutdown.

Table 2–21: Spool Flag Page Support Field (SP$NFL)

Bit Mask	Symbol	Meaning
000	SP$DEF	Default number of flagpages.
111	SP$0	No flagpages.
xxx		This number of flagpages, where *xxx* is in the range 1-6.

2.6.1.7 IND Control Status Byte ($INDST)

The following bits are defined in the IND control status byte:

Table 2–22: IND Control Status Byte (.INDDF), Offset 417

Value	Symbol	Meaning
001		Reserved.
002		Reserved.
004	CC$IND	Set if double CTRL/C abort is disabled by IND.
010	CC$GBL	Set if double CTRL/C is disabled by global .SCCA.
040	LN$IND	Set if current line passed by IND.
100	IN$RUN	Set if KMON issued RUN of IND.

Table 2–22 (Cont.): IND Control Status Byte (.INDDF), Offset 417

Value	Symbol	Meaning
200	IN$IND	Set if IND active.

2.6.1.8 Pointer to the End of the Impure Area ($IMPLO)

Subtract 2 from the contents of $IMPLO–2 to return a pointer to the foreground job impure area if a foreground job is loaded, or a 0 if no foreground job is loaded.

The lowest word in the table of pointers to impure areas contains –1. The word above the word containing –1 is the pointer to the background job impure area. The pointers to the impure area for any system jobs supported by the monitor are located between the pointers to the foreground and background impure areas.

2.6.1.9 Default Editor ($PROGD)

The following bits are defined in $PROGD. The high byte is undefined. All unused bits are reserved by Digital.

Table 2–23: Default Editor (.PGMDF), Offset 452

Value	Symbol	Meaning
022	$$KED	SET EDIT KED is effect.
023	$$K52	SET EDIT K52 is effect.
024	$$KEX	SET EDIT KEX is effect.
026	$$EDIT	SET EDIT EDIT is effect.
027	$$TECO	SET EDIT TECO is effect.

2.6.1.10 Default FORTRAN Compiler ($PROGF)

The following bits are defined in $PROGF. The high byte is undefined. All unused bits are reserved by Digital.

Table 2–24: Default FORTRAN Compiler, Offset 453

Value	Symbol	Meaning
000006	$$FORT	SET FORTRA F4 is in effect. The default.
000030	$$F77	SET FORTRA F77 is in effect.

2.6.1.11 Job Slots on the System ($JOBS)

The values in $JOBS for the distributed monitors are as follows. All unused bits are reserved by Digital.

Table 2–25: Job Slots on the System, Offset 455

Value	Meaning
001	1 job slot (distributed single-job monitor).
002	2 job slots (distributed FB monitor).
010	8 job slots (distributed multi-job mapped monitors).

2.6.1.12 Third Configuration Word ($CNFG3)

The following bits are defined in the third configuration word. All unused bits are reserved by Digital.

Table 2–26: Third Configuration Word (.CF3DF), Offset 466

Value	Symbol	Meaning
000004	CF3.IM	Each job possesses an impure area.
000010	CF3.VB	RUN is SET to VBGEXE.
000020	CF3.UI	The UB system support handler is set to not install.
000040	CF3.UA	The UB system support handler is active (set only when CF3.UB is set).
000100	CF3.UB	The UB system support handler is loaded in memory.
000200	CF3.DM	At least one handler installed on the system uses DMA (direct memory access).
000400	CF3.64	RMON has been generated for extended device-unit support and the $PNAM2 table exists in RMON.
001000	CF3.AT	$JBREL exists in monitor.
002000	CF3.OW	$OWNER table exists in RMON.
004000	CF3.US	Support for UB system support handler in the PNAME table from BSTRAP.
010000	CF3.1S	RMON supports exactly one DL11 console interface.
020000	CF3.AS	Multiterminal monitor supports asynchronous terminal status (AST) word.
040000	CF3.HI	The hardware supports separated I & D address space and Supervisor Mode.
100000	CF3.SI	The monitor supports separated I & D address space and Supervisor Mode.

2.6.1.13 KMON/UCF Interface Word (CLI.FL/CLI.TY)

The KMON/UCF interface is a single word consisting of the following bytes: CLI.FL (the low byte) and CLI.TY (the high byte). This word is located immediately below INDDEV, the word pointed to by offset $INDDV. All unused bits and values are reserved for Digital.

Table 2–27: CLI.FL (.CLIDF)

Bit	Symbol	Meaning
7	UCF.KM	Set by UCF to pass a command to KMON. Cleared by KMON.
4-6		Reserved.
3	UCL.ON	1 = Do UCL parsing.
2	CCL.ON	1 = Do CCL parsing.
1	DCL.ON	1 = Do DCL parsing.
0	UCF.ON	1 = Do UCF parsing. Set by UCF.

Table 2–28: CLI.TY (.CLIDF)

Value	Symbol	Meaning
0	UCF.RN	The program is running as UCF.
1	DCL.RN	The program is running from DCL.
2	CCL.RN	The program is running from CCL.
3	UCL.RN	The program is running as UCL.
4-255		Reserved.

2.6.2 Impure Area

The impure area is an area of memory where the monitor stores all job-dependent data. For each job, the impure area contains job-specific information, such as terminal ring buffers and I/O channels. The monitor sets up the impure area and maintains its contents.

The impure areas contain all the information the monitor requires to run a job. The information stored in the impure area is job-specific. The impure area for the background job is located in the p-sect RMON in the Resident Monitor and it is permanently resident. In a multi-job system, the impure area for a foreground or system job is located in memory below the start of the job itself. The size of the impure area is the value in the global symbol FMPUR, which you can find by looking at your monitor's link map.

The monitor maintains a table of one-word pointers to the impure areas of all jobs in the system. This table is located at $IMPUR and is either eight, two, or one word(s)

long, depending on whether the monitor is single-job or multi-job, and whether system job feature is present or not.

In RT–11, a background job is always present. It is KMON if no other background job exists. In a multi-job system, the foreground or system job impure area pointer may be 0 if no such job is in memory. When you issue an FRUN command, the monitor creates an impure area for the foreground job. Similarly, the SRUN command creates an impure area for a system job. In both cases, the monitor also updates the job's $IMPUR entry to point to the impure area.

The contents of the impure area are the same for the background and the foreground jobs, as shown in Table 2–29. The offset in the table is the offset from the start of the impure area itself. Beginning at I.JID, the contents of the impure area depend on which system generation features you select.

Table 2–29: Impure Area (.IMPDF)

Offset	Symbol	Byte Length	Description
0	I.STATE	2	Job state word bits. See Table 2–30 for the meaning of each bit.
2	I.QHDR	2	Head of available queue element linked list.
4	I.CMPE	2	Last entry in the completion queue.
6	I.CMPL	2	Head of the completion queue.
10	I.CHWT	2	Pointer to channel during I/O wait. When a job is waiting for I/O on a channel to complete, the address of that channel area is stored here.
12	I.PCHW	2	Saved I.CHWT during execution of a completion routine.
14	I.PERR	2	Error bytes 52 and 53 saved during completion routines.
16	I.TTLC	2	Terminal input ring buffer line count (for non-multiterminal systems).
16	I.CNSL	2	Multiterminals only: Pointer to terminal control block (TCB) for this job's console terminal.
20	I.PTTI	2	Previous terminal input character (for non-multiterminal systems).
20	unused	2	Multiterminals only: reserved.
22	I.TID	2	Pointer to job ID area, later in impure area.
24	I.JNUM	2	Job number of the job that owns this impure area.
26	I.CNUM	2	Number of I/O channels defined. The default is 16_{10}; you can use .CDFN to define more.
30	I.CSW	2	Pointer to the job's channel area.
32	I.IOCT	2	Total number of I/O operations outstanding.

Table 2–29 (Cont.): Impure Area (.IMPDF)

Offset	Symbol	Byte Length	Description
34	I.SCTR	2	Suspension counter. A value less than 0 means the job is suspended.
36	I.BLOK	2	Job blocking bits. See Table 2–31 for the meaning of each bit.
The following offsets are not guaranteed to remain constant from release to release. In fact, since the pointers and status words can vary depending on the special features you select through system generation, you should consult the link map from the monitor assembly to find the correct offsets for your system. Some items, such as the input and output ring buffers, have a variable length.			
-	I.JID	10	Job's terminal prompt string. If the system job feature is present, the length of I.JID is 14_8.
-	I.LNAM	6	System jobs only: Logical job name in ASCII.
-	I.NAME	10	File name and file type, in Radix–50, of the running job.
-	I.SPLS	2	Pointer to nonlinked .DEVICE list.
-	I.TRAP	2	Address of trap to 4 and 10 routine defined via .TRPSET.
-	I.FPP	2	FPU only: Address of FPP exception routine defined via .SFPA.
-	I.SPSV	2	Mapped only: Bottom of saved SP data.
-	I.SWAP	4	Multi-job only: Pointer to extra swap information specified in the .CNTXSW programmed request.
-	I.SP	2	Multi-job only: Saved stack pointer.
-	I.BITM	24	Multi-job only: Bitmap for protection.
-	I.CLUN	2	Multiterminals only: LUN of job's console.
-	I.TTLC	2	Multiterminals only: Terminal input ring buffer line count.
-	I.IRNG	2	Input ring buffer low limit.
-	I.IPUT	2	Input PUT pointer for interrupts.
-	I.ICTR	2	Input character count.
-	I.IGET	2	Input GET pointer for .TTYIN.
-	I.ITOP	2	Input ring buffer high limit.
-	——	TTYIN	Input ring buffer.
-	I.OPUT	2	Output PUT pointer for .TTYOUT.
-	I.OCTR	1	Output character count.
-	I.CTLO	1	^C flag

Table 2–29 (Cont.): Impure Area (.IMPDF)

Offset	Symbol	Byte Length	Description
-	I.OGET	2	Output GET pointer for interrupts.
-	I.OTOP	2	Output ring buffer high limit.
-	——	TTYOUT	Output ring buffer.
-	I.QUE	QWDSIZ	The initial queue element; 16_8 bytes (24 bytes if mapped).
-	——	4	Multi-job only: First internal message channel words.
-	I.SERR	2	(All monitors) The third word of the message channel is used as the hard/soft error flag.
-	I.MSG	4	Multi-job only: Further internal message channel words.
-	I.TERM	2	Terminal status word. (.TSTDF)
-	I.TRM2	2	Terminal status word 2. (.TS2DF)
-	I.SCCA	2	CTRL/C terminal status word set via .SCCA. (.TASDF)
-	I.SCC1	2	Mapped only: PAR1 value of I.SCCA.
-	I.DEVL	2	Pointer to linked .DEVICE list.
-	I.FPSA	2	Mapped and FPU only: Pointer to FPU save area, later in impure area.
-	I.SCOM	36	Mapped only: System communication save area (for non-multiterminal systems).
-	I.SCOM	40	Mapped and multiterminals only: System communication save area.
-	I.RSAV	20	Mapped only: Register save area.
-	I.MPTR	2	Mapped only; Pointer to job's mapping context area (MCA) in extended memory. The PAR1 value. See Table 2–32 for contents of the mapping context area.
-	I.MPB	16	Mapped only; Temporary copy of the job's parameter block for a PLAS request; contains an RDB or WDB, depending on type of request.
-	I.PLSP	2	Fully mapped (ZB or ZM) only; D-space / I-space 'pass' parameter for PLAS requests.
-	I.CMAP	2	Fully mapped (ZB or ZM) only; .CMAP request status. See Table 2–32 for bit definitions.
-	I.SSP	2	Fully mapped (ZB or ZM) only; saved Supervisor Mode stack pointer.
-	I.FSAV	62	Mapped and FPU only: FPU save area.

Table 2–29 (Cont.): Impure Area (.IMPDF)

Offset	Symbol	Byte Length	Description
-	I.VHI	2	Mapped only: Virtual high limit of job; nonzero if linker /V option used.
-	I.VSTP	2	Mapped only; used for nonvirtual .SETTOP; maximum high limit for completely virtual jobs.
-	I.ECTR	2	.SPCPS only: EMT depth counter.
-	I.SPCP	2	.SPCPS only: Address of .SPCPS blocks.
-	I.SPC1	2	Mapped and .SPCPS only: PAR1 for .SPCPS blocks.
-	I.SCHP	2	Pointer to the job's system channel. The monitor uses this channel for its own calls, such as .DSTATUS.
-	I.SYCH	14	The job's system channel, for all foreground and system jobs. The background job's channel is in the fixed offset area of RMON.
-	IMP.SZ	0	Symbol whose value is the length in bytes of impure area.

Job State Word Bits

The job state word, I.STATE, indicates status information about a job. Table 2–30 shows the meaning of each bit. Unused bits are reserved for future use by Digital.

Table 2–30: Job State Word Bits, Offset 0 (.ISTDF)

Mnemonic	Bit	Meaning When Set
ABPND$	0	An abort has been requested for this job.
BATRN$	1	BATCH or Error Logger is running for this job.
CSIRN$	2	The CSI is running for this job.
USRRN$	3	The USR is running for this job.
-	4	Reserved.
ABORT$	5	The job is being aborted.
-	6	Reserved.
CPEND$	7	This job has a completion routine pending.
-	8-10	Reserved.
LOAD$	11	Mapped only; foreground or system job is in the load phase; the job has not yet started running. Context switch code in RMON will initialize mapping and clear LOAD$ bit.
WINDW$	12	This is a virtual job.

Table 2–30 (Cont.): Job State Word Bits, Offset 0 (.ISTDF)

Mnemonic	Bit	Meaning When Set
VRUN$	13	This job is running under VBGEXE.
VLOAD$	14	This job is being loaded under VBGEXE.
CMPLT$	15	A completion routine is running for this job.

Job Blocking Bits

The job blocking word, I.BLOK, indicates which condition is blocking a job. Unused bits are reserved for future use by Digital. Table 2–31 shows the meaning of each bit.

Table 2–31: Job Blocking Bits, Offset 36 (.IBKDF)

Mnemonic	Bit	Meaning When Set
-	0-3	Reserved.
USRWT$	4	The job is waiting for the USR.
-	5	Reserved.
KSPND$	6	The job is suspended as a result of the SUSPEND command.
-	7	Reserved.
EXIT$	8	The job is waiting for all I/O to complete.
NORUN$	9	The job is not running (that is, it is a foreground or system job).
SPND$	10	The job is suspended.
CHNWT$	11	The job is waiting for I/O on a channel to complete.
TTOEM$	12	The job is waiting for the output ring buffer to be empty.
TTOWT$	13	The job is waiting for room in the output ring buffer.
TTIWT$	14	The job is waiting for terminal input.
-	15	Reserved.

Mapping Context (I.CMAP) Word Bits

The I.CMAP word is written by the .CMAP programmed request and shows the status of a job's mapping context. The job's mapping context is used by the system to determine which WCBs, RCBs, PARs and PDRs, are supported for which processor modes. That information is used when context switching those structures in and out of the processor's memory management unit.

Context switching is discussed in detail in Chapter 3.

The word is divided into 4 fields. Each field tracks a particular aspect of a job's mapping context.

The following bits are defined; any undefined bits are reserved by Digital.

Table 2–32: Change Mapping Context (I.CMAP) Word Bits (.CMPDF)

Bit Mask	Symbol	Meaning
000001	CM.PR0	Separate PAR0 mapping.
000002	CM.PR1	Separate PAR1 mapping.
000004	CM.PR2	Separate PAR2 mapping.
000010	CM.PR3	Separate PAR3 mapping.
000020	CM.PR4	Separate PAR4 mapping.
000040	CM.PR5	Separate PAR5 mapping.
000100	CM.PR6	Separate PAR6 mapping.
000200	CM.PR7	Separate PAR7 mapping.
001400	CM.S	Supervisor mode I & D Separation Field.
001000	CM.SII	Nonseparate Supervisor I & D space.
001400	CM.SID	Separate Supervisor I & D space.
002000		Reserved.
030000	CM.SUP	Supervisor mode context switching support field.
020000	CM.NOS	No Supervisor mode context switching.
030000	CM.JAS	Supervisor mode context switching.
140000	CM.U	User mode I & D separation field.
100000	CM.UII	Nonseparate User I & D space.
140000	CM.UID	Separate User I & D space.

2.6.3 Device Tables

The following tables in RMON keep track of the devices on the RT–11 system. The size of each table is determined by the number of device slots ($SLOT) built into the system. The value for $SLOT*2, used to calculate the size of each table, is located at $SLOT2 (RMON fixed offset 502). The relative location of each device handler in each table is the same, except for tables $UNAM1 and $UNAM2, as explained in the following sections. Therefore, pertinent information for a particular handler can be returned by sequentially indexing into each table (at the same offset from the start of the table).

Table	Size	Contents
$OWNER:	<$SLOT*2>*2	Device ownership; omitted in single-job monitors and can be removed from multi-job monitors through system generation.
$UNAM1:	<$SLOT*2>+4	Physical device name.
$UNAM2:	<$SLOT*2>+4	Logical device name.
$PNAME:	$SLOT*2	Installed handlers.
$ENTRY:	<$SLOT*2>+2	Handler addresses. Last word contains -1 and indicates end of table.
$STAT:	$SLOT*2	DSTATUS value.
$DVREC:	$SLOT*2	Handler disk blocks.
$HSIZE:	$SLOT*2	Handler memory size.
$DVSIZ:	$SLOT*2	Handler device blocks.
$PNAM2	<$SLOT*2>+2	Physical device name for extended-unit (single-letter) device names. Last word contains default device name, if assigned.

Typically, you would use the table in the following manner:

1. Find the value located in fixed offset $SLOT2.
2. Do a .CSTAT and use the mask to isolate the index, INDX$M, for the handler in question. Use this value to index into the $PNAME table for that handler.
3. Index into the various tables, using that value and $SLOT2.

2.6.3.1 $PNAME Table

The permanent name table is called $PNAME. It is the central table around which all the others are constructed. The total number of entries is fixed at assembly time; you can allocate extra slots then. Entries are made in $PNAME at monitor assembly time for each device that is built into the system.

Each table entry consists of a single word that contains the Radix–50 code for the two-character physical device name. (For example, the entry for TS11 is .RAD50 /MS/.) The TT device must be first in the table; the system device is always second. After that, the position of a device in this table is not critical. Once the entries are made into this table, their relative position (that is, their order in the table) determines the general device index used in various places in the monitor. Thus, the other tables (other than $UNAM1 and $UNAM2) are organized in the same order as $PNAME. The offset of a device name entry in $PNAME serves as the index into the other tables for a given device.

The bootstrap checks the system generation parameters of a handler with those of the current monitor (by inspecting the low byte of $SYSGE at RMON fixed offset 372), and zeroes the $PNAME entry for that device if the parameters do not match.

The INSTALL command cannot install a handler whose conditional parameters do not match those of the monitor.

2.6.3.2 $STAT Table

The device status table is called $STAT. Entries to this table are made at assembly time for those devices that are permanently resident in the RT–11 system, such as TT. When the system is bootstrapped, the entries for all other devices are filled in when each handler is installed by the bootstrap or the INSTALL command. Each device in the system has a status entry in its corresponding slot in $STAT. The device status word identifies each physical device and provides information about it, such as whether it is random or sequential access. The device status word is part of the information returned to a running program by the .DSTATUS programmed request. See *RT–11 Device Handlers Manual* for details on the status word.

2.6.3.3 $DVREC Table

The device handler block number table is called $DVREC. Entries to this table are made at bootstrap time for devices that are built into the system, and at INSTALL time for additional devices. The entries are the absolute block numbers where each of the device handlers resides on the system device. Since handlers are treated as files, their positions on the system device are not necessarily fixed. Thus, each time the system is bootstrapped, the handlers are located and $DVREC is updated with their locations on the system device. The pointer in $DVREC points to block 1 of the file. (Because handlers are linked at 1000, the actual handler code starts in the second block of the file.) A zero entry in the $DVREC table indicates that no handler file for the device in that slot was necessary (such as TT). (Note that if block 0 of the handler file resides on a bad block on the system device, RT–11 cannot install or fetch the handler.) Note also that 0 is a valid $DVREC entry for permanently resident devices.

2.6.3.4 $ENTRY Table

The handler entry point table is called $ENTRY. Entries in this table are made whenever a handler is loaded into memory by either the .FETCH programmed request or by the LOAD command. The entry for each device is a pointer to the fourth word of the device handler in memory. The entry is zeroed when the handler is removed by the .RELEASE programmed request or by the UNLOAD command.

Some device handlers are permanently resident. These include the system device handler and the TT handler. The $ENTRY values for such devices are fixed at boot time.

2.6.3.5 $DVSIZ Table

Each entry in the $DVSIZ table contains the size of a device in blocks. The value is 0 for a non-file-structured device. For devices that accept multisize volumes, the entry contains the size of the smallest possible volume.

2.6.3.6 $HSIZE Table

Each entry in the $HSIZE table contains the size of a device handler in bytes. This value indicates the amount of memory needed to load each handler.

2.6.3.7 $UNAM1 And $UNAM2 Tables

The tables that keep track of logical device names and the physical names that are assigned to them are called $UNAM1 and $UNAM2. Entries are made in these tables when the ASSIGN command is issued. The physical device name is stored in $UNAM1 and the logical name associated with it is stored in the corresponding slot in $UNAM2. When the system is first bootstrapped, there are two assignments already in effect that associate the logical names DK and SY with the device from which the system was booted.

The value of $SLOT, which is determined at system generation time, limits the total number of logical name assignments. Thus, you can issue one ASSIGN command for each device slot in your system. (The initial SY and DK assignments at bootstrap time do not come out of your total.)

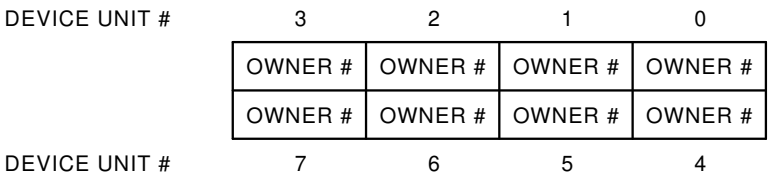
The $UNAM1 and $UNAM2 tables are not indexed by the $PNAME table offset.

2.6.3.8 $OWNER Table

The device ownership table is called $OWNER and it is used in the multi-job environments to arbitrate device ownership. There is no $OWNER table in the single-job systems.

The table is divided into two-word entries for each device. Entries are made into this table when the LOAD command is issued. Each two-word entry is in turn divided into eight four-bit fields capable of holding a job number. The low four bits of the first byte correspond to unit 0, and the high four bits correspond to unit 1. The low four bits of the next byte correspond to unit 2, and so on (see Figure 2–15). Thus, each device is presumed to have up to eight units, each assigned independently of the others. However, if the device is non-file-structured, units are not assigned independently; the monitor ASSIGN code ensures that ownership of all units is assigned to one job.

Figure 2–15: $OWNER Entry



When a background job, a foreground job, or a system job attempts to access a particular unit of a device, the monitor checks to be sure the unit being accessed is

either public or belongs to the requesting job. If another job owns the unit, a fatal error is generated.

The device is public if the four-bit field is 0. If the device is not public, the field contains a code equal to the job number plus 1. Since job numbers are always even, the ownership code is odd. For example, in a distributed foreground/background system, the owner field value for the background job is 1; for the foreground job it is 3. In a system with the system job feature, the owner field value for the background job is still 1; for the foreground job it is 17. The owner field value for a system job is 1 plus the job number.

As mentioned above, support for the $OWNER table is a system generation feature for the multi-job monitors. By default, the table is supported and lets a job 'own' a device handler unit (LOAD device=jobnam).

If support for the $OWNER table is explicitly removed during the system generation procedure, attempts to assign device unit ownership returns a fatal level error message. However, because the handler loading operation precedes the attempted ownership assignment, the handler is loaded by RT–11 before the error message is returned.

2.6.3.9 $PNAM2 Table

A system generation for extended device-unit support creates $PNAM2, a second physical device name table. The $PNAM2 table is used for extended device-unit logical to physical device name translation.

$PNAM2 contains the same index of physical device names as the $PNAME table. Handlers installed on the system that contain extended device-unit support are located in $PNAM2 with their single-letter device name in the same index position as the 2-letter device name entry in $PNAME. Handlers installed on the system that do not contain extended device-unit support are located in $PNAM2 with their 2-letter device name in the same index positon as their entry in $PNAME.

For example, if both the monitor and the DU handler are generated for extended device-unit support, the DU handler is located in $PNAME as ^RDU_ (a single space following the device name) and in the same index position in $PNAM2 as ^RD_ _ (two spaces following the device name). If the monitor is generated for extended device-unit support, the LP handler, without such support, is located as ^RLP_ (single space) at the same index position in both $PNAME and $PNAM2.

2.6.3.10 Adding a Device to the Tables

You can create free slots in the tables by deleting or renaming one or more of the device handler files from the system device and rebooting the system, or by issuing the REMOVE command. The INSTALL command can install a different device handler into the table after the system has been booted. However, INSTALL does not make a device entry permanent. For more information on installation, the DEV macro, and the bootstrap, see *RT–11 Device Handlers Manual*.

Chapter 3
Memory Mapping

This chapter provides the information you need to understand and manipulate memory in a mapped monitor system. First, it describes the as-booted mapped monitors and as-loaded job environments for the different types of jobs you can use with a mapped system. Next, it describes some of the features of mapped systems that you can control, such as processor modes and context switching. Then it describes the various programmed requests that RT–11 provides to let you control how your program maps memory. Some applications are discussed and you are shown how you can use .SETTOP to control the amount of memory available to your application. Next, a section presents some generic introductory information on how processor hardware controls mapping. (That information is also (and more completely) available in the appropriate processor handbook.) Toward the end of the chapter, some specific information is included on how the mapped monitor design affects basic RT–11 programs and some precautions you should take when writing programs for the mapped environment. Finally, two example programs are included with general information on debugging programs in the mapped environment.

The following terms are used in this chapter to describe mapped monitors and mapping memory in general. You can find detailed information about memory management hardware concepts in the handbook for your processor. If you do not have such a handbook, you can use the generic information in Section 3.9, which also contains information on these terms.

Table 3–1: Memory Mapping Terms

Term	Meaning
Active Page Register (APR)	The register through which the RT–11 monitor communicates with the MMU.
Completely virtual environment	A job environment in which none of the job is necessarily located in low memory.
Extended memory	The physical memory above the 28K word boundary that can be accessed only by using memory management hardware.
Low memory	The physical memory between 0 and 28K words.
Memory Management Unit (MMU)	A hardware unit that converts virtual addresses to physical addresses.
Page	A 4K-word section of virtual memory. Each address space is divided into 8 pages, numbered 0 - 7.

Table 3–1 (Cont.): Memory Mapping Terms

Term	Meaning
Page Address Register (PAR)	Contains information about the corresponding page (PAR2 corresponds to *page* 2).
Physical Address	The actual hardware address of a specific memory location. Physical addresses are not limited to 16 bits.
Relocation	The act of virtual to physical address conversion, performed by the MMU.
Status Registers	Two MMU registers located in the I/O page (PAR7); RT–11 uses registers 0 and 3.
Virtual Address	A value in the range of 0 through 177777. It is a 16-bit address within a program's 32K-word address space.
VMON	The completely virtual resident monitor; a simulated RMON for the completely virtual environment.
XM .SETTOP	Extended memory .SETTOP feature; not confined to XM monitor but rather applicable to all mapped monitors.

3.1 Default System Memory Layouts

Mapping is the process of associating virtual addresses with physical locations (see Section 3.9.2). The RT–11 mapped monitors manage the virtual address space by controlling the way the virtual addresses map to physical locations. The monitors do this by putting values into the Active Page Registers, thereby controlling the Memory Management Unit.

3.1.1 Mapped System Memory Layout

When you first bootstrap a mapped RT–11 system, the kernel and user mapping are identical. That is, the monitor puts the same values into both the kernel and user sets of Active Page Registers. Table 3–2 shows the initial values of the Active Page Registers. Figure 3–1 shows the default mapping that results from these values.

Table 3–2: Initial Contents of Kernel and User APRs

Page and APR No.	Kernel PAR	Kernel PDR	User I-Space PAR	User I-Space PDR
7	177600	177406	177600	177406
6	1400	77406	1400	77406
5	1200	77406	1200	77406
4	1000	77406	1000	77406
3	600	77406	600	77406

Table 3–2 (Cont.): Initial Contents of Kernel and User APRs

Page and APR No.	Kernel PAR	Kernel PDR	User I-Space PAR	User I-Space PDR
2	400	77406	400	77406
1	200	77406	200	77406
0	0	77406	0	77406

Figure 3–1: Default Mapping at Bootstrap Time

Figure 3–2 illustrates the initial locations of the mapped system components in physical memory. (Notice that the layout for low memory resembles the FB system arrangement described in Chapter 1.) When you first bootstrap a mapped system, the system device (SY) handler and RMON occupy the available memory just below the 28K-word boundary. To save space in low memory, often the system device handler is split so that various pieces, such as data structures and buffers, can be

located in extended memory. Other loaded device handlers occupy the space below RMON, followed by foreground and system jobs, if any, and the USR.

RMON executes in processor kernel mode and can access the low 28K words of memory and the I/O page. The USR also executes in kernel mode and is always memory resident in a mapped system. KMON executes in processor user mode, but since it is a privileged background job, it uses the same mapping as RMON. (Privileged jobs are described in Section 3.2.1.) Physical locations 0 through 476 contain the vectors.

Figure 3–2: XB and XM System Memory Layout

3.2 Default Job Mapping

The following three tables summarize the characteristics of privileged, virtual, and completely virtual jobs. Any distinction between characteristics for background or foreground/system jobs is included in the description.

Table 3–3: Characteristics of Privileged Jobs

Characteristic	Description
User memory starting address	Background job at kernel 0 Foreground/system (.REL) job at kernel address above USR
Vector location	Real vectors are part of the job.
Available user mapping	All of low memory and I/O page. For background jobs, only /V overlay regions are created in extended memory and are unmapped until transferred to by using overlay handler. For foreground/system jobs, no /V overlay regions are allowed.
Default context in I.CMAP word in job's impure area (can be altered by the .CMAP programmed request); For ZB and ZM only	<CM.UII!CM.NOS!CM.DUS!CM.SID>
Value in VIRT$ (bit 10) of $JSW	0
Original amount of address space available	32K words. Accesses the low 28K words of memory plus the I/O page.
Amount of potential address space	32K words. If some portions of virtual address space are already in use (by a background job, for example), this job can unmap them and remap the addresses to memory above 28K words. It must leave certain areas mapped whenever a user interrupt service routine could run.
Benefits	Compatible with unmapped systems.
Starting procedure	BG: R, RUN, or CCL command (.SAV) FG: FRUN or SRUN (.REL)
Static window	None—all are dynamic.
Static region	None—all are dynamic.
Possible number of windows	7 plus 1 window reserved
Possible number of regions	23_{10} plus 1 region reserved

)

Table 3–4: Characteristics of Virtual Jobs

Characteristic	Description
User memory starting address	Background job at kernel 500 Foreground/system (.REL) job at kernel address above USR
Vector location	Virtual vectors are used by the job; real vectors are not part of the job.
Available user mapping	Low memory from 500 to lowest address used by the USR. For background jobs, only /V overlay regions are created in extended memory and are unmapped until transferred to by using overlay handler.
Default context in I.CMAP word in job's impure area (can be altered by the .CMAP programmed request); for ZB and ZM only	<CM.UII!CM.NOS!CM.DUS!CM.SID>
Value in VIRT$ (bit 10) of $JSW	1
Original amount of address space available	Accesses only the virtual addresses within its own program bounds.
Amount of potential address space	32K words. Creates windows to describe the virtual address space between its own high limit and the 32K word boundary.
Benefits	Provides protection for operating system software and other programs; takes minimal physical memory away from other jobs.
Starting procedure	BG: R, RUN, or CCL command (.SAV) FG: FRUN or SRUN (.REL, .SAV; .SAV is recommended)
Static window	Extends from program's virtual address 0 to its high limit.
Static region	BG: Extends from physical location 500 to the lowest address used by the USR. FG: Extends from physical location 0 to the physical high limit of the job.
Possible number of windows	7 plus the static window.
Possible number of regions	23_{10} plus the static region.

Table 3–5: Characteristics of Completely Virtual Jobs

Characteristic	Description
User memory starting address	In extended memory.
Vector location	Virtual vectors are used by the job; real vectors are not used by the job.

Table 3–5 (Cont.): Characteristics of Completely Virtual Jobs

Characteristic	Description
Available user mapping	Jobs without /V overlays get 64K-bytes of user address space, the division of which depends on the setting of ALL64$ and IOPAG$ bits in $JSX word in block 0 of job file: 56K-bytes plus VMON when neither ALL64$ or IOPAG$ set in $JSX. 56K-bytes plus I/O page when IOPAG$ set in $JSX. 64K-bytes when ALL64$ set in $JSX. When separated I & D address space is supported and VMON or the I/O page is mapped, each is double-mapped for both I and D address spaces in user mode PAR7.
Default context in I.CMAP word in job's impure area (can be altered by the .CMAP programmed request)	For nonseparated I & D address space jobs is <CM.UII!CM.NOS!CM.DUS!CM.SID> For separated I & D address space jobs is <CM.UID!CM.NOS!CM.DUS!CM.SID>
Value in VIRT$ (bit 10) of $JSW	1
Original amount of address space available	Accesses only the virtual addresses within its own program bounds.
Amount of potential address space	32K-words of Instruction address space and 32K-words of Data address space
Benefits	Provides initial protection for monitor from user programs. Provides Instruction and Data address spaces and Supervisor mode under fully-mapped monitors.
Starting procedure	BG: R, RUN, V, VRUN, or CCL command (.SAV) FG: FRUN or SRUN (.REL, .SAV; .SAV is recommended)
Static window	Window mapped to PAR0
Static region	Physical memory that contains PAR0
Possible number of windows	7 plus the static window in each of User I-space and User D-space.
Possible number of regions	23_{10} plus the static region.

3.2.1 Privileged Mapping Environment

The default mapping in an extended memory system is privileged. To indicate a privileged job, VIRT$ (bit 10) of the Job Status Word remains 0. The mapped environment appears to a privileged job to be very similar to an unmapped environment. A privileged job can access the low 28K words of memory as well as the I/O page.

Privileged jobs, like virtual jobs, run in the processor User Mode. However, the monitor copies the contents of the kernel Active Page Registers into the user Active Page Registers. The default mapping for privileged jobs is thus the same as the default kernel mapping.

Privileged jobs do have all 32K words of virtual address space available to them. But much of that virtual address space is already mapped to operating system software, the I/O page, and—in the case of a privileged foreground or system job—to a background job or KMON. A privileged job can alter its default mapping through the use of extended memory overlays or programmed requests. It can map away all or part of the operating system to obtain a full 32K words of addressable memory for itself. For example, a program that needs to access the I/O page for only a limited time can explicitly map away from the I/O page when it is done using it.

Note that the static window and static region concept does not apply to privileged jobs. However, one window and one region are reserved by the monitor. Thus, privileged jobs have seven dynamic windows and 23 dynamic regions available to them, just as virtual jobs do.

When a privileged job creates a window and executes the mapping programmed requests, the default privileged mapping for that virtual address space is temporarily unmapped. The monitor maps the window using the contents of the internal window control block to the new region of memory. When the privileged job unmaps the window, the monitor remaps that virtual address space according to the contents of the kernel Active Page Register set. This differs from a virtual job that unmaps a window, in which the virtual addresses encompassed by the window are unusable until the window is remapped.

Since interrupt service routines execute in kernel mapping, privileged jobs containing user interrupt service routines should not change the mapping of interrupt service routines, the I/O page, or parts of the monitor during any time period in which an interrupt could possibly occur. The monitor depends on the fact that kernel and user mapping are identical when it services user interrupts.

3.2.1.1 Privileged Background Job

Use the monitor R or RUN commands to start a privileged background job. Figure 3–3 illustrates the mapping for a privileged background job.

3.2.1.2 Privileged Foreground or System Job

Use the monitor FRUN command to start a privileged foreground job. Use the SRUN command to start a privileged system job.

Figure 3–4 illustrates the mapping for a privileged foreground or system job.

3.2.2 Virtual Mapping Environment

Jobs that run with virtual mapping execute in the processor's User or Supervisor mode. Virtual jobs do not use kernel-compatible mapping; virtual background jobs load into memory at physical address 500. Virtual jobs cannot load over the USR, RMON, or the I/O page. Virtual mapping is the better mapping mode to use for a

Figure 3–3: Privileged Background Job

job that does not require privileged access to the vector area, the monitor, or the I/O page, since it protects these system areas from virtual jobs.

The first 500 bytes of each virtual job image are its virtual vector and system communication areas. The static window includes the virtual addresses between the program's virtual address 0 and its high limit. The size of the static region

Figure 3–4: Privileged Foreground or System Job

varies depending on whether the virtual job is a foreground or a background job and on the size of the job.

When you first run a virtual job, it can access only those virtual addresses that are within its own program bounds and that are also mapped to physical memory. However, a virtual job can use any remaining virtual address space between its own

high limit and the 32K-word address boundary. It can create one or more regions in extended memory, and one or more virtual address windows. It can then map a window to a region, thus accessing extended memory. If a virtual job unmaps a window, it cannot use the virtual addresses encompassed by the window unless it remaps the window. A virtual job can also use the XM (extended memory) .SETTOP feature and extended memory overlays.

3.2.2.1 Selecting Virtual Mapping

You indicate that a job is to use virtual mapping by setting bit 10 of the Job Status Word before you run the program. If a particular job is always virtual, set bit 10 at assembly time. Use the following instructions to do this:

```
.ASECT
.=$JSW
.WORD VIRT$
.PSECT
```

Or, if you prefer, select the program's mapping by running SIPP and patching location 44 ($JSW) in the job's .SAV or .REL file before you run the program.

NOTE

Do not change the value of VIRT$ of the $JSW when the program is running. Doing so interferes with accurate processing of I/O requests and can cause unpredictable results.

3.2.2.2 A Virtual Background Job

Use the monitor R or RUN command to start a virtual background job. You can also start the job through CCL by typing only the program name. The file should have the .SAV file type. A virtual background job loads into memory starting at physical location 500. Its highest physical address is equal to the size of the program in octal plus 500.

The static region for a virtual background job begins at physical location 500 and extends to the lowest address used by the USR. This prevents a virtual background job from directly accessing the physical vector area between locations 0 and 476. As a result, the vectors are protected from virtual jobs. Figure 3–5 illustrates the mapping for a virtual background job. Figure 3–6 shows how a virtual background job can map a window into the static region to use the available memory just below the USR.

Figure 3–5: Virtual Background Job

3.2.2.3 A Virtual Foreground or System Job

Use the FRUN monitor command to start a virtual foreground job and the SRUN command to start a virtual system job. You should link these jobs as background jobs with the .SAV file type, rather than as foreground or system jobs with the .REL file type. You can FRUN or SRUN a virtual .SAV image because virtual foreground jobs require no relocation information. Thus, the .SAV files are smaller on disk than .REL files, and they load into memory faster.

When a foreground job is loaded, it uses the physical locations just below the lowest loaded handler or previously loaded system job. The USR slides down in memory, if necessary, to accommodate the foreground job. The job's virtual addresses between 0 and 476 represent the virtual vector and system communication areas. As with the background virtual job, the static window starts at virtual address 0 and extends to this foreground program's high limit, rounded up to a 32-word multiple.

Figure 3–6: Virtual Background Job Mapping into the Static Region

The static region begins at physical location 500 and extends to the program's physical high limit. The foreground impure area is located in physical memory just below the program. However, no virtual addresses are mapped to the impure area, so a virtual foreground job cannot directly access the contents of the impure area. As a result, the impure area is protected from a virtual foreground job. Figure 3–7 illustrates the mapping for a virtual foreground or system job.

3.2.3 Completely Virtual Mapping Environment

The concept of the completely virtual environment applies equally to separated and nonseparated I-D space virtual jobs. The environment is created (the job is loaded) by a rewritten VBGEXE.SAV utility. Section 3.2.3.2 describes the environment for nonseparated I-D space jobs. Section 3.2.3.3 describes the environment for separated I-D space jobs.

Figure 3–7: Virtual Foreground or System Job

3.2.3.1 V/VRUN Keyboard Command

The V and VRUN commands run a job in the completely virtual environment and parallel the syntax of the R and RUN commands; use the V command if the job resides on the system device and the VRUN command if the job resides on the default data device.

The V and VRUN commands run VBGEXE, which closely parallels the pre-V5.6 VBGEXE virtual background utility—with one important difference. You no longer need to specify both input and output files when using the single-line command format.

The V and VRUN commands also set bit VRUNV$ (000004) in the job's memory JSW. (The R and RUN commands clear bit VRUNV$, so the state of bit VRUNV$ indicates how the job was run. A chain does not affect the state of VRUNV$.)

The following is valid command syntax:

.VRUN program RET

For example:

```
.VRUN MACRO  RET
*FOO,FOO=FOO  RET
*
```

3.2.3.2 Nonseparated I-D Space Completely Virtual Environment

A job that is not built for separate I-D address space can be loaded and run in a completely virtual environment either manually or automatically. You manually load and run a virtual job by using the V or VRUN keyboard command. You automatically load and run a virtual job by setting certain bits in various data structures, as described below.

3.2.3.2.1 Automatic Job Running

You can cause a job to be automatically loaded into a completely virtual environment and run from that environment:

- Set bit 000200 (VBGEX$) in offset 4 ($JSX word) in block 0 of the file save image at assembly time or by using SIPP. See *RT–11 Volume and File Formats Manual* for a description of the save image and $JSX.
- Include the command SET RUN VBGEXE in your start-up command file (or issue it). That command sets bit 000010 (CF3.VB) in the third configuration word ($CNFG3), as described in Chapter 2. The state of the system (whether RUN is set to VBGEXE or NOVBGEXE) is displayed by SHOW CONFIGURATION).
- VBGEXE.SAV must reside on your system device.
- Sufficient extended memory must exist to load the job.
- Run the job using R, RUN, V, or VRUN. If using R or RUN and the job or environment is not valid for running under VBGEXE, the monitor attempts to run the job as a straight background job. If using V or VRUN, all conditions must be satisfied or VBGEXE returns an error and the job does not run.

The restrictions concerning jobs under VBGEXE remain in force. Such a job cannot directly access the monitor or require executable code in low memory. The *RT–11 System Release Notes* provides more information.

3.2.3.2.2 Stopping Automatic Job Running

You can stop the automatic running of a non-I-D address space job or utility (which currently automatically runs in the completely virtual environment) by performing the following customization patch. The patch sets the bits in the $JSX word (offset 4) in absolute block 0 of a save image to zero.

```
.RUN SIPP  RET
*progrm.SAV  RET
Base?    0  RET
Offset?  4  RET
```

```
  Base        Offset        Old       New?
   000000     000004     nnnnnn       000000 RET
   000000     000006     nnnnnn       CTRL/Y RET
* CTRL/C
.
```

3.2.3.3 Separated I-D Space Completely Virtual Environment

A job that is built for separate I-D address space is always loaded and run in the completely virtual environment. A diagram and contents of the *extended* save image is located in the *RT–11 Volume and File Formats Manual*.

The following points should be kept in mind when writing and building a program for separated I-D space:

- Code and data must be respectively split into I and D p-sects.
- Code cannot be moved onto the stack for execution.
- The program cannot contain code that directly accesses the monitor or requires kernal memory.

  It cannot contain interrupt service routines.
- No .REL linking is allowed.
- The job is linked into an 'extended save image', as described in the *RT–11 Volume and File Formats Manual*. See the *RT–11 Commands Manual* for LINK commands or the *RT–11 System Utilities Manual* for LINK utility options.

  New LINK error messages are in the *RT–11 System Message Manual*.
- The .ABS . p-sect for a separated I-D space job has the D-space attribute. (For nonseparated I-D space jobs, the .ABS . p-sect continues to have the I-space attribute.)

Run the job using R, RUN, V, or VRUN. If the job or environment is not valid, the job does not run and VBGEXE returns an error message.

VBGEXE is the loader for extended save images that support separated I-D space addressing and Supervisor mode. VBGEXE loads block 0 and the root segment of the save image address spaces into a single local region in extended memory. VBGEXE uses $USRTO (offset 50) in the D-space absolute block 0 to determine which data blocks to load. VBGEXE uses $USRTO (offset 50) in the I-space virtual block 0 to determine which instruction blocks to load.

VBGEXE writes the following code into I-space virtual locations 0 and 2 in memory:

```
BIC     R0,R0
.ASTX                   ; An EMT 356 instruction
```

VBGEXE also marks any overlay segments as nonresident. VBGEXE performs different operations to the overlay segments, depending on whether the overlay is extended memory (/V) or disk (/O):

- If extended memory, VBGEXE writes a RETURN instruction in the first word of each /V overlay partition to indicate each /V overlay partition is nonresident.

- If disk, VBGEXE writes a RETURN instruction to all address locations in the /O overlay area to mark as nonresident each overlay segment in that area. The overlay handler will actually load the overlay segments, as called.

Once the extended save image is loaded in memory, the job is started, based on its transfer address (stored in $USRPC (offset 40) of the I-space virtual block 0).

3.2.3.4 Chaining in the Completely Virtual Environment

The environment into which a program chains is determined by data structures in both the running program and the program to which it chains.

The most important data structure in the running program is the VRUNV$ bit (000004) in the program's memory JSW. A chain does not alter the state of VRUNV$.

- If set, the program was started by a V or VRUN command.
- If clear, the program was not started by a V or VRUN command.

In the program to be chained to, the contents of $JSX (location 4 in the file save image block 0) determine the environment.

- If VBGEX$ (bit 00200) is set, the job can be run in the completely virtual environment.
- If NOVBG$ (bit 000100) is set, the job cannot be run in the completely virtual environment. If the job is run by R or RUN, it runs in the standard environment. If the job is run by V or VRUN, it will abort.

In the case of the running job's JSW (in memory) having VRUNV$ set, the chain process is:

- In the program to be chained to, the monitor checks location 0 in the file save image block 0. If location 0 is RAD50 HAN, the second program is a runnable handler and cannot be run in completely virtual environment. The handler is run in standard manner.
- If location 0 is not RAD50 HAN, the monitor then checks $JSX (location 4). If $JSX bit NOVBG$ (000100) is clear, the monitor loads and runs second program as a completely virtual job. If NOVBG$ is set, the monitor runs second program as a standard job.
- Whether the chained-to job runs in the completely virtual environment or not, the chain has no effect on the state of VRUNV$ bit in the memory JSW. Therefore, the process is repeated for each chain.

In the case of the running job's JSW (in memory) having VRUNV$ clear, the chain process is essentially the same; the same checks are made. However, because VRUNV$ is clear, the VBGEX$ bit (bit 000200) in the chained-to job's $JSX word is also checked for permission to run the job in the completely virtual environment. If VBGEX$ is set and the conditions described above are met, the chained-to job is run in the completely virtual environment. Otherwise, the job is run in the standard environment.

3.2.3.5 Completely Virtual Background Job

Use the monitor V or VRUN command to start a completely virtual background job. The R or RUN command can also be used to automatically start a completely virtual background job (see Section 3.2.3.2.1). The file should have the .SAV file type. A completely virtual background job loads into available extended memory. None of the job's address space is in low memory although some low memory is used by the monitor on behalf of the job for satisfying any .FETCH, .CDFN, and .QSET requests. A completely virtual job's impure area resides in RMON like all other background jobs.

Since none of the job's memory is mapped to low memory, the physical vector area between physical locations 0 and 476 is not directly accessible by a completely virtual background job.

The following figures show the initial mapping for completely virtual background jobs:

- Figure 3–8 shows the initial mapping for a completely virtual background job without separated I & D address spaces and with no extended memory overlays.
- Figure 3–9 shows the initial mapping for a completely virtual background job with separated I & D address spaces and with no extended memory overlays.
- Figure 3–10 shows the initial mapping for a completely virtual background job without separated I & D address spaces and with extended memory overlays.
- Figure 3–11 shows the initial mapping for a completely virtual job background with separated I & D address spaces and with extended memory overlays.

3.2.3.6 Completely Virtual Foreground/System Job

Use the FRUN SY:VBGEXE.SAV/NAME:prgnam command to start a completely virtual foreground job. Use the SRUN SY:VBGEXE.SAV/NAME:prgnam command to start a completely virtual system job. The file should have the .SAV file type. A completely virtual foreground or system job loads into available extended memory. None of the job's address space is in low memory although some low memory is used by the monitor on behalf of the job for satisfying any .CDFN and .QSET requests. A completely virtual foreground or system job's impure area also resides in low memory like all other foreground and system jobs. Since none of the job's memory is mapped to low memory, the physical vector area between physical locations 0 and 476 is not directly accessible by a completely virtual foreground or system job.

Figure 3–8: Completely Virtual Background Job

The following figures show the initial mapping for completely virtual foreground /system jobs:

Figure 3–9: Completely Virtual Background Job with Separated I & D Address Spaces

- Figure 3–12 shows the initial mapping for a completely virtual foreground/system job without separated I & D address spaces and with no extended memory overlays.

Figure 3–10: Completely Virtual Background Job with Extended Memory Overlays

- Figure 3–13 shows the initial mapping for a completely virtual foreground/system job with separated I & D address spaces and with no extended memory overlays.

Figure 3–11: Completely Virtual Background Job with Separated I & D Address Spaces and Extended Memory Overlays

- Figure 3–14 shows the initial mapping for a completely virtual foreground/system job without separated I & D address spaces and with extended memory overlays.

- Figure 3–15 shows the initial mapping for a completely virtual foreground/system job with separated I & D address spaces and with extended memory overlays.

3.3 Kernel, User, and Supervisor Processor Modes

In addition to its primary function of managing address spaces, the memory management system must provide some kind of protection for the monitor. To implement protection, the processor provides three modes of operation: **Kernel mode**, **Supervisor mode**, and **User mode**. The modes provide a mechanism for separating system-level functions (kernel mode) from application-level functions (User and Supervisor modes).

Each mode has its own stack pointer and its own set of eight Active Page Registers, one set for each supported address space. Therefore, each processor mode also makes its own assignments of virtual addresses to physical locations: each mode has its own mapping. Figure 3–16 shows how the value in bits 14 and 15 of the Processor Status word determine in which processor mode execution takes place.

Routines that run in **Kernel mode** are generally part of the run-time operating system software and must not be corrupted by other programs. RT–11 uses the processor's kernel mode for RMON and the USR, for interrupt service routines, and for device handlers, including .SYNCH and .FORK routines. Interrupts and traps vector through kernel mapping and cause execution to continue in kernel mode.

Routines that run in **User mode** are generally part of application programs. They are prevented from executing instructions that could corrupt the monitor or halt the computer. For example, a RESET instruction acts as a NOP instruction in user mode, and a HALT instruction generates a trap to 10. RT–11 uses the processor's User mode for KMON, for system utility programs, and for application programs and their completion routines.

Routines that run in **Supervisor mode** generally manage system or application libraries. Supervisor mode provides the same protection for the system internals as User mode.

Since each processor mode uses its own set of Active Page Registers, mapping between modes is not necessarily identical. For example, if user virtual address 20010 is associated with physical address 40210, it does not necessarily mean that kernel virtual address 20010 is also mapped to physical address 40210. In fact,

Figure 3–12: Completely Virtual Foreground/System Job

kernel virtual addresses are often mapped to different sections of physical memory from user virtual addresses. The mapping depends entirely on the contents of the Active Page Registers. Thus, changing between processor modes has some

Figure 3–13: Completely Virtual Foreground/System Job With Separated I & D Address Spaces

interesting implications: referencing the same virtual addresses in different modes can cause a program to access different physical locations. Figure 3–17 shows an

Figure 3–14: Completely Virtual Foreground/System Job With Extended Memory Overlays

example in which virtual address 0 in kernel mode maps to physical location 0; in

Figure 3–15: Completely Virtual Foreground/System Job With Separated I & D Address Spaces and Extended Memory Overlays

User mode, virtual address 0 maps to physical location 500. This is the mapping scheme RT–11 uses for a virtual job at load time.

Figure 3–16: Processor Status Word and Active Page Registers

Figure 3–17: Mapping the Same Virtual Addresses to Different Physical Locations

3.4 How Programs Control Mapping

Mapping is associating virtual addresses with physical locations. Mapped monitors control mapping by putting values into the Active Page Registers, thus controlling the Memory Management Unit. The mapping process can be, but is not necessarily, transparent to the program.

The monitor provides the means by which system and application programs can direct mapping operations and experience the benefits of accessing extended memory without concern for the specifics of the Memory Management Unit operations. In fact, your programs should never directly access the Memory Management Unit Status Registers. Programs communicate their extended memory requirements to the monitor through a collection of programmed requests. These requests store or modify information in data structures within the programs. Based on the contents of these data structures, the monitor modifies its own internal control blocks and puts the correct values into the Active Page Registers to perform the appropriate mapping action.

In order to access extended memory, a program must:

- Tell the monitor how much physical address space it needs.
- Describe the virtual addresses it needs to the monitor.
- Direct the monitor to associate the virtual addresses with the physical locations. That is, it must **map** the virtual addresses to the physical locations.

Background, foreground, and system jobs can all access extended memory by following the three steps described above.

The monitor and the programs use certain software concepts to describe the virtual addresses and the physical memory locations. The following sections describe the concepts of **physical address regions**, **virtual address windows**, and the **program's logical address space**.

3.4.1 Physical Address Regions

A physical address region is a segment of physical memory consisting of contiguous 32_{10} word units. Each region must begin on a 32-word boundary.

Regions are created and allocated to programs by RMON. The monitor maintains information about each region in a region control block that is located in the Mapping Context Area (MCA) region.

Every program is allocated at least one region—identified as region 0. This is a static region and cannot be altered or eliminated by the program for which it is allocated.

In addition to static region 0, a program can have up to 23 dynamic regions—identified as regions 1 through 23. These regions provide a program with access to extended memory. The program describes each region it needs in a data structure called a region definition block and requests the regions by means of an RT–11 programmed request.

The monitor assigns an identification number to each region requested by the program. The region identification number is actually a pointer within your job's MCA region to the start of the region's control block.

The purpose of a region is to describe a portion of the physical address space, thus making it available for mapping and permitting a program to use the physical addresses. Sections of physical address space that are not part of a region are unavailable to a program.

Information about a physical address region is contained in a 5-word data structure in your program called a region definition block. The monitor collects information from the region definition block and stores it in a different internal data structure called the region control block. The region control block is located in the MCA region. Section 3.6 provides more detailed information on the region definition and control blocks.

A program can request two types of region: local regions, which are controlled by programs, and global regions, which are controlled by the system.

3.4.1.1 Local Regions

A local region is a segment of extended memory that a program requests for its exclusive use. The program defines the region and maps its physical addresses to a range of virtual addresses. As long as the region exists, the region is the exclusive property of the program that requested it. When the local region is no longer needed, the program issues a request to eliminate it. When the program exits, the local region is implicitly eliminated.

The data structures and programmed requests that a program uses to create, map, and eliminate a local region are described in Section 3.6.

3.4.1.2 Global Regions

A global region is a segment of extended memory that a program can request for its exclusive use or share with other programs. The program maps the physical addresses to virtual addresses in the same way as for a local region. The monitor creates the global region and attaches the program to it. As long as the program remains attached, it has sole possession of the region. To share a region with other programs, the program issues a request to detach itself from the region. Any program can then attach itself to the global region. The operating system manages the sharing of a global region by allocating a local region within the global region to each attached program. The program that originally requested the region can reattach itself to the region but cannot regain exclusive possession.

A program can define a global region to be permanent, subject to elimination by any program that is currently attached to it, or set for automatic elimination after the last program has finished using it. An example of an address space that is located in a permanent global region is the RT–11 I/O page, which cannot be eliminated.

Up to ten global regions can exist concurrently in extended memory; a program can attach to any or all global regions at the same time.

The data structures and programmed requests that a program uses to create, use, and eliminate global regions are the same ones that it uses for local regions.

In multi-job systems, programs can share global regions simultaneously, whereas in the single-job systems, programs must share global regions serially. That is, in single-job systems, a program runs as the single background job and accesses the region. When the program exits, another program can run and attach itself to the same region, accessing the data left by the former program.

3.4.1.3 Handler Global Regions

Certain RT–11 handlers create global regions in extended memory. The name that a handler assigns to a handler region has a format that includes a space between the handler name and the required dollar sign ($), as follows: :

dd $

where *dd* is the handler name.

For example, the following name identifies a global region created by the VM handler:

VM $

A program that wishes to attach to a global region created by an RT–11 handler must use this format to identify the region.

A user-written handler that wishes to create a global region should follow the RT–11 format described above for naming a handler region.

3.4.2 Virtual Address Windows

A program that needs to access extended memory must also communicate to the monitor a description of the virtual addresses it plans to use. While the monitor uses the concept of pages to describe virtual addresses to the memory management unit, programs describe the virtual address space to the monitor by using the software concept of virtual address windows.

A virtual address window is a section of the 32K-word virtual address space consisting of contiguous 32_{10}-word units. A window, like a page, must begin on a 4K-word boundary. However, unlike a page, whose maximum size is 4K words, a window can be as large as 32K words and can encompass one or more pages. For each supported address space, there can be as many as eight virtual address windows or as few as one. The monitor assigns identification numbers to the windows when your program creates them.

The purpose of a window is to describe a section of virtual address space to the monitor, and thus permit a program to use those virtual addresses. Windows cannot overlap each other. (While a job can describe a new window that overlaps an existing one, the old one is eliminated when the new one is created.) And, sections of virtual address space, if any, that are not part of a window are not available for a program to use, unless the job is privileged. Each window that is less than 4K words causes a

discontinuity in the program's virtual address space. A memory management fault results if the program tries to access a virtual address that does not fall within a mapped window. (A window is not useful until it is also mapped.)

The monitor can assign physical addresses to the virtual addresses encompassed by windows by calculating the number and size of the pages involved and putting values into the corresponding Active Page Registers for those pages. Figure 3–18 shows how virtual address space can be divided into windows.

Figure 3–18: Virtual Address Space and Three Windows

Information about a virtual address window is contained in a seven-word data structure in your program called a *window definition block*. The monitor collects information from the window definition block and stores it in a different internal data structure called the *window control block*. The window control block is located in Mapping Context Area (MCA) region. Section 3.6 provides more detailed information on the window definition and control blocks.

3.4.2.1 The Static Window

The first window, called the static window, is created for a job by the monitor at run time. You cannot use the static window in privileged jobs; its data structures are reserved.

For a virtual job, the static window begins at virtual address 0, and its size is equal to the size of your program's base segment, up to the program's high limit. It contains the program's root, stack, virtual vectors, overlay handler, and low memory overlays. Instructions, data, and buffers can appear in extended memory overlays or in extended memory .SETTOP buffers; they are contained in a different window and region. You can refer to the static window by using an identification of 0. Your program cannot eliminate the static window or change its mapping.

For a completely virtual job without address space separation, the static window starts at virtual address 0 and goes to the minimum of 17776 or the program's high limit. It contains the program's root, stack, virtual vectors, overlay handler, and low memory overlays. Instructions, data, and buffers can appear in extended memory overlays or in extended memory .SETTOP buffers; they are contained in a different window and region. You can refer to the static window by using an identification of 0. Your program cannot eliminate the static window or change its mapping.

For a completely virtual job with I & D address space separation, the I and D static windows start at virtual address 0 and go to the minimum of 17776 or the program's high limit. The program's root, stack, virtual vectors, overlay handler, and low memory overlays are split across the static windows. No instructions are allowed in XM .SETTOP buffers. Your program cannot eliminate the static window or change its mapping.

3.4.2.2 Dynamic Windows

If your program needs to access more memory than the amount allocated at run time, it can create one or more dynamic windows and map their virtual addresses to physical locations. A program can create up to seven dynamic windows for each supported User mode address space and eight dynamic windows for each supported Supervisor mode address space. A program can create, eliminate, map, and remap any of the dynamic windows.

3.4.3 Program's Logical Address Space (PLAS)

A program's logical address space is the range of physical address space effectively available to the program as a result of mapping operations. That is, all physical locations that are part of a region can be accessed by the program through mapping operations, and are thus part of its logical address space. The Program's Logical Address Space is abbreviated as PLAS, a term often used to refer to extended memory support in general.

3.4.4 Mapping and Context Switches with Virtual and Privileged Jobs

Besides the context information described in detail in Section 2.4.2, mapping context information is saved when virtual and privileged jobs are switched out. The following is the basic process by which mapping information is saved and restored:

- The program issues a .CMAP request and defines (or redefines) the mapping support for this job.
- Data structures that describe the mapping support and point to the MCA region are stored in the job's impure area.

- The actual mapping structures for the job are saved in a region in extended memory called the Mapping Context Area or MCA. For each job, the MCA contains:
  - Region control blocks

    Twenty-four (decimal) RCBs for each job.
  - PARs and PDRs

    PARs and PDRs are stored to support each processor mode and address space specified by the .CMAP request for this job. This could mean 8 of each for a single-mapped (XM) system up to 32 of each for a fully mapped (ZM) system that supports all processor modes/address space combinations.
  - Window control blocks

    Depending on system mapping, between 8 and 32 WCBs can be supported for each job (8 for XB,XM and 32 for ZB,ZM).
  - Memory mapping register 3 (MMR3)

    One copy of MMR3 is stored for each job.

See Section 3.6.6 for specific information on the MCA region.

The monitor restores the mapping structures enabled by .CMAP when the job is switched in to execute. The monitor behaves differently when switching in a job, depending on whether the job is new (is being switched in for the first time) and whether the job is privileged, virtual, or completely virtual.

When the monitor switches in a new job (for the first time), it assumes that the job is privileged. It copies the contents of the kernel mapping registers into the user registers. The job can then access the low 28K words of memory plus the I/O page.

If the new job is in fact privileged, the monitor checks the window and region control blocks in the Mapping Context Area (MCA) region. If the job defined and mapped one or more windows, the monitor sets up the mapping based on the contents of the internal control blocks, thus altering the default privileged mapping for those windows.

If the new job is virtual, the monitor clears the user mapping registers. Then it scans the window and region control blocks. The monitor maps only the portion of the job's virtual address space that was defined in a window and mapped to a region at the time the job was switched out. Of course, any attempt to access an unmapped address causes a memory management fault. Unused portions of virtual address space remain unmapped unless the virtual job explicitly maps them.

The monitor performs the switching operation differently when it switches in a job that has previously been switched out. Single-mapped monitors restore user APRs from that job's MCA. Fully-mapped monitors restore all user and supervisor APRs from that job's MCA that are enabled by I.CMAP.

The monitor determines whether a job has been previously switched out by examining LOAD$ (bit 11) in the I.STATE word (offset 2) in the job's impure area.

If LOAD$ is set, it signifies that the job has yet to be switched in. Thus, it is set for the first context switch of a job. If LOAD$ is clear, it signifies that the job has been switched in at least once. LOAD$ is cleared by the code that sets up the job's mapping the first time the job is switched in.

The monitor also ignores a .CNTXSW programmed request if it occurs in a virtual job. The entire job is saved by the switch, and the virtual job is not permitted to directly access the vector area in any case.

3.5 Introduction to the Extended Memory Programmed Requests

It is not difficult to access extended memory in a MACRO–11 program through the programmed requests, once you understand the general procedures you must follow and the tools RT–11 provides. Essentially, if your program does its own management of extended memory, you must first establish window and region definition blocks. Next, you must specify the amount of physical memory the program requires, and describe the virtual addresses you plan to use. Do this by creating regions and windows. Then, associate virtual addresses with physical locations by mapping the windows to the regions. You can then remap a window to another region or part of a region. You can also eliminate a window or a region. In any case, once the initial data structures are set up, you can manipulate the mapping of windows to regions to suit your needs.

Table 3–6 summarizes the actions a program that uses extended memory may need to take. It also lists the appropriate procedures for the program to follow. Familiarize yourself with the procedures and the corresponding programmed requests and macro calls. The *RT–11 System Macro Library Manual* provides detailed information on the format of each programmed request and macro call. Study this information before you attempt to write an extended memory program.

Table 3–6: Summary of Activities for a Program in an Extended Memory System

Activity	Procedure to Follow
Define offsets and symbols for a region definition block.	Use the .RDBDF or .RDBBK macro.
Set up a region definition block and specify the region name, size, and base.	Use the .RDBBK macro.
Define-memory mapping context.	Use the .CMAP programmed request.
Create a local region.	Use the .CRRG programmed request.
Create and attach to a global region.	Use the .CRRG programmed request.
Attach to an existing global region.	Use the .CRRG programmed request.
Detach from a global region.	Use the .ELRG programmed request.

Table 3–6 (Cont.): Summary of Activities for a Program in an Extended Memory System

Activity	Procedure to Follow
Confirm the status of the new region.	Examine the contents of the region definition block after you use the .CRRG request to create the region. (Check the status bits in the status word.)
Define offsets and symbols for a window definition block.	Use the .WDBDF or .WDBBK macro.
Set up a window definition block and describe the window.	Use the .WDBBK macro.
Create a window.	Use the .CRAW programmed request.
Confirm the status of the new window.	Examine the contents of the window definition block after you use the .CRAW request to create the window. (Check the status bits in the status word.)
Associate a window with a particular region as preparation for mapping the window.	Move the region identification from R.GID in the region definition block to W.NRID in the window definition block.
Map a window to a region (explicitly).	Use the .MAP programmed request.
Map a window to a region (implicitly).	Set WS.MAP in the window definition block and load W.NRID before you issue the .CRAW request to create the window. This procedure creates the window and then maps it to a region.
Obtain the current memory mapping context.	Use the .GCMAP programmed request.
Obtain the current mapping status of a particular window.	Use the .GMCX programmed request.
Unmap a window (explicitly).	Use the .UNMAP programmed request.
Unmap a window (implicitly).	Use the .MAP programmed request to map the window elsewhere. You can also unmap a window by eliminating the region to which it is mapped, or by eliminating the window itself.
Eliminate a window.	Use the .ELAW programmed request.
Eliminate a local or global region.	Use the .ELRG programmed request.

3.6 Extended Memory Data Structures

A program in an extended memory environment communicates with the monitor through special data structures. For each region it defines, a program contains one *region definition block* to describe the size of the extended memory region. The monitor also maintains a set of internal data structures. The *region control block*, located in the Mapping Context Area (MCA) region in extended memory, describes a region. The monitor can maintain up to 24 region control blocks per job. For each

window it defines, a program also uses one *window definition block* to describe the virtual addresses encompassed by that window. The *window control block*, located in the MCA region, is the monitor's internal description for a window. The monitor can maintain up to eight window control blocks for each job under single-mapped monitors. Under fully-mapped monitors, up to eight WCBs can be maintained for each supported address space for each job. Finally, the monitor allocates regions in extended memory based on its internal *free memory list*.

The following sections describe these data structures and show, where necessary, how to create them.

3.6.1 Region Definition Block

A *region definition block* is a 6-word area in your program that contains information about a region you define in extended memory. The monitor uses the region definition block to communicate with your job when you issue a .CRRG or .ELRG programmed request. You must set up the region definition block in your program and define its symbolic offsets before you can create a region in extended memory. You must then place the region's size in the region definition block. After you create the region, the monitor returns its identification and some status information to you through the region definition block. Each time your program needs to refer to this region, it uses the region identification. (Since the monitor creates the static region for you, you do not know its identification. You can always refer to the static region by using 0 as its identification.) Figure 3–19 and Table 3–7 show the structure of a region definition block.

Figure 3–19: Region Definition Block

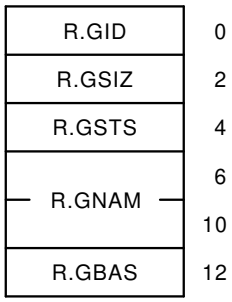


Table 3–7: Region Definition Block (.RDBDF)

Offset	Symbol	Modifier	Contents
0	R.GID	Monitor's .CRRG routine	A unique region identification. Use it later to reference that region. The region identification is a pointer within the job's impure area to the region control block. The identification for the static region in a virtual job is 0.
2	R.GSIZ	.RDBBK macro or user program	The size of the region you need, in 32_{10}-word units. When attaching a local region to an existing global region, specifying a zero value obtains the whole global region.
4	R.GSTS	Monitor's .CRRG routine	The region status word.
6	R.GNAM	.RDBBK macro	Two-word global region name.
12	R.GBAS	.RDBBK macro	Global region base address.

3.6.1.1 Region Status Word (R.GSTS)

The region status word contains information on the status of local and global regions. Table 3–8 shows the bits in the region status word and their meaning. Bits 0 through 3 are reserved for future use by Digital.

Table 3–8: Region Status Word (.RDBDF)

Bit	Symbol	Pattern	Meaning When Set
15	RS.CRR	100000	The monitor created the region successfully. The .CRRG routine sets this bit; the .ELRG routine clears it.
14	RS.UNM	40000	One or more windows were unmapped as a result of eliminating the region. The .ELRG routine sets this bit when necessary.
13	RS.NAL	20000	Region was not previously allocated.
12	RS.NEW	10000	An attach request to a global region was made, and the global region was not found. The global region was created.
11	RS.GBL	4000	Create a local region within a global region. If global region is not found, returns error. (No error returned if RS.CGR is set.)
10	RS.CGR	2000	Create a local region within a global region. If a global region is not found, create a global region.

Table 3–8 (Cont.): Region Status Word (.RDBDF)

Bit	Symbol	Pattern	Meaning When Set
9	RS.AGE	1000	Enable automatic global elimination. Eliminates global region when last job using global region detaches; when count in GR.SHC of global region control block is zero. RS.EGR need not be set.
8	RS.EGR	400	Eliminates global region. Global region area is returned to free memory list. Count in GR.SHC of global region control block must be zero.
7	RS.EXI	200	Eliminate global region when exiting or aborting from job.
6	RS.CAC	100	Enable cache-bypass.
5	RS.BAS	40	Explicitly assign base address for this global region.
4	RS.NSM	20	Explicitly assign the base address for this global region exclusive of system memory. *System memory* is defined as the I/O page and that memory below the address you derive from the value stored in RMON fixed offset 420, $MEMSZ.
1	RS.DSP	2	Reserved.
0	RS.PVT	1	Reserved.

3.6.1.2 .RDBDF Macro

The .RDBDF macro defines symbols for the local and global region definition block. It defines the symbolic offset names for the definition block and the names for the region status word bit patterns. In addition, the macro defines the length of the definition block by setting up the following symbol:

```
R.GLGH = 10.
```

The .RDBDF macro does not reserve space for the region definition block.

The format of the .RDBDF macro is as follows:

.RDBDF

The .RDBDF macro expands as follows:

```
R.GID   =: 0
R.GSIZ  =: 2.
R.GSTS  =: 4.
R.GNAM  =: 6.
R.GLGH  =: 10.
RS.CRR  =: 100000
RS.UNM  =: 40000
RS.NAL  =: 20000
RS.NEW  =: 10000
RS.GBL  =: 4000
RS.CGR  =: 2000
RS.AGE  =: 1000
```

```
RS.EGR  =: 400
RS.EXI  =: 200
RS.CAC  =: 100
RS.BAS  =: 40
RS.NSM  =: 20
```

3.6.1.3 .RDBBK Macro

The .RDBBK macro defines symbols for the local and global region definition block and reserves space for it. The .RDBBK macro invokes the .RDBDF macro. If you call the .RDBBK macro, you need not call the .RDBDF macro.

The format of the .RDBBK macro is as follows:

.RDBBK rgsiz,rgsta,name[,base]

where:

rgsiz is the size of the dynamic region expressed in 32_{10}-word units.

rgsta is the region status byte.

name is the name of the global region in RAD50 characters.

base is the base address of the global region in 32_{10}-word units. A value of zero (or omitted parameter) means any available base address is acceptable.

3.6.2 Region Control Block

A region control block is a three-word area in the Mapping Context Area (MCA) region, whose contents are maintained by the monitor. A virtual job dedicates one region control block to the static region. For a privileged job, one region control block is reserved by the monitor and cannot be used by a program. Thus, all jobs can have up to 23 dynamic regions whose status is maintained by the monitor in the region control blocks.

Figure 3–20 and Table 3–9 show the structure of a region control block. The .ELRG programmed request clears all its fields.

Figure 3–20: Region Control Block

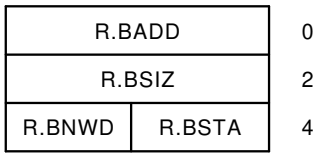


Table 3–9: Region Control Block (.RCBDF)

Offset	Symbol	Modifier	Contents
0	R.BADD	Monitor's .CRRG routine	The starting address of the region, expressed in 32-word units.
2	R.BSIZ	Monitor's .CRRG routine	The size of the region in 32-word units. If this word is 0, this region control block is free.
4	R.BSTA	The monitor at job initiation; the monitor's .CRRG routine clears this byte	Bit 0 (R.STOP) is set by the monitor. Bit 1 (R.SHAR) indicates that the region is a shared global region. Bit 2 (R.EXIT) indicates that the global region should be eliminated when the job exits or aborts. Bit 3 (R.CACH) indicates the local region is attached to a global region with cache-bypass set.
5	R.BNWD	Monitor's .CRRG routine clears this byte; .MAP increments it; .UNMAP decrements it	The number of windows currently mapped to this region.

3.6.3 Global Region Control Block

The global region control block is a 5-word area in RMON. It is set up by the .CRRG programmed request and maintained by the monitor. The global region control block contains the size, starting address, status, count of local region attachments, and name assigned to the global region. By default, RT–11 provides 10_{10} global region control blocks (X$RCBS=10.). There are also 3 permanent region control blocks that are not available to a program.

Figure 3–21 and Table 3–10 describe the global region control block.

Figure 3–21: Global Region Control Block

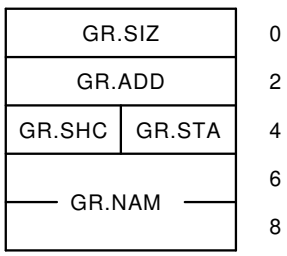


Table 3–10: Global Region Control Block (.GRBDF)

Byte Offset	Symbol	Modifier	Contents
0	GR.SIZ	Monitor's .CRRG routine	The size of the global region expressed in 32-word units. A 0 indicates the global region is not in use.
2	GR.ADD	Monitor's .CRRG routine	The starting physical address of the global region expressed in 32-word units.
4	GR.STA	.RDBBK macro	The global region control block status byte.
5	GR.SHC	.RDBBK macro	A count kept of attachments to the global region. The count is incremented with each successful .CRRG request, decremented with each successful .ELRG request. GR.PRV (Bit 7) = 1, global region private (no new attachments honored until original attached local region is detached.) GR.PRV (Bit 7) = 0, any job can attach to the global region.
6	GR.NAM	.RDBBK macro	2-word name of global region in Radix–50.

3.6.3.1 Global Region Control Block Status Byte (GR.STA)

Table 3–11 describes the global region control block status byte. Bits 0-3 are reserved for future use by Digital.

Table 3–11: Global Region Control Block Status Byte, GR.STA

Bit	Symbol	Pattern	Meaning When Set
7	GR.AGE	200	Enables automatic global elimination. The global region area is returned to the free memory list when the last job using the global region detaches. Count is kept in GR.SHC.
6	GR.PRM	100	The global region is permanent; it is never eliminated.
5	GR.NRF	40	Do not return the global region's allocated memory to the free memory list when the global region is eliminated.
4	GR.CAC	20	Cache-bypass is enabled for the global region.
0-3			Reserved.

3.6.4 Window Definition Block

A window definition block is a seven-word area in your program that contains information about a virtual address window you define. The monitor uses the window definition block to communicate with your program when you issue a .CRAW, .ELAW, .GMCX, or .MAP programmed request. You must set up the window definition block in your program and define its symbolic offset names before you

can create a virtual address window. You must then place a description of the window you need in the window definition block. After you create the window, the monitor returns its identification and some status information to you through the window definition block. Figure 3–22 and Table 3–12 show the structure of a window definition block.

Figure 3–22: Window Definition Block

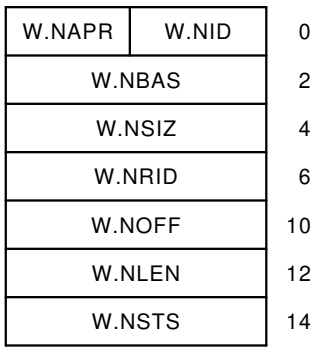


Table 3–12: Window Definition Block (.WDBDF)

Offset	Symbol	Modifier	Contents
0	W.NID	Monitor's .CRAW routine	A unique window identification. Remember that you can always refer to the static window by using 0 as its identification.
1	W.NAPR	.WDBBK macro; monitor's .GMCX routine	The number of the Active Page Register that includes the window's base address. Remember that a window must start on a 4K-word boundary. See Table 3–13 for the correspondence between Active Page Registers and virtual addresses. For privileged jobs, the valid range of values is from 0 to 7. For virtual jobs, the new window must not overlap the static window. You can find the lowest valid value for W.NAPR by issuing a .GMCX request for the static window, converting the high virtual address to an APR value, and incrementing it.
2	W.NBAS	Monitor's .CRAW and .GMCX routines	The base virtual address of this window. This value should indicate the same address as W.NAPR. It is provided as a validity check. Note that it is expressed as a 16-bit address, not in 32_{10}-word units.

Table 3–12 (Cont.): Window Definition Block (.WDBDF)

Offset	Symbol	Modifier	Contents
4	W.NSIZ	.WDBBK macro; monitor's .GMCX routine	The size of this window, expressed in 32-word units.
6	W.NRID	.WDBBK macro; monitor's .GMCX routine	Identification of the region to which this window maps. The .GMCX request returns a 0 if the window is not mapped. Otherwise, it returns the identification of the region to which it is mapped. Note that the value is also 0 if the window is mapped to the static region.
10	W.NOFF	.WDBBK macro; monitor's .GMCX routine	The offset, expressed in 32_{10}-word units, into the region at which to start mapping this window. The .GMCX request clears this word if the window is not mapped; otherwise, it puts the offset value here.
12	W.NLEN	.WDBBK macro; monitor's .MAP and .GMCX routines	The amount of this window to map, expressed in 32-word units. If you put 0 here, .MAP (or.CRAW with WS.MAP set) maps as much of the window as possible. On successful completion of the mapping operation, .MAP puts the actual length it mapped in W.NLEN. If you put a value (other than 0) here, .MAP does not change it. The .GMCX request clears this word if the window is not mapped; otherwise, it puts the actual length mapped here.
14	W.NSTS	.WDBBK macro; monitor's .CRAW, .ELAW, and .GMCX routines	The window status word. The .GMCX request clears this word if the window is not mapped; otherwise, it sets WS.MAP to 1.

Table 3–13: Correspondence Between Active Page Registers and Virtual Addresses

Virtual Address Range	Active Page Register Number
0000000-17776	0
0020000-37776	1
0040000-57776	2
0060000-77776	3
100000-117776	4
120000-137776	5
140000-157776	6

Table 3–13 (Cont.): Correspondence Between Active Page Registers and Virtual Addresses

Virtual Address Range	Active Page Register Number
160000-177776	7

3.6.4.1 Window Status Word (W.NSTS)

The window status word serves a dual purpose. First, it allows the .CRAW request to create a window and map it to a region in one step when you put a value of 1 in bit 8. Second, the window status word allows the monitor to communicate status information to your program. Table 3–14 shows the bits in the window status word and their meaning. Bits 4 through 7 are reserved for future use by Digital.

Table 3–14: Window Status Word, W.NSTS

Bit	Symbol	Pattern	Meaning When Set
0-1	WS.MOD	000003	Processor mode field.
	WS.U	000000	Selected User mode.
	WS.S	000001	Selected Supervisor mode.
	WS.C	000002	Selected current mode.
		000003	Reserved.
2-3	WS.SPA	000014	Address space field.
	WS.DEF	000000	Default; selected I-space (or D-space if I & D address space separation.)
	WS.I	000004	Selected I-space.
	WS.D	000010	Selected D-space.
		000014	Selected both I-space and D-space.
8	WS.MAP	000400	The .CRAW request should also map the new window in addition to creating it. Set this bit in the window definition block by specifying it in the .WDBBK macro. Be sure to load W.NRID before using .CRAW.
9	WS.RO	001000	Access to the addresses mapped by this window is read-only.
10	WS.OVR	002000	Reserved.
11	WS.IDD	004000	The request set WS.I and WS.D and indicated identical action on I-space and D-space windows (the request assumed identical window virtual addresses). The window virtual addresses are different, but the request is processed anyway. Not returned by .CRAW request.

Table 3–14 (Cont.): Window Status Word, W.NSTS

Bit	Symbol	Pattern	Meaning When Set
12	WS.DSI	010000	The request set WS.I and WS.D, indicating action on both I-space and D-space. However, D-space is inactive. I-space is processed and D-space is ignored.
13	WS.ELW	020000	The monitor eliminated one or more windows as a result of the current operation. The .CRAW and .ELAW routines can set this bit.
14	WS.UNM	040000	The monitor unmapped one or more windows as a result of the current operation. The .CRAW and .ELAW routines can set this bit. The .MAP and .UNMAP routines set or clear this bit, as required.
15	WS.CRW	100000	The monitor created this window successfully. The .CRAW routine sets this bit; the .ELAW routine clears it.

3.6.4.2 .WDBDF Macro

Use the .WDBDF macro to define symbols for the window definition block (see the description of .WDBBK in Section 3.6.4.3). It defines the symbolic offset names for the window definition block and the names for the window status word bit patterns. In addition, this macro also defines the length of the window definition block by setting up the following symbol:

```
W.NLGH =: 16
```

Note that the .WDBDF macro does not reserve any space for the window definition block.

The format of the .WDBDF macro is as follows:

.WDBDF

The .WDBDF macro expands as follows:

```
W.NID  =: 0
W.NAPR =: 1
W.NBAS =: 2.
W.NSIZ =: 4.
W.NRID =: 6.
W.NOFF =: 10
W.NLEN =: 12
W.NSTS =: 14
W.NLGH =: 16
WS.CRW =: 100000
WS.UNM =: 40000
WS.ELW =: 20000
WS.DSI =: 10000
WS.IDD =: 4000
WS.OVR =: 2000
WS.RO  =: 1000
WS.MAP =: 400
```

```
WS.SPA =: 14
WS.D   =: 10
WS.I   =: 4
WS.MOD =: 3
WS.U   =: 0
WS.S   =: 1
WS.C   =: 2
```

3.6.4.3 .WDBBK Macro

The .WDBBK macro uses the .WDBDF macro to define symbols for the window definition block. This macro also actually reserves space for it (unlike the .WDBDF macro). The macro permits you to specify enough information about the window to simply create it. Or you can use the optional arguments to provide more information in the window definition block. The extra information allows you to create a window and map it to a region by issuing just the .CRAW programmed request. If you use .WDBBK, you need not use .WDBDF, since .WDBBK automatically invokes .WDBDF.

The format of the .WDBBK macro is as follows:

.WDBBK wnapr,wnsiz[,wnrid,wnoff,wnlen,wnsts]

where:

wnapr is the number of the Active Page Register set that includes the window's base address. Remember that a window must start on a 4K-word boundary. See Table 3–13 for the correspondence between Active Page Registers and virtual addresses. The valid range of values is from 0 through 7.

wnsiz is the size of this window. Express it in 32_{10}-word units.

wnrid is the identification for the region to which this window maps. This argument is optional. It is usually filled in at run time, rather than at assembly time.

wnoff is the offset into the region at which to start mapping this window. Express it in 32_{10}-word units. This argument is optional; supply it if you need to map this window. The default is 0, which means that the window starts mapping at the region's base address.

wnlen is the amount of this window to map. Express it in 32_{10}-word units. This argument is optional; supply it if you need to map this window. The default value is 0, which maps as much of the window as possible.

wnsts is the window status word. This argument is optional; supply it if you need to map this window when you issue the .CRAW request. Set bit 8, called WS.MAP, to cause .CRAW to perform an implied mapping operation.

The example in Figure 3–23 uses the .WDBBK macro to create a window definition block. First, it establishes a convention for expressing K words in units of 32_{10} words. Then, it defines the window definition block, creates the window, and maps the window to a region.

The macro call sets up a window definition block for a window that is 2K words long. The window begins at address 120000, so Active Page Register 5 controls its mapping. The .CRAW request to create this window will also map it to an area in extended memory. The window will map to the region starting 2K words from the beginning of the region, and the .CRAW request will map as much of the window as possible. Note that the program must move the region identification into this block to select the correct region before it issues the .CRAW request.

Figure 3–23: .WDBBK Macro Example

```
        .MCALL  .WDBBK, .RDBBK, .CRRG, .CRAW, .EXIT
        KMMU=   1024./32.               ;1K in 32-word units
START:  .CRRG   #AREA,#RGADR            ;Create a region
        ;       .
        ;       .
        ;       .
        MOV     RGADR+R.GID,WNADR+W.NRID ;Move region
                                        ;ID to window definition
                                        ;block
        .CRAW   #AREA,#WNADR            ;Create a window and map it
        ;       .
        ;       .
        ;       .
        .EXIT                           ;Exit program
RGADR:  .RDBBK  2*KMMU                  ;Region definition block
WNADR:  .WDBBK  5,2*KMMU,2*KMMU,0,0,WS.MAP  ;Window
                                        ;definition block
AREA:   .BLKW   2                       ;EMT area
        .END    START
```

3.6.5 Window Control Block

Each window control block is an eight-word area in the Mapping Context Area (MCA) region, whose contents are maintained by the monitor. A virtual job dedicates one window control block for each User Mode address space to the static window. For a privileged job, one window control block for each User Mode address space is reserved by the monitor and cannot be used by a program. Thus, all jobs can have up to seven dynamic windows for each User Mode address space, whose status is maintained by the monitor in the window control blocks. All eight Window Control Blocks in Supervisor Mode are dynamic. Figure 3–24 and Table 3–15 show the structure of a window control block.

Figure 3–24: Window Control Block

Table 3–15: Window Control Block (.WCBDF)

Offset	Symbol	Modifier	Contents
0	W.BRCB	Monitor's .MAP routine; the .UNMAP request clears it	A pointer to the region control block of the region to which this window is mapped. If the value is 0, the window is not mapped.
2	W.BLVR	Monitor's .CRAW routine	The window's low virtual address limit.
4	W.BHVR	Monitor's .MAP routine.	The window's high virtual address limit.
6	W.BSIZ	Monitor's .CRAW routine; the .ELAW request clears it	The window's size, in 32_{10}-word units. If the value is 0, this window control block is free.
10	W.BOFF	Monitor's .MAP routine	The offset into the region at which this window begins to map, in 32_{10}-word units.
12	W.BFPD	Monitor's .CRAW routine	The low byte of the address of the first (and therefore the User I-space) Page Descriptor Register that affects this window.
13	W.BNPD	Monitor's .MAP routine	The number of Page Descriptor Registers that affect this window.
14	W.BLPD	Monitor's .MAP routine	The contents of the last Page Descriptor Register that affects this window.

3.6.6 Mapping Context Area (MCA) Region

The MCA region resides in extended memory and contains space for the following data structures for each job:

- Region Control Blocks (RCBs)
- Page Address Registers (PARs)
- Page Descriptor Registers (PDRs)
- Window Control Blocks (WCBs)
- Memory Management Register 3 (MMR3)

Relevant mapping information for each job is stored in the MCA region. When the job is swapped out of memory, mapping information is copied from the I/O page and stored in the MCA region. When the job is swapped back into memory, the contents of the MCA region are used by RT–11 to reestablish the job's mapping context.

Arguments you supply for the .CMAP request *value* parameter determine the mapping context for each job. The MCA region stores the entire mapping context for any of the mapped monitors. However, mapping context is kept for only those registers and blocks you enable though the .CMAP request.

When mapping is controlled by the mapping programmed requests listed below, the contents of the APRs are always in alignment with the values from the corresponding data structures in the WCBs. As part of their function, the following requests perform an alignment between the affected WCBs and the corresponding APRs:

- .CMAP
- .CRAW/.ELAW
- .ELRG
- .MAP/.UNMAP
- .MSDS
- .SETTOP (XM type)

See Section 3.9.5 for more information on the page registers.

User mode static windows (window 0) cannot be modified or deactivated.

All Supervisor mode windows can be modified or deactivated; however, note the following about window 0:

- SHANDL should be loaded to initialize Supervisor I-space, beginning with block 0. SHANDL sets up locations 0 through 12 to correctly process completion routines and correctly handle transfers to Supervisor Mode.
- Because any EMT request issued from supervisor mapping must be able to return information to the SYSCOM area (mapped by window 0), do not deactivate S-D window 0 when issuing EMT requests.
- Because completion routines issued from supervisor mapping require access to the I-space window 0, do not deactive S-I window 0 when issuing completion routines. The same restriction applies when issuing the CSM instruction—do not deactivate S-I window 0.

RMON fixed offset $MEMPT contains an offset from the base of RMON to a data structure (.MEMDF) that provides MCA offset and other mapping information. The data structure begins at location symbol CORPTR, just after the RMON fixed offset area:

Label	Off-set	Initial Contents	Meaning
CORPTR::	0	00	Free core pointer (low memory)
CORPTX::	04	<$XMSIZ-$RMON>	Free core offset (high memory).
OFS.MC	06	I.MPTR	From impure area; offset to MCA chunk address.

Label	Off-set	Initial Contents	Meaning
NUM.RG	10	R.GNUM	Number of local RCBs in MCA for this job.
NUM.WN	11	(<SUP$Y*3.+1.)*W.NNUM>	Number of WCBs in MCA for this job.
OFS.RG	12	M.RGN	Offset in MCA to start of job's RCBs.
OFS.WN	14	M.WCB	Offset in MCA to start of job's WCBs.
OFS.AP	16	M.APR	Offset in MCA to start of job's APRs.
M.VHI	20	I.VHI	Offset in impure area to job's virtual high limit.

The MCA region structure and the actual offsets can be seen by invoking .MCADF in the distributed file, SYSTEM.MLB. The following is the general structure of the region:

Symbol	Contents
M.RGN	Base of the MCA region and location of the RCBs. Can contain up to 24_{10} RCBs.
M.APR	Beginning of APR (Active Page Register) section. APRs consist of PDRs (Page Descriptor Registers) and PARs (Page Address Registers). Individual APRs can be mapped by specifying the CM.DUS and CM.PAR arguments in the .CMAP request. Alternately, the .MSDS request can map individual APRs after general support has been established by the .CMAP request.
M.PDUI	User mode/Instruction space PDRs (8 words). These PDRs are present in all mapped systems and are always context-switched.
M.PDUD	User mode/Data space PDRs (8 words). These PDRs are context-switched only when CM.UID has been specified in a .CMAP request. With single-mapped monitors, this space is undefined but reserved.
M.PAUI	User mode/Instruction space PARs (8 words). These PARs are present in all mapped systems and are always context-switched.

Symbol	Contents
M.PAUD	User mode/Data space PARs (8 words). These PARs are context-switched only when CM.UID has been specified in a .CMAP request. With single-mapped monitors, this space is undefined but reserved.
M.PDSI	Supervisor mode/Instruction space PDRs (8 words). These PDRs are context-switched only when CM.JAS has been specified in a .CMAP request. Fully-mapped monitors only.
M.PDSD	Supervisor mode/Data space PDRs (8 words). These PDRs are context-switched only when CM.JAS and CM.SID have been specified in a .CMAP request. Fully-mapped monitors only.
M.PASI	Supervisor mode/Instruction space PARs (8 words). These PARs are context-switched only when CM.JAS has been specified in a .CMAP request. Fully-mapped monitors only.
M.PASD	Supervisor mode/Data space PARs (8 words). These PARs are context-switched only when CM.JAS and CM.SID have been specified in a .CMAP request. Fully-mapped monitors only.
M.WCB	Beginning of the WCB section. Up to 8 WCBs are supported for each processor mode/address space. WCBs are maintained in each processor mode/address space configuration in alignment with the corresponding APR (the APR that the WCB describes).
W.WCUI	User mode/Instruction space WCBs. User mode/Instruction space WCBs are active for all mapped environments.
W.WCUD	User mode/Data space WCBs.
W.WCSI	Supervisor mode/Instruction space WCBs.
W.WCSD	Supervisor mode/Data space WCBs.
M.MMR3	Memory mapping register 3. Do not modify the contents of M.MMR3. The relevant bits for mapping context are:

Bit	Meaning When Set
0	Separate User mode I-D address space.
1	Separate Supervisor mode I-D address space.
3	The CSM instruction is enabled.

3.6.7 I/O Queue Element

The I/O queue element in a mapped memory system is ten words long, rather than seven words long as it is in an unmapped system. Section 2.3.2 describes the I/O queue element in detail.

3.6.8 Free Memory List

The monitor maintains a data structure called the free memory list, which it uses to allocate areas of extended memory. The list consists of a table of 10_{10} doublewords. The address of the top of the table is $XMSIZ, and the table is located in p-sect XMSUBS. The high-order word of each word pair indicates the size of an available area in extended memory, expressed as a number of 32_{10}-word units. The low-order word of the pair contains the address of the area, divided by 100_8. A value of -1 ends the table.

At the start of bootstrap time, the table contains only one entry. The high-order word of the pair contains the total amount of extended memory. The low-order word contains the value 1600. When a job requests an extended memory region, the monitor searches through the table for an area large enough to meet the request. It returns the area in extended memory that meets the size requirement and has the lowest starting address. During the bootstrap process, handlers often create regions in memory. As each region is created, the monitor reduces the amount of memory in the first doubleword of the free memory list, and adjusts its starting address.

The other nine words of the free memory list are used when jobs return areas of extended memory to the available pool. In a very active system, the extended memory area can become quite fragmented.

3.7 Flow of Control Within Each Programmed Request

This section summarizes the activities that take place internally for each programmed request your program can issue. Consult the *RT–11 System Macro Library Manual* for the detailed syntax of each request.

3.7.1 Defining the Memory Mapping Context: .CMAP, .GCMAP, .MSDS

Use the .CMAP, .GCMAP, and .MSDS programmed requests to define the memory mapping context.

.CMAP

Issue the .CMAP programmed request with fully-mapped (ZB and ZM) monitors to enable or disable processor modes and address spaces for the program.

The symbols you specify for the .CMAP *value* parameter are written to the I.CMAP word in the impure area, and are used to control the Mapping Context Area (MCA) region in extended memory. The MCA contains the current mapping context for the program.

The symbols supported by .CMAP are located in Table 2–32. The structure of the MCA is located in Section 3.6.6.

The following mapping context exists at job creation:

- User mode nonseparated I-D-space.
- Supervisor mode separated I-D space.
- No Supervisor mode context.
- All Supervisor mode D-space locked to User I-space.

That mapping context is symbolically (as would be specified to .CMAP):

```
CM.UII!CM.SID!CM.NOS!CM.DUS
```

Supervisor mode and User mode address spaces can be locked according to the arguments you supply to .CMAP (or .MSDS). Precisely which address spaces can locked is determined by the following rules. In the rules, *S* and *U* are Supervisor and User mode, respectively, and *I* and *D* are instruction and data address spaces, respectively. Therefore, for example, *SI* means support for Supervisor mode, Instruction address space.

- If SI-SD and UI-UD, then SD locked to UD.
- IF SI-SD and UI, then SD locked to UI.
- If SI and UI-UD, then SI locked to UD.
- If SI and UI, then SI locked to UI.

Issuing the .CMAP request has the following effects on WCBs and APRs:

- If .CMAP enables or disables User mode D-space (regardless of its previous state), all existing User D-space windows are eliminated. If .CMAP enables User mode D-space (even if currently enabled), the User I-space windows and APRs are duplicated for User D-space.
- If .CMAP enables or disables Supervisor mode (regardless of its previous state), all existing Supervisor I-space and D-space windows are eliminated.

  If .CMAP enables or disables Supervisor mode D-space (regardless of its previous state), all existing Supervisor D-space windows are eliminated.

  If .CMAP enables Supervisor mode D-space (even if currently enabled), the Supervisor I-space windows and APRs are duplicated for Supervisor D-space. Further, the Supervisor mode stack pointer is initialized to the same value as the User mode stack pointer.

Whenever .CMAP is issued, the previous value in I.CMAP is returned in R0.

.GCMAP

Although typically defined and then left static, mapping context can be changed by the program dynamically. The program can issue a .GCMAP request to read the contents of I.CMAP and determine current mapping context at any time. The contents of I.CMAP are returned in R0.

.MSDS

Use the .MSDS to specify which Supervisor mode APRs are locked to their corresponding User mode APRs. You can determine which Supervisor mode APRs were previously locked by examining the low byte of I.CMAP, which is returned in R0.

3.7.2 Creating a Local Region: .CRRG

Issue the .CRRG programmed request to create a local region in physical address space.

The monitor's .CRRG routine first checks R.GSIZ in the region definition block to make sure that you have requested a region with a valid size. (The size must be nonzero.) If the size is invalid, the request returns with error code 10 in byte 52.

Next, the routine looks for a free region control block. The request returns with error code 6 in $ERRBY (byte 52) if no region control blocks are free.

The routine attempts to allocate the appropriate amount of memory for the region, based on the amount you specified in the programmed request. To get the most memory possible, ask for 2044K words. The routine scans the free memory list for a region with the correct size. The request returns with error code 7 in $ERRBY if it cannot allocate a region with the size you requested. In addition, R0 contains the largest amount of memory available. Issue the .CRRG request again for this amount of memory. If this second request fails in a multi-job mapped system, it means that some other job in the system just acquired some of the memory. Continue to reissue the .CRRG request with the new value from R0 until you finally obtain a region.

The request succeeds when the monitor allocates the region. The routine puts the region identification into R.GID in the region definition block. It sets RS.CRR in the region status word; it clears R.BSTA and R.BNWD in the region control block, and it puts values into R.BADD and R.BSIZ, which are also located in the region control block. The memory obtained is then removed from the monitor's free memory list and reserved for your job.

3.7.3 Creating and Attaching to a Global Region

Issue the .CRRG programmed request to create a global region in physical address space and attach to it.

There is no limit to the size of a global region other than available physical (extended) memory.

When a program creates and attaches to a global region, the program has sole possession of the global region. This exclusive use is indicated by the setting of GR.PRV (bit 7) in the global region's control block (GR.SHC).

To make the global region available for attachment by other programs in a multi-job mapped system, the program must detach from the global region with a .ELRG request. Detaching from a global region clears bit 7 of the control block GR.SHC byte. A program wishing to create and attach to a global region and make the global region available to other programs must execute a .ELRG request and then execute a .CRRG request to reattach to the global region.

In a single job mapped system, global regions are shared sequentially. If programs are to share a global region in an XB system, the job that creates the global region must insure that the region is not automatically eliminated when the job exits. Subsequent jobs can then attach to the job and use it.

3.7.4 Attaching to an Existing Global Region

The procedure for attaching to a global region is the same as the procedure for creating a global region, except that you should not specify the RS.CGR symbol in the status argument of the .RDBBK macro. The following shows the form of the .RDBBK macro for attaching to the global region created in the example shown in Section 3.13.

```
REGION: .RDBBK  4096./32.,<RS.GBL>,NAME=MYDATA
```

When a program attaches to a global region, RT–11 creates a local region within the global region. The area requested for the attached local region is taken from the memory allocated to the global region, not from the free memory list. The local region is attached to the global region at the base address of the global region. RS.CRR (bit 15 of status word R.GSTS in the global region definition block) is set.

As described in Section 3.7.3, a program cannot attach to a global region unless the program that created the global region has detached from it.

Specifying 0 for R.GSIZ (byte offset 2 of the region definition block) attaches a program to all memory allocated to an existing global region. The global region must have been previously created by another program or at boot time. Attempts to attach in this manner to a local region, or to a global region not previously created, returns error code 10_8 and RS.CRR is cleared.

Attempting to attach to a local region larger than the global region returns octal error code 7. Attempting to attach to a nonexistent global region returns octal error code 12. Either case clears RS.CRR.

Attaching a local region to a global region increments the count of attachments for that global region kept in GR.SHC (byte 5 in the global region control block). The reference count in GR.SHC controls automatic global elimination (see Section 3.7.12).

3.7.5 Detaching from a Global Region

Issue the .ELRG macro to detach a local region from a global region, in the same manner as you use it to eliminate a local region. Memory allocated to the local region within the global region is retained by the global region and is not returned to the free memory list.

The following example illustrates detaching the local region attached in the example shown in Section 3.13.

```
        .ELRG   #AREA,#REGION
        .
        .
        .
REGION: .RDBBK  4096./32.,<RS.GBL>,NAME=MYDATA
```

Detaching a local region from a global region decrements the count of attachments for that global region kept in GR.SHC (byte 5 in the global region control block). In a multi-job mapped system, the reference count in GR.SHC controls automatic global elimination (see Section 3.7.12).

3.7.6 Creating a Window: .CRAW

Issue the .CRAW programmed request to create a virtual address window.

Prior to performing its main functions, the .CRAW programmed request does the following initial procedure:

- The routine clears the following bits in the window status word: WS.CRW, WS.UNM, WS.ELW, WS.DSI, and WS.IDD.
- If D-space of the requested mode is inactive and WS.I is set in the window status word, the routine sets WS.DSI in the window status word.
- If the mode requested is not active, or if WS.D is set and WS.I is cleared in the window status word and D-space is inactive in the requested mode, the routine returns error code 17_8.
- If WS.D and WS.I are both set in the window status word, and D-space is inactive for the requested mode, the routine sets WS.DSI in the window status word and processes the request for only I-space.

First, the monitor's .CRAW routine checks W.NAPR in the window definition block for a valid value. The request returns with error code 0 in $ERRBY (byte 52) if the number of the Active Page Register set is invalid for any reason.

Next, the routine shifts W.NAPR to set up the window's base address in W.NBAS, which is also located in the window definition block.

The routine then checks W.NSIZ in the window definition block to make sure that you requested a valid size for the window (the window cannot exceed the 32K-word boundary). If there is any problem with the size, the request returns with error code 0 in $ERRBY.

The next check is to see if the new window will overlap with an existing window. If the job is a virtual or completely virtual job and the new window overlaps with the static window, the request returns with error code 0. In all other situations where the new window overlaps an existing window, the routine eliminates the existing window. If the existing window is mapped, the routine unmaps it. The .CRAW routine sets WS.ELW in the window status word if it eliminates a window to create the new one. It sets WS.UNM if it also unmaps a window as it eliminates it.

Next, the routine looks for an available window control block. The request returns with error code 1 if there are no free window control blocks.

The request succeeds when the monitor modifies the appropriate data structures. It puts values in W.BSIZ, W.BLVR, and W.BFPD in the window control block; it puts the window identification in W.NID in the window definition block, and it sets WS.CRW in the window status word.

If WS.MAP in the window status word was set when you issued the .CRAW request, the routine now maps the window to the region whose identification is stored in the window definition block. To do this, the routine follows the steps outlined for the .MAP programmed request's main functions.

3.7.7 Mapping a window to a Region: .MAP

Issue the .MAP programmed request to map a virtual address window to a physical address region. The window definition block must contain the identification of the region to which the window will map.

Prior to performing its main functions, the .MAP programmed request does the following initial procedure:

- The routine clears the following bits in the window status word: WS.CRW, WS.UNM, WS.ELW, WS.DSI, and WS.IDD.
- If D-space of the requested mode is inactive and WS.I is set in the window status word, the routine sets WS.DSI in the window status word.
- If the mode requested is not active, or if WS.D is set and WS.I is cleared in the window status word and D-space is inactive in the requested mode, the routine returns error code 17_8.
- If WS.D and WS.I are both set in the window status word, and D-space is inactive for the requested mode, the routine sets WS.DSI in the window status word and processes the request for only I-space.
- If both I-space and D-space operations are requested and D-space is active, the routine verifies that the windows involved in the operation are identical. If they are not, the routine sets WS.IDD in the window status word.

First, the monitor's .MAP routine finds the window control block that corresponds to the window you specify in the request. It checks W.NID to do this, and returns with error code 3 if the value is 0 and the window is not Supervisor mode or otherwise not valid.

Next, the routine finds the region control block for the region to which this window will map. The request returns with error code 2 if the region identification is invalid for any reason.

The routine looks at the offset into the region at which the window is to begin mapping. This value is contained in W.NOFF in the window definition block. If the offset is beyond the end of the region, the request returns with error code 4.

The routine checks the length of the window it is to map. This value is contained in W.NLEN in the window definition block. If the value is 0, the routine picks up the size of the region from the offset value to the end of the region. If this amount of memory is bigger than the window, the routine reduces the amount until it equals the window size, which it stores in W.NLEN. Note, that if you put 0 into W.NLEN, the value that is there after the .MAP request executes is not 0, but is instead the actual length of the window that was mapped. When WS.D and WS.I are both set in the window status word, W.NLEN is returned with I-space information. If WS.IDD

is cleared in the window status word, the value returned in W.NLEN reflects D-space information as well. If WS.IDD is set, the corresponding value in W.NLEN for D-space is indeterminate.

If the value of W.NLEN is not 0 at the start of the .MAP routine, it indicates the explicit length of the window to map. If the value is larger than the window size, or if the window would extend beyond the bounds of the region, the request returns with error code 4.

The routine increments R.BNWD in the region control block, which maintains a count of the number of windows mapped to this region.

If this window is already mapped elsewhere, this routine unmaps it and sets WS.UNM in the window status word; otherwise, this routine clears WS.UNM.

The routine next loads the User mode Active Page Register set with the correct values to map this window to this region.

Finally, the routine updates the window control block values W.BRCB, W.BHVR, W.BOFF, W.BNPD, and W.BLPD.

3.7.8 Getting the Mapping Status: .GMCX

Issue the .GMCX programmed request to obtain the current mapping status of a particular virtual address window.

Prior to performing its main functions, the .GMCX programmed request does the following initial procedure:

- The routine clears the following bits in the window status word: WS.CRW, WS.UNM, WS.ELW, WS.DSI, and WS.IDD.
- If D-space of the requested mode is inactive and WS.I is set in the window status word, the routine sets WS.DSI in the window status word.
- If the mode requested is not active, or if WS.D is set and WS.I is cleared in the window status word and D-space is inactive in the requested mode, the routine returns error code 17_8.
- If WS.D and WS.I are both set in the window status word, and D-space is inactive for the requested mode, the routine sets WS.DSI in the window status word and processes the request for only I-space.
- If both I-space and D-space operations are requested and D-space is active, the routine verifies that the windows involved in the operation are identical. If they are not, the routine sets WS.IDD in the window status word.

If both WS.I and WS.D are requested, the routine only returns information for I-space.

First, the .GMCX monitor routine looks at the corresponding window control block for this window. If you specify a window whose identification is 0, you obtain the status of the static window for a virtual job. User mode (either address space) window 0 is reserved in a privileged job. If there is any problem with the window, the request returns with error code 3.

The routine sets W.NAPR in the window definition block to be equal to the top three bits of W.BLVR in the window control block. This sets up the starting Active Page Register set number.

Next, the routine puts values into W.NBAS, W.NSIZ, and W.NRID in the window definition block.

If the window is not currently mapped, the routine clears W.NOFF and W.NLEN in the window definition block. It also clears all bits in the window status word, except for WS.RO, WS.SPA, and WS.MOD, which it leaves unmodified.

If the window is mapped, the routine puts the offset into the region in W.NOFF, puts the length of the window in W.NLEN, and sets the bit WS.MAP in the window status word. If the window is mapped read-only, the routine also sets bit WS.RO in the window status word; otherwise it clears WS.RO.

3.7.9 Unmapping a Window: .UNMAP

Issue the .UNMAP programmed request to explicitly unmap a window from a region.

Prior to performing its main functions, the .UNMAP programmed request does the following initial procedure:

- The routine clears the following bits in the window status word: WS.CRW, WS.UNM, WS.ELW, WS.DSI, and WS.IDD.
- If D-space of the requested mode is inactive and WS.I is set in the window status word, the routine sets WS.DSI in the window status word.
- If the mode requested is not active, or if WS.D is set and WS.I is cleared in the window status word and D-space is inactive in the requested mode, the routine returns error code 17_8.
- If WS.D and WS.I are both set in the window status word, and D-space is inactive for the requested mode, the routine sets WS.DSI in the window status word and processes the request for only I-space.
- If both I-space and D-space operations are requested and D-space is active, the routine verifies that the windows involved in the operation are identical. If they are not, the routine sets WS.IDD in the window status word.

First, the monitor's .UNMAP routine finds the appropriate window control block. It checks W.NID in the window definition block. If the value is 0 and the window is not a Supervisor mode window, or if it is invalid for any reason, the request returns with error code 3. If the window is not currrently mapped, the request returns with error code 5.

To unmap the window, the routine modifies the appropriate data structures. It clears W.BRCB in the window control block and decrements R.BNWD in the region control block.

If the job is virtual or completely virtual, the routine clears the Page Descriptor Registers that correspond to this window so that your program can no longer reference the virtual addresses in this window.

If the job is privileged, the monitor copies the kernel Page Descriptor Register (PDR) values into the PDR for the specified mode, so that the mapping defaults to that of kernel mode.

Finally, the routine sets WS.UNM in the window status word.

3.7.10 Eliminating a Local Region: .ELRG

Issue the .ELRG programmed request to eliminate a physical address region.

First, the monitor's .ELRG routine checks to see if the region identification you specified is 0. In a virtual job, a region identification of 0 indicates the static region, which you cannot eliminate. In a privileged job, region 0 is reserved. In either case, the request returns with error code 2.

Next, the routine looks for the corresponding region control block for this region. If the region identification is invalid for any reason, the request returns with error code 2.

Then, the routine clears RS.CRR and RS.UNM in the region status word. If there are any windows mapped to this region, the routine unmaps them and sets RS.UNM.

The routine deallocates the region by returning its physical address space to the monitor's list of free memory.

Finally, the routine clears the region control block.

3.7.11 Eliminating a Global Region

Issue the .ELRG macro to eliminate a global region. Specify the status argument RS.EGR in the .RDBBK macro.

The following example eliminates the global region created by the example in Section 3.13.

```
        .ELRG    #AREA,#REGION
          .
          .
          .
REGION: .RDBBK  4096./32.,<RS.GBL!RS.EGR>,NAME=MYDATA
```

Eliminating a global region returns the memory allocated to the global region to the free memory list.

Observe the following when you eliminate a global region:

- Your program must be attached to a global region to eliminate it.
- The same global region definition block used to create a global region is normally used to eliminate it.
- Permanent global regions cannot be eliminated.
- In a multi-job mapped system, attempting to eliminate a global region that is in use by another job returns error code 14_8. The global region is not eliminated, but the program requesting the elimination is detached.

3.7.12 Automatic Global Region Elimination

When you create a global region for use by a number of programs in a multi-job mapped system, you can use the automatic global region elimination feature (AGE) to automatically eliminate the global region when the last program has finished using it.

Specify the .RDBBK status argument RS.AGE when you create the global region. As explained in Section 3.7.3, a program that has created a global region has sole possession of that region (GR.PRV (bit 7) of GR.SHC is set). To make the global region available to other programs, the program must clear that bit by detaching with a .ELRG request. Automatic global elimination is turned on when the next program attaches to the global region.

As programs attach and detach local regions, a reference count is incremented and decremented in GR.SHC (byte 5) in the global region control block. AGE tracks the value in GR.SHC. When the value in GR.SHC is 0, AGE automatically eliminates the global region. The allocated memory is returned to the free memory list.

3.7.13 Eliminating a Window: .ELAW

Issue this programmed request to eliminate a virtual address window.

Prior to performing its main functions, the .ELAW programmed request does the following initial procedure:

- The routine clears the following bits in the window status word: WS.CRW, WS.UNM, WS.ELW, WS.DSI, and WS.IDD.
- If D-space of the requested mode is inactive and WS.I is set in the window status word, the routine sets WS.DSI in the window status word.
- If the mode requested is not active, or if WS.D is set and WS.I is cleared in the window status word and D-space is inactive in the requested mode, the routine returns error code 17_8.
- If WS.D and WS.I are both set in the window status word, and D-space is inactive for the requested mode, the routine sets WS.DSI in the window status word and processes the request for only I-space.
- If both I-space and D-space operations are requested and D-space is active, the routine verifies that the windows involved in the operation are identical. If they are not, the routine sets WS.IDD in the window status word.

First, the monitor's .ELAW routine finds the appropriate window control block. It checks W.NID in the window definition block. If the value is 0 and the window is not a Supervisor mode window, or if it is invalid for any reason, the request returns with error code 3.

If the window was mapped, the routine unmaps it by modifying the appropriate as follows:

- It clears W.BRCB in the window control block, and decrements R.BNWD in the region control block.

- If the job is virtual or completely virtual, the routine clears the Page Descriptor Registers that correspond to this window so that your program can no longer reference the virtual addresses in this window.
- If the job is privileged, the monitor copies the kernel Page Descriptor Register values into the user Page Descriptor Registers so that the mapping defaults to that of kernel mode.
- The routine sets WS.UNM in the window status word.

Finally, the routine clears W.BSIZ in the window control block and sets bit WS.ELW in the window status word.

3.8 Typical Extended Memory Applications

The following sections assume you understand the fundamental concepts of mapped systems; they should help you see how to use extended memory. Some arrangements are suggested that may suit your own particular situation. As you read, keep in mind what benefits you want from an extended memory system. In other words, why do you want to use it?

3.8.1 Completely Virtual Environment (VRUN)

Probably the easist way to take advantage of extended memory is to run your program in the completely virtual environment. See Section 3.2.3 for information on the completely virtual environment.

3.8.2 Minimizing Low-Memory Usage

Make all the programs virtual jobs, unless they really need access to the interrupt vectors in the SYSCOM area. Instead of using interrupt service routines in your program, consider writing a device handler. Special functions requests allow a great deal of flexibility in writing special handlers for unusual devices.

Virtual jobs can access the I/O page by attaching to the IOPAGE global region and mapping it. Completely virtual jobs can automatically map PAR7 to the I/O page by setting IOPAG$ in the file's $JSX word. Low memory can be accessed by virtual and completely virtual jobs by attaching to the KERNEL global region, mapping it, and using .PEEK, .POKE, .GVAL, and .PVAL requests.

The low 28K words of memory fill up rapidly with RMON, device handlers, the USR, a foreground job, one or more system jobs, and a background job. To optimize use of this space and relieve the congestion, make the root segments of the foreground, system, and background jobs (if they are overlaid) as small as possible. Segment the programs and put the overlays into extended memory instead of using disk overlays.

The root segment can be minimal in size. All you need put there are queue elements, channels, interrupt service routines (if any - there can be none in virtual jobs), and a JMP instruction to the first overlay. The overlay segments can be permanently resident in extended memory to speed up execution.

A simple and effective technique to minimize the amount of low memory required by a program is to use the XHANDL overlay pseudohandler. XHANDL creates a

small root in low memory and overlays the rest of the program into a single region in extended memory. XHANDL is described in the *RT–11 System Utilities Manual*.

You can use the linker's /V option to put multiple overlay segments into extended memory. KMON creates a region at run time, using information in the overlay handler and tables. The overlay handler creates and maps windows. Figure 3–25 shows a simple virtual background program that uses extended memory overlays. You can find detailed information on the /V option and extended memory overlays in the *RT–11 System Utilities Manual*.

Figure 3–25: Virtual Background Job with Extended Memory Overlays

3.8.3 Large Buffers or Arrays in Extended Memory

In order to put a large buffer or array into extended memory, you first create a region large enough to accommodate the array. Next, decide how much virtual address space your program can commit to accessing the array and create a virtual address window of that size. Then, simply write a subroutine that translates references to the array into instructions to remap the window into the correct part of the region. Figure 3–26 illustrates this situation. (The extended memory feature of the .SETTOP programmed request can create an extended memory buffer automatically. See Section 3.8.5 for information.)

Figure 3–26: Virtual Background Job with an Array in Extended Memory

3.8.4 Multi-User Program

An extended memory system is ideal for implementing a multi-user application. For example, you could develop a language interpreter that several programmers could use simultaneously. To implement this application, separate your program into two sections: a pure code section that contains the interpreter, and a separate read/write work area for each user. Select part of your virtual address space to be the user scratch area, and create a window of that size. Next, decide how many users you want and create a region equal to the number of users times the size of the window. The interpreter can change user context by remapping the window. Figure 3–27 shows a multi-user program.

Figure 3–27: Multi-User Virtual Background Program

Your multi-user program can use extended memory overlays. In this case, use one region for the overlays and one for the work areas.

3.8.5 Work Space in Extended Memory

Another application for you to consider is putting a work area into extended memory instead of writing it to disk.

Consider how jobs in an unmapped system obtain the most space possible for dynamic buffering. A background job gets extra space by issuing a .SETTOP programmed request. It can obtain the space above the job image up to the top of the USR. To obtain extra space for a foreground job, you must allocate it with the FRUN/BUFFER:n command. Once the space is reserved by FRUN, the program can determine its size and claim it with a .SETTOP programmed request. In both cases, the extra space is within the 28K words of low memory.

In mapped systems, extra space can be allocated from the physical space either above or below the 28K-word boundary. This feature can make jobs runnable that require too much memory for an unmapped RT–11 system. The ability to allocate extra space is most useful to virtual jobs because they can obtain space up to virtual address 177776 (32K words) by using the extended memory feature of the .SETTOP programmed request, hereafter referred to as ''XM .SETTOP'' or ''The XM features of .SETTOP''. All the memory obtained by .SETTOP is in extended memory; virtual foreground jobs do not require the FRUN/BUFFER:n command to allocate extra space.

3.8.5.1 Enabling the XM Feature of the .SETTOP Programmed Request

There are two ways to enable the XM feature of the .SETTOP programmed request:

- Use extended memory overlays. Using the linker /V option to create extended memory overlays automatically enables the XM .SETTOP programmed request. The linker /V option also enables the extended memory feature of the .LIMIT directive (see Section 3.8.5.4), links the extended memory overlay handler (VHANDL) into your job image, and establishes an extended memory overlay structure. You use the /V option by issuing the LINK/PROMPT monitor command, and then specifying /V on a subsequent command line.
- If your program has no overlays, or if it has only low memory overlays that you create with the linker /O option, you enable the XM feature of the .SETTOP programmed request by using the LINK command with the /XM option. The /XM option enables the XM .SETTOP programmed request and the XM .LIMIT directive. It does not link the extended memory overlay handler into your job image, nor does it establish an extended memory overlay structure for your program.

For all programs, the .LIMIT directive returns as its high value the next available location for the job. The extra space your program obtains with .SETTOP in an extended memory system always begins at the address returned as the high value from the .LIMIT directive. This is true for all programs, whether or not they enable the XM feature of the .SETTOP programmed request.

Section 3.8.5.3 describes how .SETTOP works when you execute a program in an extended memory environment without enabling the XM feature of .SETTOP.

Section 3.8.5.4 shows how the XM feature of .SETTOP works after you enable it at link time; it also describes the XM feature of the .LIMIT directive.

3.8.5.2 Program and Virtual High Limits and the Next Free Address

To understand XM .SETTOP, it is important that you understand the differences between the **program high limit**, the **virtual high limit**, and the **next free address**. Figure 3–28 shows a program's virtual address space. This program has both low memory overlays created with the /O linker option, and extended memory overlays created with the /V linker option. The **program high limit** is the highest virtual address used by the program's root segment and its low memory (/O) overlay regions, if any exist. The **virtual high limit** is the highest virtual address used by the extended memory (/V) overlay regions, rounded up to a 32_{10}-word boundary, minus 2. (In octal, the low-order two digits of this address is always 76.) This is the value that prints on the link map as *nnnnnn*, as the following example shows:

```
Virtual high address = nnnnnn = ddddd. words, next free address = mmmmmm
```

The linker has to calculate the value of the *next free address*. For a job that enables the XM feature of .SETTOP, it rounds up the virtual high limit to the next 4K-word boundary. The next free address, then, is the last word of the virtual address space encompassed by the highest Page Address Register used by the job, plus 2. It is always on a 4K-word boundary. (The next free address is always a multiple of 20000_8.)

As an example, consider a job with extended memory overlays whose virtual high limit is 55076. Its next free address calculated by the linker is 60000, or the start of the next 4K words of virtual address space. This is the value that prints on the link map as the ''next free address''. The following example shows the values in our example situation:

```
Virtual high address = 055076 = ddddd. words, next free address = 060000
```

Of course, if a program has no extended memory overlays, it does not have a virtual high limit, and its program high limit is not rounded up. The link map for programs without overlays and for programs whose overlays were created solely by the /O option prints the program high limit as *mmmmmm*, as the following example shows. (The following line prints on all link maps, whether or not extended memory is present.)

```
Transfer address = nnnnnn, High limit = mmmmmm = ddddd. words
```

3.8.5.3 Non-XM .SETTOP

If you do not enable the XM .SETTOP feature through the linker, using .SETTOP in an extended memory program has only limited value.

For a privileged job that does not alter the default mapping, .SETTOP works the way it does in an unmapped system. If a privileged job creates a virtual address window and maps it to an extended memory region, the program high limit is not affected by

Figure 3–28: Program and Virtual High Limits, and the Next Free Address

the mapping. The value returned by .SETTOP still represents the highest address available to the program in the low 28K words of memory.

When the monitor performs address checking for programmed requests, it looks first to see if the address (of an argument block, a data buffer, and so on) is entirely and contiguously mapped by the job. If it is not, the monitor checks to see if the address is within the job's low-memory area. If the address fails both these checks, a monitor error results and the job aborts.

If the job is virtual, the program high limit at load time is set to the highest virtual address used by the root segment and any low-memory (/O) overlays. If your job performs its own mapping operations, they do not affect the program high limit as far as .SETTOP is concerned. So, the .SETTOP request is meaningless to these virtual jobs. The non-XM .SETTOP request deals exclusively with the low 28K words of memory. The virtual job is prevented from accessing memory outside itself (because it is not mapped to any memory but its own dedicated physical space), so issuing a .SETTOP request in a virtual job without the LINK/XM command or the linker /V option does not obtain any extra memory. The value returned can be used by the virtual job to do its own mapping of the area available and then use it.

If the job is completely virtual, .SETTOP simply returns the value requested, subject to an overriding maximum .SETTOP value. (See Section 3.8.5.6.) This is because all of a completely virtual job's virtual address space is preallocated in extended memory and is mapped by the job loader. Thus, in a completely virtual job, there are no low memory or mapping boundary impacts on .SETTOP that must be accounted for.

When the monitor performs address checking for a (completely) virtual job, it ignores the program limits and simply checks to see that the virtual address block is entirely and contiguously mapping. If it is not, a *?MON-F-Inv addr <address>* error results. When the monitor performs address checking for a virtual job, it ignores the program limits and simply checks to see that the virtual address is currently mapped. If the address is not mapped, an invalid address error results.

3.8.5.4 XM .SETTOP

When you enable the XM feature of .SETTOP, as Section 3.8.5.1 describes, .SETTOP becomes valuable to privileged and virtual jobs alike, although its value to privileged jobs is limited.

For virtual jobs, not only does .SETTOP obtain virtual address space above the virtual high limit starting at the program's next free address, but it also automatically maps the extra space to physical space. As a result, a job in an extended memory environment can issue a .SETTOP programmed request and obtain more usable virtual address space without concern for the details of managing extended memory.

For completely virtual jobs with separated I & D address spaces, XM .SETTOP obtains only Data address space; you must explicitly provide any additional Instruction space by creating extended memory regions and mapping to them. (The monitor provides Instruction address space for the root, low memory (/O), and extended memory (/V) overlays.)

For privileged jobs, XM .SETTOP functions the way non-XM .SETTOP does, with the following exception: in privileged jobs, the XM .SETTOP request uses the new XM .LIMIT high value as the next free address, thus always returning the start of the buffer on a 4K-word boundary. A .SETTOP to any address below this 4K-word boundary is not permitted.

For both privileged and virtual programs, the linker puts two words of information into locations 0 and 2 of the job image file. Location 0 contains the Radix–50 code for *VIR*. Location 2 contains the value of the *next free address minus 2*, which can be significantly different from the *virtual high limit*.

.LIMIT Directive

In mapped jobs without the XM feature of .SETTOP, the .LIMIT MACRO directive returns two values to your program. These values are:

- The lowest virtual address used by the program (usually 0)
- The program high limit + 2 (for example, 1644 + 2, or 1646)

In mapped jobs that enable the XM feature of .SETTOP, .LIMIT returns a significantly different value:

- The lowest virtual address used by the program (usually 0)
- The next free address (always on a 4K-word boundary), which is usually not equal to the program high limit + 2.

Gaps in Virtual Address Space

The linker always starts each extended memory (/V) overlay region at a 4K-word boundary in your program's virtual address space. This restriction results from hardware requirements. Because of this, there can be a gap between the program high limit and the start of the virtual overlay region. Your program causes an error if it attempts to reference the virtual addresses within this gap. Similarly, any extra virtual address space that XM .SETTOP obtains for your program also starts on a 4K-word boundary. This means that a gap can exist between your program's virtual high limit and the start of the extra space. Your program cannot reference the addresses within this gap. Figure 3–29 illustrates a typical program with both low memory (/O) and extended memory (/V) overlays.

3.8.5.5 XM .SETTOP and Privileged Jobs

When a privileged job issues a .SETTOP request, if the next free address is above the base of the USR, the program is already using the virtual address space above the start of the monitor. Since there is no free memory that can be mapped starting at the program's next free address, the monitor cannot obtain any more space for this program. Thus, a privileged job can never obtain space above SYSLOW, the base of the USR. The .SETTOP request returns the value of the next free address, minus 2, to $USRTO (location 50) in your program and to R0. This is the highest usable address.

If there is memory available, the monitor tries to obtain it, basing the size of the area on the argument you specify with .SETTOP. The memory is always within the low 28K words. A privileged job can never obtain an amount of virtual address space less than its own next free address, minus 2. In addition, the next free address obtained with XM .SETTOP is always on a 4K-word boundary, and the job cannot issue a .SETTOP for any address below that. Therefore, the job loses the space between its last used address and the next 4K-word boundary.

Privileged Background Jobs

Figure 3–30 shows a privileged background job and all its limits. In a mapped single-job system, or in a mapped multi-job system when no foreground job is present in memory, the background job can obtain some space through .SETTOP. There is often still space available in a mapped multi-job system even when a foreground program is present.

Figure 3–29: Gaps in Virtual Address Space

Privileged Foreground Jobs

Since foreground jobs load into memory just below the last device handler and above the USR, there is no extra space available for them through a .SETTOP request.

Privileged foreground jobs are prohibited from using extended memory overlays. This also means they cannot use the linker /V option (either through LINK /FOREGROUND/PROMPT or through LINK/FOREGROUND/XM) to enable the XM feature of .SETTOP and .LIMIT.

Figure 3–30: Privileged Background Job with .SETTOP

3.8.5.6 XM .SETTOP and Virtual Jobs

The monitor checks to see if there is some extended memory available. If the next free address is 200000, the program is already using the virtual address space controlled by Page Address Register 7. The request returns the value 177776 in $USRTO (location 50) and in R0.

If .SETTOP can obtain virtual space starting with the next free address (on a 4K-word boundary), the monitor creates a region in extended memory for the necessary amount of space. If not enough space is available, the monitor creates as large a region as possible. (Be sure to check the value .SETTOP returns.) Then the monitor creates a window and maps it to the new region. It returns the new value of the highest available address in $USRTO and in R0. If there is no space at all available, or if there are no region or window control blocks available, the request returns the value of the original highest available address in $USRTO and in R0.

So, for example, if you issue a .SETTOP request with an address argument, the monitor maps the virtual address space starting at the next 4K-word boundary above the program's virtual high limit, up to and including the address you specify. It maps so that the address specified is mapped, but up to 31_{10} additional words can also be mapped.

If the address you specify in the .SETTOP request is below the highest used address, .SETTOP returns the value of the next free address, minus 2, in $USRTO and in R0. The static window and virtual overlay regions created with the linker /V option cannot be eliminated by using an argument to .SETTOP.

Assuming your first .SETTOP succeeded and an extended memory region exists for your program, you can issue subsequent .SETTOP requests to control the region. Note, however, that you cannot create yet another region to obtain any more space.

If the argument you specify in your next .SETTOP request is lower than the original next free address, minus 2, from the link map, the monitor returns the old next free address, minus 2, in $USRTO and in R0 and eliminates the region and window, if present (along with any data stored there). You can, of course, issue another .SETTOP later to create a new region again. You can also adjust the size of the buffer by remapping within the same region.

To obtain a larger region, first issue a .SETTOP for a value below the current high limit, which eliminates the region and any data stored there. Then issue another .SETTOP for a larger value, which creates a new region. (Any data stored in the first buffer will be lost.) Note also that to ensure the integrity of your data, only one window exists for the .SETTOP area in an extended memory system.

To get less memory than a previous .SETTOP obtained, issue another .SETTOP with an address argument less than the first one but equal to or greater than the next free address. As a result, the size of the window still equals the size of the region, but a smaller amount of the window is mapped. This does not make any extended memory available for other users or other regions.

Virtual Background Jobs

Virtual background and foreground jobs are the most likely candidates for using the XM feature of the .SETTOP request. The request permits jobs to create large buffers in extended memory quickly and easily, which can help to reduce congestion in low memory. Figure 3–31 shows a virtual background job.

Virtual Foreground Job

In the multi-job mapped systems, the .SETTOP request works in much the same way for foreground jobs as for background jobs. For a virtual foreground job without the XM .SETTOP feature, the only extra space available is the space allocated through the FRUN/BUFFER:n command. For a job with the XM .SETTOP feature, the space allocated by the /BUFFER option is not used. (The job cannot have buffers in both

Figure 3–31: Virtual Background Job with .SETTOP

low and extended memory.) Figure 3–32 shows a virtual foreground or system job with a large buffer in extended memory.

3.8.5.7 .SETTOP and Completely Virtual Jobs

Both standard .SETTOP and XM .SETTOP in completely virtual jobs can create a region up to the 32K-word boundary (177776) and map to it in the same manner as XM .SETTOP in standard virtual jobs. However, an overriding limit can be set in block 0 of the program's .SAV image, using $JSX (word 4) and $VBGTO (word 6). (The structure is defined in the .SAV image section of the *RT–11 Volume and File Formats Manual*.)

Figure 3–32: Virtual Foreground or System Job with .SETTOP

The relevant bits in the $JSX word are ALL64$ and IOPAG$. The following values are returned by .SETTOP (and therefore are mapped) when $JSX and $VBGTO are set as indicated:

Table 3–16: XM .SETTOP and the Completely Virtual Job Environment

Condition	Address Limit
ALL64$ clear IOPAG$ clear	.SETTOP#-2 returns 157776
ALL64$ clear IOPAG$ set	.SETTOP#-2 returns 157776
ALL64$ set IOPAG$ clear $VBGTO equals 0	.SETTOP#-2 returns 177776 (-2)
ALL64$ set IOPAG$ clear $VBGTO equals *nnnnnn*	.SETTOP returns *nnnnnn*

3.8.5.8 Summary of .SETTOP Action

Figures 3–33, 3–34, and 3–35, and Tables 3–17, 3–18, and 3–19, work together to summarize the results of all possible .SETTOP requests. In Figure 3–33, Job A is a background job whose next free address is below SYSLOW, the base of the USR. Job B is a background job whose next free address is above SYSLOW. (In the tables and figures, *next free address* is abbreviated to *NFA*.) The values in parentheses represent specific ranges for .SETTOP arguments.

Completely virtual .SETTOP applies to both background and foreground/system jobs.

Figure 3–33: Background .SETTOP Summary

Table 3–17: Background .SETTOP Summary

.SETTOP Argument	Virtual Non-XM .SETTOP	Virtual XM .SETTOP	Privileged Non-XM .SETTOP	Privileged XM .SETTOP
High Limit for Job A After .SETTOP				
(1)	(1)	NFA—2	(1)	NFA—2
(2)	(2)	NFA—2	(2)	NFA—2
(3)	(3)	Map to (3)[1]	(3)	(3)
(4)	SYSLOW—2	Map to (4)[1]	SYSLOW—2	SYSLOW—2
#0	0	NFA—2	0	NFA—2
#–2	SYSLOW—2	Map to 32K[1]	SYSLOW—2	SYSLOW—2
High Limit for Job B After .SETTOP				
(1)	(1)	NFA—2	(1)	NFA—2
(2)	(2)	NFA—2	(2)	NFA—2
(3)	SYSLOW—2	NFA—2	SYSLOW—2	NFA—2
(4)	SYSLOW—2	Map to (4)[1]	SYSLOW—2	NFA—2
#0	0	NFA—2	0	NFA—2
#–2	SYSLOW—2	Map to 32K[1]	SYSLOW—2	NFA—2

[1]If available; otherwise, as much extended memory as possible is obtained for the .SETTOP region.

Figure 3–34: Foreground .SETTOP Summary

Table 3–18: Summary of Foreground Job High Limit After .SETTOP

.SETTOP Argument	Virtual Job Non-XM .SETTOP	Virtual Job XM .SETTOP
(1)	(1)	NFA—2
(2)	Greater of OHIGH or BUFF	NFA—2
0	0	NFA—2
–2	Greater of OHIGH or BUFF	Map to 32K

Figure 3–35: Completely Virtual Job .SETTOP Summary

Table 3–19: Completely Virtual Job .SETTOP Summary

.SETTOP Argument	Non-XM .SETTOP	XM .SETTOP
(1)	(1)	NFA—2
(2)	(2)	NFA—2
(3)	(3)[2]	Map to (3)[12]
#0	0	NFA—2
#–2	32K[2]	Map to 32K[12]

[1]If available; otherwise, as much extended memory as possible is obtained for the .SETTOP region.
[2]Subject to the overriding maximum .SETTOP limit; see Section 3.8.5.7.

3.8.6 Directly Modifying Address Page Registers (APRs)

As described in Section 3.6.6, the following requests perform an alignment between the affected WCBs and the corresponding APRs:

- .CMAP
- .CRAW/.ELAW
- .ELRG
- .MAP/.UNMAP
- .MSDS
- .SETTOP (XM type)

Also, you can directly modify the contents of an APR that is enabled by the .CMAP request, and the modification is maintained across all context switches.

You can use the MOV instruction to directly modify APRs, bypassing the mapping requests. Modifications are maintained across all context switches for User mode APRs. However, such a changed APR is no longer aligned with the corresponding WCB, but is preserved across a context switch so long as no request listed above is issued for the corresponding WCB. (Those requests align the affected APRs with the corresponding WCBs.)

By default, all Supervisor mode D-space APRs are initially locked with the corresponding User mode I-space APRs (see Section 3.7.1). As mentioned in Section 3.7.1, once the User mode address spaces are separated (and CM.DUS remains in effect), all Supervisor mode D-space APRs are locked with the corresponding User mode D-space APRs. If you directly modify a Supervisor mode D-space APR such that it is no longer aligned with the corresponding User mode APR, that Supervisor mode APR will remain unaligned until one of the following mapping requests is issued. Then, the monitor will force all Supervisor mode D-space APRs into alignment with the corresponding User mode APRs:

- .CMAP/.CGMAP/.MSDS
- .CRRG/.ELRG
- .CRAW/.ELAW
- .MAP/.UNMAP
- .GMCX
- .SETTOP (XM type, when it implicitly calls any of the above requests)

If Supervisor mode D-space is not enabled, Supervisor mode I-space APRs are locked down instead.

3.9 Hardware Concepts

The following sections provide general information on PDP–11 hardware concepts that are applicable to mapped systems. You should consult the hardware documentation for your processor for more complete information.

3.9.1 Virtual and Physical Addresses with Extended Memory Hardware

The virtual addresses your program uses are always limited to 16 bits so that your program's virtual address space is always limited to 32K words. A single job can have multiple virtual addresses by splitting User mode I & D address space. Virtual addresses can be further split by using Supervisor mode I & D address spaces.

However, an 18-bit address can reference any location between 0 and 128K words; a 22-bit address can reference any location between 0 and 2048K words. On RT–11 systems with more than 28K words of memory, physical locations are referenced by the hardware as 18- or 22-bit addresses.

As Figure 3–36 shows, there can no longer be a direct one-to-one correspondence between virtual and physical addresses.

3.9.2 Circumventing the 32K-Word Address Limitation

Through its **mapped** monitors (XB, XM, ZB, and ZM), RT–11 provides a mechanism to associate a virtual address with a physical address. This process is called mapping. RT–11 permits programs to access extended memory by mapping their virtual addresses to physical locations in memory. In summary:

- Every location in memory has an 18- or 22-bit physical address; there are more physical addresses than virtual addresses.
- A program cannot access specific physical addresses unless its virtual addresses are mapped to those physical locations.
- Programs can access all the available physical memory by using their virtual addresses over and over again, but with different mapping each time.

In an extended memory system, programs are no longer limited to using 28K words of memory. However, they must still deal with the 32K-word addressing limitation for each address space. Typically, large programs are still divided into smaller segments, as in the 28K-word systems. While the instructions and data in separate segments of a program share the same virtual addresses, they can have unique physical addresses. Figure 3–37 shows a program that is divided into three overlay segments. The three segments are resident simultaneously in extended memory, but they share the virtual addresses in overlay region 1.

Figure 3–36: Virtual and Physical Addresses with Extended Memory Hardware

3.9.3 Concept of Pages

In an extended memory system the 32K-word virtual address space is divided into eight sections called **pages**. Each page begins on a 4K-word boundary, and the pages are numbered from 0 through 7. A page is made up of units of 32_{10} words each. Since there can be as many as 128 of these units, a page can vary in size from 0 words to 4096 words, in 32-word increments. Figure 3–38 shows the virtual address space divided into eight 4K-word pages.

Figure 3–39 shows the virtual address space divided into five pages of varying lengths. The shaded areas in the virtual address space are not part of the pages,

Figure 3–37: Program Segments Sharing Virtual Address Space

and are therefore inaccessible. Thus, short pages cause gaps in the virtual address space.

3.9.4 Relocation

When the Memory Management Unit converts a 16-bit virtual address to an 18- or 22-bit physical address, it *relocates* the virtual address. This means that two or more programs can have the same virtual addresses but different physical addresses. The Memory Management Unit relocates virtual addresses in units of pages. It assigns a page to a section of physical memory that starts on a 32_{10} word boundary.

Figure 3–38: 4K-Word Pages

Figure 3–40 shows how the Memory Management Unit can relocate the virtual addresses of two different programs in a 124K-word memory.

Program 1 in Figure 3–40 is relocated by 20000. So, when program 1 references virtual address 0, for example, it actually accesses memory location 20000.

Since the Memory Management Unit relocates each page of virtual address space separately, a program can reside in disjoint sections of memory, as Figure 3–41 shows.

3.9.5 Active Page Register (APR)

The RT–11 monitor communicates with the Memory Management Unit through the Active Page Registers, which are located in the I/O page. Each Active Page Register consists of two 16-bit words, as shown in Figure 3–42: a Page Address Register (PAR), and a Page Descriptor Register (PDR).

The Page Address Register and the Page Descriptor Register always act as a pair. A set of eight Active Page Registers contains all the information necessary to describe

Figure 3–39: Smaller Pages

and relocate the eight virtual address pages. The Page Descriptor Register describes how much of a virtual page to map to memory. The Page Address Register describes where in memory to put the virtual page.

The eight Active Page Registers are numbered from 0 through 7. There is one Active Page Register for each page in the 32K-word virtual address space, as Figure 3–43 shows.

3.9.5.1 Page Address Register (PAR)

The eight Page Address Registers correspond directly to the eight virtual address pages. The Page Address Register contains the physical memory address in 32_{10}-word units, or Page Address Field, for a particular virtual address page. Each 32_{10}-word unit is commonly called a *chunk*. Figure 3–44 shows the contents of the Page Address Register. Bits 0 through 11 are used for 18-bit addressing; bits 0 through 15 are used for 22-bit addressing.

Figure 3–40: Relocation by Program

3.9.5.2 Page Descriptor Register (PDR)

The Page Descriptor Register contains information about page expansion, page length, and access control for a particular page. Like the Page Address Registers, the Page Descriptor Registers correspond directly to the virtual address pages, as Figure 3–43 shows. Figure 3–45 shows the contents of the Page Descriptor Register. Unused bits are reserved for future use by Digital.

In Figure 3–45, the field marked **ACF** represents the **Access Control** field. This field describes how a particular page can be accessed, and whether or not a particular access should cause an abort of the current operation. The values in this field are as follows:

Figure 3–41: Relocation by Page

Figure 3–42: Active Page Register (APR)

Figure 3–43: Correspondence Between Pages and Active Page Registers

Figure 3–44: Page Address Register (PAR)

Figure 3–45: Page Descriptor Register (PDR)

Value	Meaning
00	Nonresident page. Abort any attempt to access it.
01	Resident read-only page. Abort any attempt to write into it.
10	Unused code. Abort all attempts to access this page. (RT–11 does not use this value.)
11	Resident read/write page. All accesses are valid.

The field marked **ED** is the **Expansion Direction** field. This bit indicates the direction in which a page can expand. The codes and their meanings are as follows:

Value	Meaning
0	The page expands to higher addresses. (In RT–11, this field is always 0.)
1	The page expands to lower addresses. (RT–11 does not use this value.)

The field marked **W** is the **Written Into** field. It indicates whether the page has been modified since it was loaded into memory. (RT–11 does not use this field.)

Some PDP–11 processors, instead of using bit 6 to indicate the page's modification status, use one or more of the reserved bits in the Page Descriptor Register. RT–11 ignores these other bits.

The field marked **PLF** is the **Page Length field**. It indicates the length of a page, in 32_{10}-word units.

The field marked **CB** (bit 15) indicates references bypass cache. If clear, use cache, if available.

3.9.6 Converting a 16-Bit Address to an 18- Or 22-Bit Address

The information necessary for the Memory Management Unit to convert a 16-bit virtual address to an 18- or 22-bit physical address is contained in the virtual address and in its corresponding Active Page Register set. Figure 3–46 shows the meanings of the fields in the virtual address. These fields represent a breakdown of the virtual address that is convenient for RT–11 and the MMU to use.

Bits 13 through 15 of the virtual address constitute the *Active Page Field*. This field determines which Active Page Register the Memory Management Unit will use to create the physical address.

Bits 0 through 12 of the virtual address are the *Displacement Field*, which contains an address relative to the beginning of a page.

Figure 3–46: Virtual Address

Figure 3–47: MMU Address Conversion (Detail)

The rest of the information necessary to create a physical address is contained in the Page Address field of the appropriate Page Address Register. Figure 3–47 shows how, in a multi-job extended memory system, the Memory Management Unit converts a 16-bit virtual address to an 18- or 22-bit physical address. In this example, Page Address Register 6 contains 5460, so virtual address 157746 converts to physical address 565746. Bits 12-15 of the Page Address Register are included for 22-bit addressing.

As you can see from Figure 3–47, bits 13, 14, and 15 of the virtual address specify which Active Page Register to use. The Memory Management Unit adds the value in

bits 6 through 12 of the virtual address to the corresponding Page Address Register. The Memory Management Unit places the result of this addition in bits 6 through 17 or 6 through 21 of the physical address. The Memory Management Unit copies the value in bits 0 through 5 of the virtual address into bits 0 through 5 of the physical address to form the final 18- or 22-bit physical address.

3.9.7 Status Registers

The Memory Management Unit also communicates with the RT–11 monitor through two status registers. Status Register 0, located at 777572 in the I/O page, contains abort error flags, the memory management enable bit, and other essential information required by RT–11 to recover from an abort or to service a memory management trap. Status Register 2, located at 777576, is a read-only register containing the 16-bit virtual address that the Memory Management Unit is currently converting to an 18- or 22-bit physical address. (RT–11 does not use Status Register 2. However, if a memory management unit fault occurs in your system, you can examine this register yourself.) RT–11 also uses Memory Management Register 3 (MMSR3), located at 772516, to enable 22-bit addressing and to manage Supervisor mode and I & D address space.

3.10 Restrictions and Design Implications

The manner in which RT–11's support for extended memory is implemented imposes some restrictions on the ways you can use the system. The following sections outline the implications of the design of the extended memory system.

3.10.1 PAR1 Restriction

The PAR1 restriction does not apply to completely virtual programs.

The RT–11 monitor sometimes ''borrows'' kernel Page Address Register 1 for its own use. For example, it uses PAR1 to map to the EMT area blocks when it processes a programmed request.

Because the monitor alters kernel PAR1, references to virtual addresses in the range 20000 through 37777 do not always access the corresponding physical addresses. To avoid problems due to the occasional remapping of the virtual addresses controlled by kernel PAR1, observe the following programming restrictions.

1. Any channel areas you allocate with the .CDFN programmed request must be entirely within the low 28K words of memory. In addition, they must not be located within the addresses 20000 through 37777.

2. Any queue elements you allocate with the .QSET programmed request must be entirely within the low 28K words of memory. In addition, they must not be located within the addresses 20000 through 37777. Remember to allow 10_{10} words per queue element.

3. Interrupt service routines must be located entirely within the low 28K words of memory. In addition, if your single-mapped monitor has been generated without .FETCH support, they must neither reside in nor reference addresses in the range 20000 through 37777. Chapter 5 describes the factors you must take

into consideration if your program includes an in-line interrupt service routine. Be sure to execute your program as a privileged job if it contains an interrupt service routine, so that it can access the monitor and the device I/O page. The *RT–11 Device Handlers Manual* lists the implications of mapped monitor design restrictions on device handlers and I/O.

This aspect of RT–11's design is important for you to understand if you have a program with its own in-line interrupt service routine, if you put a data buffer for I/O in extended memory, or if you write a device handler for a single-mapped system.

3.10.2 PAR2 Restriction

You cannot run a completely virtual job on a monitor that is built for MQ to use kernel PAR2 to map user buffers (MQH$P2=1).

The MQ message handler uses Page Address Register 2. If you use the MQ handler to send and receive messages in the multi-job XM system, be sure to read Section 2.5.7. If the MQ handler was built with the conditional assembly symbol MQH$P2 set qual to 1, MQ will use kernel PAR2 to map the user buffers. In that case, all the PAR1 restrictions apply as well to the virtual addresses in the range 40000 through 57777, controlled by PAR2. Therefore, the USR, queue elements, channels, and interrupt service routines cannot reside within locations 20000 through 57777 (kernel PARs 1 and 2) in a system that is actively using the MQ handler. Note that the QUEUE program uses the MQ handler.

3.10.3 Programmed Requests

Some of the RT–11 programmed requests have special restrictions when you use them in an extended memory system. These requests and their restrictions are as follows:

Programmed Request	Restriction
.CDFN	The channel area you specify in this request must be entirely within the low 28K words of memory.
.QSET	The queue element space you specify must be entirely within the low 28K words of memory. In addition, you must allow 10_{10} words for each queue element.
.CNTXSW	Virtual jobs cannot use this request, since they have no need for it in an extended memory system.

3.10.4 Synchronous System Traps

A synchronous system trap is a software interrupt that takes place synchronously with your program's execution. For example, a TRAP instruction that a program issues is a synchronous system trap. A program that issues an illegal instruction causes a trap to 10 to occur, which is also a synchronous system trap. When a trap occurs, the PDP–11 computer pushes the current PS and PC onto the stack and

loads the new PS and PC from the contents of the trap vector. Table 3–20 lists the synchronous system traps and their corresponding vectors.

Table 3–20: Synchronous System Traps and Their Vectors

Vector	Synchronous System Trap
4	Trap to 4, caused by a reference to an odd address, or by a bus time-out.
10	Trap to 10, caused by an attempt to execute a reserved instruction.
14	Breakpoint trap, usually issued by a debugging utility program such as ODT.
20	I/O trap.
30	EMT vector (reserved to Digital).
34	TRAP instruction, issued by a program to change the flow of execution.
114	Memory parity trap, caused by a memory parity error.
244	FPU trap, caused by a floating point unit exception or error.
250	Memory management trap, caused by a program's attempt to reference a virtual address that is not mapped to a physical address.

In a mapped system, synchronous system traps, like device interrupts, take the new PS and PC from the appropriate vector in kernel space. For example, when a program issues a BPT instruction, the new PS and PC are taken from physical locations 14 and 16. As you remember, a privileged job is initially mapped to the kernel vector area, so virtual address 14 in the program maps to physical location 14. A virtual job, on the other hand, is prevented from directly accessing the kernel vector area. Initially, the background virtual job's vector area maps to physical addresses starting at location 500, not 0. For a virtual job then, the virtual vector 14 is not in physical location 14.

For each synchronous system trap, RT–11 provides a mechanism to field the trap and provide values for the new PS and PC from the virtual vector. The following sections describe the effect of the extended memory environment on specific synchronous system traps.

3.10.4.1 TRAP, BPT, And IOT Instructions

When a program in an extended memory system issues a TRAP, BPT, or IOT instruction, execution switches to the processor's kernel mode. The hardware picks up the contents of the appropriate vector (see Table 3–20) from kernel space. However, rather than dispatching immediately to the trap handling routine specified in the kernel vector, the monitor replaces the new PS and PC with values that cause execution to continue within a monitor routine. The purpose of the monitor routine is to pick up the contents of the corresponding virtual vector in user space, and then transfer control to the routine specified by the virtual PC. The kernel and user vectors for a privileged job are identical. A virtual job cannot directly access the kernel vectors; you can, however, put values into the virtual vectors so that the monitor will pick them up when a trap occurs. In summary, the net effect of the

monitor's trap handling routine is that control is transferred to a job's specific trap routine through the contents of the job's virtual vector.

The monitor does not clear the vector after the first trap. This permits recursion with no effort on the part of the program.

3.10.4.2 Traps to 4 and 10, and FPU Traps

For traps to 4 and 10, and floating point unit exception traps, the monitor provides a mechanism that protects the vectors while still permitting you to use your own trap handling routines. The .TRPSET and .SFPA programmed requests permit your program to set up the addresses of trap handling routines without modifying either the kernel or the user virtual vector area. Thus, you specify the address of your trap handling routine when you issue the programmed request and the monitor puts this information in the job's impure area. The monitor clears out the routine address in the impure area, so your trap handling routine should reset this area by issuing either .TRPSET or .SFPA as its last instruction before returning to the main program.

3.10.4.3 Memory Management Faults

A memory management fault occurs when a program references a virtual address that is not mapped to a physical address. If a memory management fault occurs while execution is in system state, the entire system halts. If a memory management fault occurs while execution is in user state, the monitor fields the trap through the kernel vector and provides a new PS and PC from the user virtual vector area. Once the monitor picks up the contents of a job's virtual vector, it clears the vector. If a second fault occurs and the virtual vector is 0, the monitor prints its *?MON-F-MMU fault* message and aborts the job.

To permit recursion, your program's trap handling routine must reset the contents of the memory management fault vector (at locations 250 and 252) in the job's virtual vector area. If RT–11 permitted automatic recursion, your program could loop indefinitely on a memory management fault until you halted the processor.

3.10.4.4 Memory Parity Errors

A hardware device that is an optional part of your PDP–11 computer system performs memory parity checking. You enable RT–11 support of this hardware option by selecting the memory parity special feature at system generation time. If you have memory parity hardware but do not generate a system with the memory parity checking special feature, a memory parity error causes a system halt.

For systems that support memory parity checking, the synchronous system trap procedure is similar to the procedure for memory management faults. Thus, the monitor fields the trap through the kernel vector at locations 114 and 116. It then picks up the contents of your program's virtual addresses 114 and 116, clears them, and passes control to your trap handling routine based on the new PS and PC.

If a second memory parity error occurs and the virtual vector is 0, the message *?MON-F-Mem err* prints and the job aborts. To enable recursion, your program's

trap handling routine must reset the contents of the memory parity fault vector at virtual addresses 114 and 116.

3.11 Debugging an Extended Memory Application

The first choice in debugging a mapped application is the symbolic debugger, DBG–11, described in the *DBG–11 Symbolic Debugger User's Guide*. If you are not familiar with DBG–11, look there for information. An example debugging session with a mapped application is included.

If for some reason you cannot or choose not to use DBG–11, use VDT, the Virtual Debugging Technique. Use VDT.OBJ the same way you use ODT.OBJ; link it with the program you need to debug. The transfer address for VDT is O.ODT. The syntax for VDT commands is the same as the syntax for ODT. See the *RT–11 System Utilities Manual* for instructions on using ODT.

VDT does not contain the interrupt service or priority routines that ODT does. Unlike ODT, which runs at priority 7 and performs its own terminal I/O, VDT runs at the same priority as your program, and uses .TTYIN and .TTYOUT programmed requests to perform terminal I/O.

Because VDT uses .TTYIN and .TTYOUT requests, you can run it from a job's console terminal; it is not limited to the hardware console interface. Since VDT alters the contents of the Job Status Word, it must save the original contents elsewhere. You can use the $J/ command to obtain the original contents of the JSW; you can also modify it there.

VDT runs in user, not in kernel mode. When you debug a virtual job with VDT, you are limited to accessing the job's area only. You cannot access the protected system areas such as the monitor, the vectors, and the I/O page. When you debug a privileged job with VDT, you have access to the same memory the job does.

3.12 Extended Memory Example Program

The *RT–11 System Macro Library Manual* provides an example program that uses extended memory programmed requests.

3.13 Procedure to Create and Map a Global Region

The procedure in Figure 3–48 creates and maps to a global region in extended memory.

1. From a program, issue a .CRRG programmed request pointing to a global region definition block. Specify the .RDBBK macro with a decimal value for R.GSIZ in the same manner as for a local region. Specify status arguments RS.GBL (attach to a global region) and RS.CGR (create a global region).

   Specify status argument RS.AGE if you want to enable automatic global elimination in a multi-job mapped system.

   Specify a name for the created global region.

2. After executing .CRRG, check RS.CRR (bit 15, region definition status word R.GSTS) for success.
3. Digital recommends that if there is any possibility of a race condition (two programs attempting to attach at exactly the same time), include a .TWAIT request in your retry code in multi-job mapped systems.
4. Map to the created region in the same manner as you map to a region local to a program. Use the .CRAW and .MAP requests.
5. Check RS.NEW (bit 12, region definition block) to confirm that the program did in fact create the global region and is not attaching to an already created one.
6. If required, load the global region with data.
7. If you want to make the global region available to other programs at creation in a multi-job mapped system, execute a .ELRG request. Then reattach your program to the global region using .CRRG. If you want to make the global region private to your program, do not execute a .ELRG/.CRRG reattach sequence.

Successful creation of a global region sets RS.CRR and RS.NEW (bits 15 and 12 in global region definition block status word R.GSTS).

The following example, written for a multi-job mapped system, creates and maps to a 4K word global region named MYDATA and (because of the .ELRG request) allows other programs to attach to it.

Figure 3–48: Creating and Mapping a Global Region

```
        .MCALL  .CRAW,  .CRRG,  .ELRG,  .EXIT,  .MAP,   .TWAIT
        .MCALL  .RDBBK, .WDBBK

        ERRBYT  =       52
        USERRB  =       53
        FATAL$  =       10
        XE.PRV  =       15

REGION: .RDBBK  4096./32.,<RS.GBL!RS.CGR>,NAME=MYDATA
WINDOW: .WDBBK  1,4096./32.
TIME:   .WORD   0,1.*60.        ;1 second (60 ticks/second)
AREA:   .BLKW   6.              ;Programmed request area

;+
; The following four entry points are the error handlers for the
; PLAS directives.  They may be expanded as is appropriate.
;-

CRRGER:
CRAWER:
ELRGER:
MAPERR: BISB    #FATAL$,@#USERRB        ;Indicate severe error
                                        ; to system
        .EXIT                           ; and exit
```

Figure 3–48 (continued on next page)

Figure 3–48 (Cont.): Creating and Mapping a Global Region

```
START:  .CRRG   #AREA,#REGION           ;Create a 4K word global
                                        ;region, "MYDATA"
        BCC     10$                     ;Branch on success
        CMPB    #XE.PRV,@#ERRBYT        ;Is global region
                                        ;privately owned?
        BNE     CRRGER                  ;If not, then goto
                                        ;.CRRG error handler
        .TWAIT  #AREA,#TIME             ;Wait a second
        BR      START                   ;Try again

;+
; We have now successfully allocated/created the "MYDATA" global
; region.  Now map it to a window, initialize if necessary, and
; then use it.
;-
10$:   MOV     REGION+R.GID,WINDOW+W.NRID ;Put region ID in the WDB
        .CRAW   #AREA,#WINDOW           ;Create a window
        BCS     CRAWER                  ;If error, go handle it
        .MAP    #AREA,#WINDOW           ;Map the window to the region
        BCS     MAPERR                  ;If error, go handle it
        BIT     #RS.NEW,REGION+R.GSTS   ;Did we just create the
                                        ;global region?
        BEQ     30$                     ;Branch if not

;+
; We just created global region "MYDATA".  We now initialize the
; region with zeros.
;-
        MOV     #4096.,R0               ;Set up word count of 4K
        MOV     WINDOW+W.NBAS,R1        ;Get virtual address
                                        ;of base of region
20$:    CLR     (R1)+                   ;Clear a word
        SOB     R0,20$                  ; until we're done

;+
; Now detach from global region "MYDATA" to make it shareable.
; Then loop back and reattach to global region "MYDATA".
;-
        .ELRG   #AREA,#REGION           ;Detach from global region
        BCS     ELRGER                  ;Branch if there is an error
        BR      START                   ;Go attach to global
                                        ;region "MYDATA"

;+
; We are attached to a shareable global region.  If we also created
; it, we have initialized the region as well.  The following code can
; use global region "MYDATA" as it sees fit.
;-
30$:    .
        .
        .
```

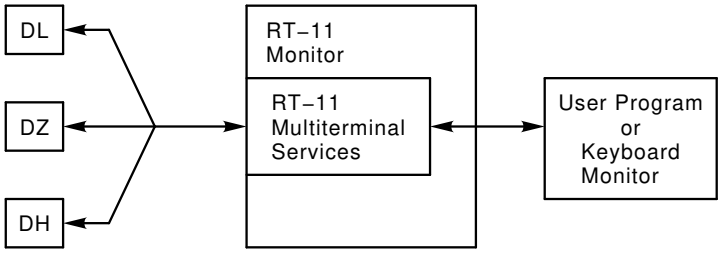
Figure 3–48 (continued on next page)

Figure 3–48 (Cont.): Creating and Mapping a Global Region

```
        .END    START
```

Chapter 4
Multiterminal Feature

In describing the multiterminal feature of RT–11 this chapter provides background information on the hardware and describes the data structures of a multiterminal system. It also describes the interrupt service and polling routines, the programmed requests available to application programs, and typical situations in which you can use two terminals without making use of the multiterminal special feature. An example program is provided at the end of the chapter.



4.1 Components of a Multiterminal System

RT–11 implements support for multiple terminals as a special feature that you select during system generation. Essentially, the multiterminal feature permits an application program to control one or more terminals. It does not change RT–11's basic characteristic of being a single-user operating system. Specifically, multiterminal support does not permit more than one terminal at a time to be the command terminal, the terminal at which you communicate with RT–11 through DCL commands.

Support for multiple terminals is implemented through the following components:

- MTTEMT.MAC processes the multiterminal programmed requests.
- MTTINT.MAC contains the multiterminal interrupt service and polling routines.
- TRMTBL.MAC defines the multiterminal terminal control blocks.

MTTEMT, MTTINT, and TRMTBL assemble and link together as part of RMON for a multiterminal system.

There are also some important data structures and concepts in multiterminal systems:

- *Terminal control blocks*, called TCBs (one per terminal), contain information about the terminal and controlling job. The TCBs also contain the input and output ring buffers for the terminal.

- *Logical unit numbers*, called LUNs, through which RT–11 refers to the terminals that are part of your system.
- *Asynchronous terminal status words*, called AST words (one per LUN), in which RT–11 maintains event flags to reflect the current status of each terminal. Support for AST words is a special feature you can select through system generation. The address of AST words are supplied to the system by the .MTATCH request and reside in the user program.
- *Terminal hooks*, called THOOKS, is a data structure that contains addresses of RMON routines that pertain to the terminal support code. This data structure is created in RMON when you request the terminal hooks option.

4.2 Hardware Background Information

This section provides some background information that is useful if you are unfamiliar with the communication hardware RT–11 supports.

RT–11 supports the serial interfaces: DL series (including DL11 and DLVJ1, or compatible equivalent), the DH series (but not the DH11), and the entire DZ series. An interface is similar to a device controller; it stands between the computer and a serial line. The other end of the line can be connected to a terminal, a modem, or another computer.

The DL interface connects the computer system to a single serial line. Each DL interface has its own Control and Status Register (CSR) address and vector address. RT–11 supports up to eight DL interfaces on your computer system, including the hardware console interface. Since each DL interface is a separate controller, there is no real physical unit number; 0 is assigned for consistency. Note that even though the DLVJ1 module contains four serial lines, they appear to the software as four separate and distinct DL interfaces.

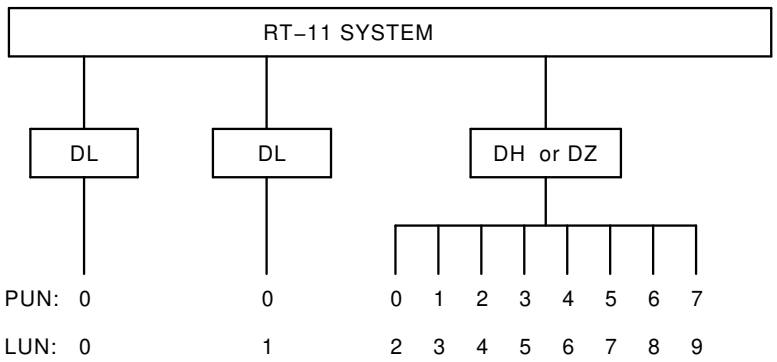
Each RT–11 system must have a hardware console interface so that the hardware can use it at bootstrap time to communicate with the console terminal. The hardware bootstrap on many systems requires that a terminal be connected at the standard console addresses for diagnostic purposes and for operator communication at bootstrap time. Your hardware console interface must be a local DL. Its interrupt vectors are located at 60 and 64 in low memory, and its LUN is always 0.

A DH or DZ interface is called a *multiplexer*; it connects several serial lines through a single pair of CSR and vector addresses.

- RT–11 supports up to two DH interfaces for a total of up to 16 serial line connections. At least one DL interface is required for the console terminal, for a permitted total of 17 lines.
- The DZ11 interface connects the computer system to eight lines that have physical unit numbers from 0 through 7. The DZV11 is similar to the DZ11, but it connects the system to only four lines that have physical unit numbers from 0 through 3. You can have two DZ11 or four DZV11 interfaces, for a total of 16 additional lines.

Figure 4–1 illustrates the interfaces and their physical and logical unit numbers.

Figure 4–1: Interfaces and Physical and Logical Unit Numbers



During system generation, you specify how many DL, DH, and DZ interfaces your target system has. You also indicate how many of their physical units are actually connected to terminals on the system. Of those terminals, you must indicate which are local and which are remote lines. Unlike physical unit numbers, which are numbered starting at 0 for each interface, the logical unit numbers that RT–11 uses are unique. They begin at 0 and continue until all terminals have been accounted for.

SYSGEN assigns the physical unit numbers of the interfaces to its software logical unit numbers in the following order:

1. Local DL lines (the hardware console interface is always LUN 0)
2. Remote DL lines
3. Local DZ lines
4. Remote DZ lines
5. Local DH lines
6. Remote DH lines

The order in which SYSGEN assigns physical lines to logical unit numbers is also the order in which it generates the terminal control blocks. It generates one TCB for each line you specify in the SYSGEN dialogue. The TCBs are arranged in RMON in the order in which you specify the lines to SYSGEN. There are no TCBs for any unused interface physical lines.

When you bootstrap a multiterminal system, RT–11 checks for the presence of each interface for which a TCB exists by attempting to access its CSR, as specified in the SYSGEN dialogue. If the interface does not exist, the logical unit number associated with that interface is marked as nonexistent, and any attempt to attach such a LUN results in an error. The space occupied by the TCB of a nonexistent LUN is not

recoverable. You can use the SHOW TERMINALS monitor command to verify that the information you supplied during system generation was correct.

Note that RT–11 does not attempt to determine whether or not a terminal or modem is actually connected to an interface line; it assumes the connection is present. For an unconnected line, no meaningful input characters can be generated; output directed to the line is sent out and lost.

4.3 What is the Console Terminal?

A potentially confusing aspect of RT–11's multiterminal support is its ability to change the console terminal. This section defines precisely what is meant by the terms *hardware console interface*, *boot-time console*, *background console*, and *private console*. You will avoid confusion if you familiarize yourself with these terms and use them consistently.

The *hardware console interface*, as Section 4.2 describes, is the terminal interface located at vectors 60 and 64, whose control and status registers begin at 177560 in the I/O page. This is the serial line interface the hardware bootstrap uses at bootstrap time. (Generally, you must have a terminal connected to the hardware console interface in order to bootstrap the system.) This is almost always the terminal on which RT–11 prints its startup message. Remember that the hardware console interface is always LUN 0.

The *boot-time console* is the terminal on which RT–11 prints its startup message. This is almost always the same as the terminal connected to the hardware console interface. In a system without the multiterminal feature, the CSR for this terminal, 177560, is contained in $TTKS. ($TTKS is located at fixed offset 304 from the start of RMON.) In a multiterminal system, the CSR is located at offset T.CSR in the first TCB in RMON.

The *background console*, also called the *command console*, is originally the same as the boot-time console. (It remains the same until you use the SET TT: CONSOL command, described below, to move the background console.) It is the terminal on which you type commands to KMON, and through which you communicate with the background job. If you run a foreground job or system jobs, they can share the background console. In this case, you must use CTRL/B to communicate with the background job, CTRL/F for the foreground job, and CTRL/X for the system jobs. For example, to abort a job from a shared console, you must either type the appropriate CTRL sequence, followed by two CTRL/C characters, or use the DCL ABORT command. (See Chapter 2 for more information on control sequences.)

The programmed requests .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT interact with the background console for the background job, and also for any foreground or system jobs that happen to be sharing this terminal.

NOTE

RT–11 ignores any unit number you specify with device TT. Therefore, references to TT:, TT0:, TT1:, and so on,

are all equivalent, and refer to the console of the job that issues the request.

In a multiterminal system you can move the background console to another terminal by issuing the SET TT: CONSOL monitor command. By specifying another logical unit number in the SET command, you can move the background console to any other local terminal in the system, except to a private console.

A *private console* is a local terminal used by a single foreground or system job. You give a job its own private console when you start the job by using the FRUN /TERMINAL:n or SRUN/TERMINAL:n commands. No other job can share a private console with the original job. A job's private console is the terminal with which its .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT programmed requests interact. In addition, any .READ or .WRITE requests to TT that this job makes access the private console. When a job has its own private console, you can no longer communicate with the job through the background console. Thus, you can no longer use CTRL/F at the background console, for example, to interact with a foreground job that has its own private console; instead, you must type on the private console. To abort this foreground job, you must type two CTRL/Cs on its private console or abort the job, using the DCL ABORT command from the background console. You cannot issue DCL commands from a private console.

You cannot change a private console to a different terminal by using the SET TT: CONSOL command; that command is valid only for the background console. This is because KMON runs as a background job, and it can run only on the background console.

A *shared console* refers to the background console unless the following conditions apply:

- In a system without the system job feature, the foreground job is running with a private console;
- In a system with the system job feature, all six system jobs and the foreground job are running, and each has a private console.

Remember that a private console can never be shared.

A *console* simply refers to a terminal being used as the background shared console, or as a foreground or system job private console.

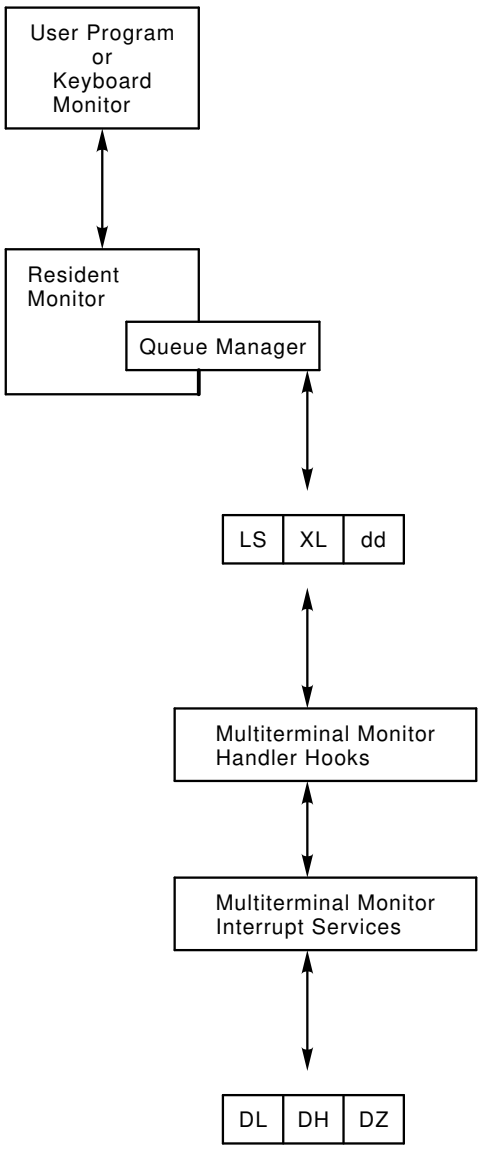
4.4 Connecting Handlers to Terminal Lines

Through the SYSGEN procedure, you can build support for a monitor and handlers that lets each such handler connect to the port controller on a terminal line without regard for the interface to which it is ultimately connected. The multiterminal interrupt service manages the port controller. Such support is called *multiterminal handler hooks* and the *hooks* are the connection between the handler and the interrupt service.

Multiterminal handler hooks support is valid for DH, DL, and DZ controllers and all multiterminal monitors. For example, the distributed LS and XL handler

source files can be built to make use of such support. Standard interface support and multiterminal handler hooks support can be included in the same handler. Figure 4–2 illustrates the multiterminal handler hooks connection.

Figure 4–2: Multiterminal Handler Hooks Connection



See the following sections for information on monitor support and a short discussion of such support in LS and XL. See the *RT–11 Device Handlers Manual* for more information on LS and XL and for information on including multiterminal handler support within your own handler. See the *RT–11 System Generation Guide* for SYSGEN dialogue questions that build this support in the monitor and handlers.

4.4.1 Monitor Support

Support for the multiterminal handler hooks is built into a monitor when the system conditional, MTY$HK, is set to 1 (MTY$HK=1). That support places a pointer in RMON fixed offset 472, $THKPT, to the monitor handler hooks data structure, THOOKS. (If MTY$HK=0, $THKPT has a zero value.)

Further, when a handler built for such support is fetched or loaded, it writes values into the TCB for the terminal line it will use. The handler places the address of the handler hook routine entry point in T.OWNR (TCB offset 12) and sets HANMT$ (bit 3) in the T.STAT word (TCB offset 14). The handler must also ensure that the word before the handler hook routine entry point contains the RAD50 physical name of the handler.

The basic protocol between the monitor interrupt service and the device handler is as follows:

1. An interrupt occurs and the interrupt service routine in the multiterminal monitor is entered.
2. If the line on which the interrupt occurred is not attached by a handler, the interrupt is processed normally.
3. If the line on which the interrupt occurred is attached by a handler (HANMT$ is set in T.STAT), the interrupt service code passes control to the handler hook routine (using the address in T.OWNR).

   R0 indicates the type of interrrupt and R5 is used to pass characters between the handler and the interrupt service code.

   If the monitor is passing a character to the handler (an input interrupt), the monitor clears R0 (function code TH.PIC) and passes the character to the handler in R5. The handler processes the character, preserving all other registers, and returns.

   If the monitor is requesting an output character from the handler because of an output interrupt, the monitor places the value 1 in R0 (function code TH.GOC) and calls the handler hook code, which returns a character in R5 with carry clear. If the handler cannot pass a character, it sets the carry bit and returns. All other registers are preserved across the call.

The monitor part of the protocol is described in the following sections. The handler part of the protocol is described in the *RT–11 Device Handlers Manual*. In particular, see the XL handler section for information on an example handler hook routine, and the XL handler description in the Appendix.

4.4.1.1 Terminal Hooks Data Structure, THOOKS

The terminal hooks data structure, THOOKS, is defined by the .THKDF macro in the system definition library, SYSTEM.MLB. THOOKS contains the number of TCBs in the system and pointers to monitor routines that process character input and output and manage serial lines for modem control:

Table 4–1: Terminal Hooks Data Structure, THOOKS

Offset	Name	Contents
00	THK.LE	Length (in bytes) of the data structure; for Version 5.6, THK.LE contains THK.SZ (14_8). (A byte offset.)
01	THK.NU	Number of TCBs in system (and therefore the number of possible LUNs). (A byte offset.)
02	THK.TC	Pointer to list of pointers to TCBs. The first word points to the TCB for LUN 0, the second word points to the TCB for LUN 1, and so forth.
04	THK.OE	Pointer MTOENB, the terminal output enable routine. See Section 4.4.1.2.
06	THK.BK	Pointer to MTYBRK, the terminal line BREAK routine. See Section 4.4.1.3.
10	THK.CT	Pointer to MTYCTL, the terminal modem control routine. See Section 4.4.1.4.
12	THK.ST	Pointer to MTYSTA, the terminal modem status routine. See Section 4.4.1.5.
	THK.SZ	Length of the structure, stored in THK.LE (offset 00).

4.4.1.2 Terminal Output Enable Routine, MTOENB

The handler calls MTOENB to inform the monitor that the handler has waiting output. The handler cannot initiate output but rather only responds to the monitor. The handler uses MTOENB to tell the monitor that it should enable output interrupts for the terminal line. The MTOENB offset calls the monitor's output interrupt routine, TTOENB.

MTOENB is called through offset 4 of the THOOKS data structure.

On entry to MTOENB:

Register	Contents
R3	TCB address

All registers are preserved across the call.

The following is sample code for the call:

```
        MOV     TCBADX,R3           ; R3 -> TCB
        CALL    @MTOENX             ; Enable output interrupts
        .
        .
        .
TCBADX: .BLKW                       ; Points to TCB
MTOENX: .BLKW                       ; Points to MTOENB in monitor (from
                                    ;  THK.OE in THOOKS data structure)
```

The handler is entered asynchronously as the interface can accept output characters.

4.4.1.3 Terminal Line Break Routine, MTYBRK

The handler calls MTYBRK to set or reset the BREAK signal on the specified serial line.

MTYBRK is called through offset 6_8 of the THOOKS data structure.

On entry to MTYBRK:

Register	Contents
R0	Low bit indicates the desired BREAK signal status. If low bit is clear, CLEAR the BREAK. If low bit is set, SET the BREAK.
R3	TCB address

All registers are preserved across the call.

```
        BREAK$  = 000001
        .
        .
        .
        MOV     #BREAK$,R0              ; Prepare to assert BREAK
;       CLR     R0                      ; Prepare to deassert BREAK
        MOV     TCBADX,R3               ; R3 -> TCB
        CALL    @MTYBRX                 ; Set BREAK as desired
        .
        .
        .
TCBADX: .BLKW                           ; Points to TCB
MTYBRX: .BLKW                           ; Points to MTYBRK in monitor
                                        ;  (from THK.BK in THOOKS data
                                        ;  structure)
```

4.4.1.4 Terminal Modem Control Routine, MTYCTL

The handler calls MTYCTL to set or reset certain EIA control signals on a particular serial port. Calling MTYCTL is appropriate only for ports for which REMOTE (modem) support has been specified during SYSGEN.

Set HANMC$ (bit 8) in the TCB T.STAT word before calling MTYCTL.

MTYCTL is called through offset 10_8 of the THOOKS data structure.

The following control signals can be set or reset on the specified controllers:

Controller	Signal	Bit Mask
DL, DH, DZ	DTR	000002

Controller	Signal	Bit Mask
DL, DH	RTS	000004

On entry to MTYCTL:

Register	Contents
R0	Bit pattern (mask) to be set for line
R3	TCB address

On return from MTYCTL:

Register	Contents
R0	Current status of EIA control signals
R3	TCB address
R1,R2,R5	Preserved across the call

The following is example code for the call:

```
        MOV     #bits,R0            ; R0 = EIA signals to assert
        MOV     TCBADX,R3           ; R3 -> TCB
        CALL    @MTYCTX             ; Set EIA signals as desired
        .
        .
        .
TCBADX: .BLKW                       ; Points to TCB
MTYCTX: .BLKW                       ; Points to MTYCTL in monitor
                                    ;  (from THK.CT in THOOKS data
                                    ;  structure)
```

4.4.1.5 Terminal Modem Status Routine, MTYSTA

The handler calls MTYSTA to obtain the status of certain EIA control signals of a particular serial port. Calling MTYSTA is appropriate only for ports for which REMOTE (modem) support has been specified during SYSGEN.

MTYSTA is called through offset 12_8 of the THOOKS data structure.

The status of the following control signals can be returned if the signal is supported by the interface:

Signal	Bit Mask	Meaning
DLRI$	040000	Ring indicator
DLCTS$	020000	Clear to send

Signal	Bit Mask	Meaning
DLDCD$	010000	Data carrier detect
DLRIE$	000100	Receiver interrupt enable
DLTIE$	000100	Transmitter interupt enable
DLRTS$	000004	Request to send
DLDTR$	000002	Data terminal ready

On entry to MTYSTA:

Register	Contents
R3	TCB address

On return from MTYSTA:

Register	Contents
R0	Current status of EIA control signals as described above
R3	TCB address
R1,R2, R4,R5	Preserved across the call

The following is example code for the call:

```
        MOV     TCBADR,R3           ; R3 -> TCB
        CALL    @MTYSTX             ; Obtain current EIA signals
        .
        .
        .
TCBADR: .BLKW                       ; Points to TCB
MTYSTX: .BLKW                       ; Points to MTYSTA in monitor
                                    ;  (from THK.ST in THOOKS
                                    ;  data structure)
```

4.4.2 Connecting a Serial Interface Printer Handler (LS)

Through the SYSGEN procedure, you can build an LS handler for connection to any LUN in a multiterminal system. Any serial line can be used as the printer port.

The monitor must be built with the terminal hooks option and the handler must also be built with multiterminal hooks support. During system generation a LUN can be specified as the default multiterminal line number for the handler. The default can be overridden with the command, SET LS LINE=n, before the handler is loaded. See the LS section of the *RT–11 Device Handlers Manual* for information.

The handler hooks option for LS can be useful in many applications. For example, you can use a hardcopy boot-time console as the system printer by setting LS to the

console DL (SET LS LINE=0). You could assign a video terminal at LUN1 as the console by issuing SET TT CONSOL=1. When you finish the printing job, you can unload LS and (only these circumstances) reassign the hardcopy terminal to LUN0 by issuing SET TT CONSOL=0.

4.4.3 Connecting a Serial Communications Handler (XL)

Through the SYSGEN procedure, you can build an XL handler for connection to any LUN in a multiterminal system. Any serial line can be used as the communications port.

The monitor must be built with the terminal hooks option and the handler must be built with multiterminal serial line support. During system generation a LUN can be specified as the default multiterminal line number for the handler. The default can be overridden with the command, SET XL LINE=n, before the handler is loaded. See the *RT–11 Device Handlers Manual* for information.

4.5 Using Two or More Terminals

The following sections discuss the methods you can use to support two or more terminals on your multiterminal system. Section 4.15 describes methods you can use to support two terminals on a system that has not been generated for multiterminal support. The methods in Section 4.15 pertain only to systems that contain more than one DL controller and make a system on which you can use only one terminal at any time. If you want to use more than one terminal at the same time, you must build a multiterminal monitor.

4.5.1 A Separate Terminal for Each Job

Once you perform a system generation for the multiterminal feature, you can easily establish private consoles for up to eight jobs. However, you must be running a multi-job monitor with the system job feature in order to support more than two jobs.

As Section 4.3 describes, simply use the FRUN/TERMINAL:n or SRUN /TERMINAL:n commands to start foreground and system jobs, and assign them to private consoles. You need not use any multiterminal programmed requests to do this. Remember that each console is truly private—no two jobs can share terminals through the FRUN or SRUN /TERMINAL:n mechanism.

Each job can attach its own console terminal and issue subsequent multiterminal programmed requests.

4.5.2 Multiterminal Applications

Some applications need to take advantage of RT–11's multiterminal feature by using the programmed requests to manage more than one terminal for each job. In these applications, one program controls several terminals. Jobs that must control more than one terminal use the multiterminal data structures and programmed requests.

4.6 Introduction to Multiterminal Programmed Requests

It is not difficult for a program to use more than one terminal in a multiterminal system. Table 4–2 summarizes the actions a program may need to take in order to use a terminal in addition to its own console terminal. It also lists the appropriate procedures for the program to follow. Familiarize yourself with the procedures and the corresponding programmed requests. The *RT–11 Programmer's Reference Manual* provides detailed information on the format of each programmed request. Study this information before you attempt to write a multiterminal application program.

Table 4–2: Summary of Activities for a Program in a Multiterminal System

Activity	Procedure to Follow
Obtain the status of a multiterminal system.	Use the .MTSTAT programmed request.
Acquire a terminal.	Use the .MTATCH programmed request to attach the terminal and dedicate it to this program. As part of its startup procedure, a program usually attaches all the terminals it needs. Note that only one job can attach a shared console, and only the terminal's owner can issue multiterminal programmed requests for it. However, all the jobs sharing the background console can issue .TTYIN, .TTYOUT, .CSIGEN, .CSISPC, .GTLIN, and .PRINT requests for it, as well as .READ and .WRITE requests for TT. To detect status changes without issuing a programmed request, examine the AST word for each terminal.
Examine the characteristics of each attached terminal.	Use the .MTGET programmed request.
Change terminal characteristics if necessary.	Use the .MTSET programmed request.
Get a character from a terminal and wait for it.	Use the .MTIN programmed request.
Get a character from a terminal; do not wait for it.	Use .MTSET to set the status word, then use the .MTIN programmed request. (You need issue the .MTSET only once.)
Send a character to a terminal and wait for it.	Use the .MTOUT programmed request.

Table 4–2 (Cont.): Summary of Activities for a Program in a Multiterminal System

Activity	Procedure to Follow
Send a character to a terminal; do not wait for it.	Use .MTSET to set the status word, then use the .MTOUT programmed request. (You need issue the .MTSET request only once.)
Send a line to a terminal; wait until it prints.	Use the .MTPRNT programmed request.
Reset CTRL/O for a terminal, enabling output.	Use the .MTRCTO programmed request.
Relinquish ownership of a terminal so that another job can use it.	Use the .MTDTCH programmed request.

4.7 Multiterminal Data Structures

The following sections describe the two important data structures for multiterminal systems: terminal control blocks, and asynchronous terminal status words.

4.7.1 Terminal Control Block (TCB)

RT–11 creates one terminal control block, called a TCB, for each terminal you describe at system generation time. Each TCB located in RMON contains terminal characteristics, terminal status, and the input and output ring buffers and pointers for the terminal. The length of a TCB varies, depending on the special features you select through system generation. Note, though, that the first 20 decimal words in each TCB are fixed.

4.7.1.1 Format

Figure 4–3 illustrates the format of the TCB; Table 4–3 describes its contents. The size, offset, or existence of the data structure elements following I.ITOP depend on the special features you select through system generation.

Figure 4–3: Format of the Terminal Control Block (TCB)

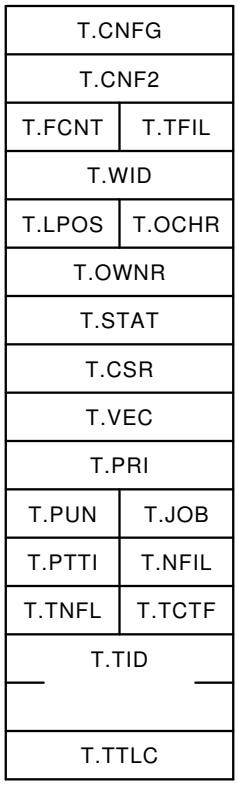


Figure 4–3 (continued on next page)

Figure 4–3 (Cont.): Format of the Terminal Control Block (TCB)

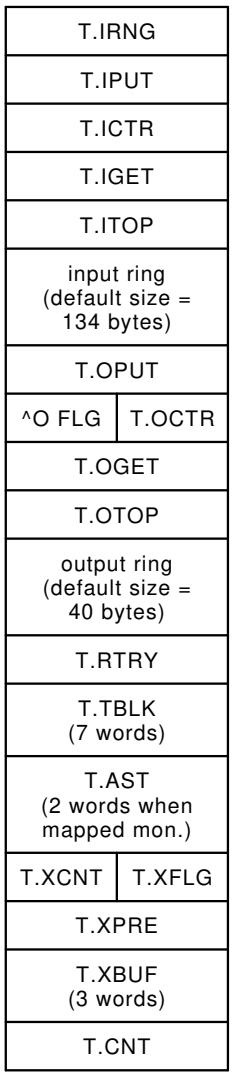


Table 4–3: Contents of the Terminal Control Block (TCB), .TCBDF

Offset	Name	Description
0	T.CNFG	The terminal configuration word. A program and the monitor communicate with each other about terminal characteristics through the .MTGET and .MTSET programmed requests. These requests use a four-word status block within the program to store terminal information. The first word, M.TSTS, has the same structure as T.CNFG. Table 4–4 describes the meaning of each bit in T.CNFG.

Table 4–3 (Cont.): Contents of the Terminal Control Block (TCB), .TCBDF

Offset	Name	Description
2	T.CNF2	The second terminal configuration word. The structure of this word is the same as that of M.TST2, the second word of the four-word status block for .MTGET and .MTSET programmed requests. Table 4–5 describes the meaning of each bit in T.CNF2.
4	T.TFIL	Contains the character after which this terminal requires one or more fill characters. The counterpart of this byte in the four-word status block for .MTGET and .MTSET programmed requests is called M.TFIL.
5	T.FCNT	Contains the number of fill characters that this terminal requires. The counterpart of this byte in the four-word status block for .MTGET and .MTSET programmed requests is called M.FCNT.
6	T.WID	Contains the carriage width of this terminal. The counterpart of this word in the four-word status block for .MTGET and .MTSET programmed requests is called M.TWID. The maximum value is 255 decimal.
10	T.OCHR	Contains the character to output.
11	T.LPOS	Contains the current carriage position for this terminal.
12	T.OWNR	A pointer to the impure area of the job that currently owns this terminal. This word has a value when this terminal is a private console for a job, or, when it is a shared console and one job has attached it. This word is 0 when this terminal is a shared console and no job has attached it, or when it is not a console and no job has attached it. Alternately, if the HANMT$ bit is set in the T.STAT word (offset 14), T.OWNR is a pointer to the multiterminal hook routine in a handler.
14	T.STAT	Contains the terminal status. Table 4–6 describes the meaning of each bit in T.STAT.
16	T.CSR	Contains the CSR for the keyboard of this terminal. It is 0 if the bootstrap could not find the CSR; this makes the LUN unusable.
20	T.VEC	Contains the first interrupt vector for this terminal.
22	T.PRI	Contains the device interrupt priority.
24	T.JOB	Contains the job number of the job that currently owns this terminal.
25	T.PUN	Contains the physical unit number of this terminal. This value is always 0 for terminals on DL interfaces. For terminals on DZ interfaces, the value ranges from 0 through 7 (0 through 3 for DZV11s). For DH interfaces, the value ranges from 0 through 15_{10}.
26	T.NFIL	Active fill character counter. This byte contains the number of nulls left to print.
27	T.PTTI	Contains the last character typed on the terminal keyboard.

Table 4–3 (Cont.): Contents of the Terminal Control Block (TCB), .TCBDF

Offset	Name	Description
30	T.TCTF	Contains the special fill character. (For example, a space is the special fill character for a tab, and a line feed is the special fill character for a form feed.)
31	T.TNFL	Contains the count for the special fill character. The value is stored as a negative number.
32	T.TID	A pointer to the terminal identification prompt string, which contains the job name, and is used only when the monitor is actually printing an identification. It is 0 at all other times.
34	—	PAR address in mapped systems; else reserved.
36	T.TTLC	Contains the terminal line count (the number of lines in the input buffer).
40	T.IRNG	A pointer to the first byte of the input ring buffer. (For more information on ring buffers, see Chapter 2.)
42	T.IPUT	Input PUT pointer.
44	T.ICTR	Input character count.
46	T.IGET	Input GET pointer.
50	T.ITOP	Indicates the top of the input ring buffer. This word points to the byte just beyond the high limit of the buffer.
52	—	Input ring buffer. Its length is determined at system generation time. It is TTYIN bytes long. (Default is 134_{10})
	T.OPUT	Output PUT pointer.
	T.OCTR	Output character count.
	—	CTRL/O flag. A value of 0 means CTRL/O is not in effect; a value of 377_8 means that CTRL/O is in effect.
	T.OGET	Output GET pointer.
	T.OTOP	Indicates the top of the output ring buffer. This word actually points to the byte just beyond the high limit of the buffer.
	—	Output ring buffer. Its length is determined at system generation time. It is TTYOUT bytes long. (Default is 40_{10})
	T.RTRY	Present if device time-out support or support for modems was selected at system generation time. This word contains the retry count for output.
	T.TBLK	Present if device time-out support or support for modems was selected at system generation time. This seven-word area is the time-out block for this terminal.

Table 4–3 (Cont.): Contents of the Terminal Control Block (TCB), .TCBDF

Offset	Name	Description
	T.AST	Present if the asynchronous terminal status word was selected at system generation time. This word is a pointer to the AST word. In mapped systems, the AST pointer is followed by a second word that contains a PAR1 value for mapping to the AST word.
	T.XFLG	Present if the system job feature was selected at system generation time. If this flag byte is nonzero, it indicates that a CTRL/X sequence is in progress.
	T.XCNT	Present if the system job feature was selected at system generation time. This byte contains the number of characters typed in a CTRL/X sequence.
	T.XPRE	Present if the system job feature was selected at system generation time. This word contains the previous character typed on the terminal keyboard.
	T.XBUF	Present if the system job feature was selected at system generation time. This three-word area contains the characters typed as part of a CTRL/X sequence.
	T.CNT	Present if the system job feature was selected at system generation time. This word contains the number of jobs that are sharing the background console.

Table 4–4: Terminal Configuration Word, T.CNFG (.TCFDF)

Bit	Name	Meaning
0	HWTAB$	Hardware tab bit. When set, it indicates that this terminal has hardware tab support. The monitor does not convert a tab character to spaces before sending it to the output ring buffer. Your program can set this bit for a particular terminal through the .MTSET programmed request (described in Section 4.8.3). The SET TT: TAB command sets this bit for the background console.
1	CRLF$	When this bit is set, the monitor sends a carriage return/line feed combination to the terminal when its carriage width is exceeded. Your program can set this bit for a particular terminal through the .MTSET request. The SET TT: CRLF command sets this bit for the background console.
2	FORM$	Hardware form feed bit. When set, it indicates that this terminal has hardware form feed support. The monitor does not convert a form feed character to line feeds before sending it to the output ring buffer. Your program can set this bit for a particular terminal through the .MTSET programmed request. The SET TT: FORM command sets this bit for the background console.

Table 4–4 (Cont.): Terminal Configuration Word, T.CNFG (.TCFDF)

Bit	Name	Meaning
3	FBTTY$	When this bit is clear, the monitor treats CTRL/F, CTRL/B, and CTRL/X as ordinary characters and ignores their special meanings. The SET TT: NOFB command clears this bit for the background console. Your program cannot set this bit for other terminals; only the shared console can use it.
4-5	——	Reserved.
6	TCBIT$	The inhibit TT wait bit. It is similar to bit 6 in the Job Status Word, which a program can set. When this bit is set, the program does not wait for I/O to complete on the terminal before execution continues. Note that bit 6 in the JSW affects only the job's current console; it does not affect any other terminals attached to this job. If the program uses other terminals for I/O, it can set this bit in each TCB by using the .MTSET programmed request. If this terminal is a private console for this job, the job can set bit 6 in the JSW. In a multiterminal application, the job can set bit 6 in either the JSW or in the TCB for the console terminal. In any case, setting bit 6 in one place (the TCB or the JSW) results in both bits being set. Be sure and issue a .CRTLO request after modifying the JSW.
7	PAGE$	The XON/XOFF bit. When set, it enables recognition of the XON (CTRL/Q) and XOFF (CTRL/S) characters. The SET TT: PAGE command sets this bit for the background console. (See Chapter 2 for more information on XON/XOFF processing.)

Table 4–4 (Cont.): Terminal Configuration Word, T.CNFG (.TCFDF)

Bit	Name	Meaning
8-11	LINSP$	The baud rate mask for terminals on DH or DZ lines. (The baud rate for terminals on DL lines is not programmable through the .MTSET request.) Notice that the values for masks 5400 and 7400 are different between DZ and DH interfaces. The values are as follows:

Mask	DZ Rate	DH Rate
0000	50	50
0400	75	75
1000	110	110
1400	134.5	134.5
2000	150	150
2400	300	300
3000	600	600
3400	1200	1200
4000	1800	1800
4400	2000	2000
5000	2400	2400
5400	3600	38400
6000	4800	4800
6400	7200	7200
7000	9600	9600
7400	not used	19200

There are baud rate restrictions for DH; see the appropriate DH interface technical manual for information.

Table 4–4 (Cont.): Terminal Configuration Word, T.CNFG (.TCFDF)

Bit	Name	Meaning
12	TTSPC$	The special mode bit. It is similar to bit 12 in the Job Status Word, which affects the job's console. If this terminal is a private console for this job, the job can set bit 12 in the JSW to enable special mode. In a multiterminal application, the job can set bit 12 in either the JSW or in the TCB for the console terminal. In any case, setting bit 12 in one place (the TCB or the JSW) results in both bits being set. (See the description of .TTYIN in the *RT–11 System Macro Library Manual* for more information on special mode.) Be sure to issue a .MTRCTO request after modifying the JSW. If the program uses other terminals for I/O, it can set this bit in each TCB by using the .MTSET programmed request.
13	REMOT$	The remote terminal bit. It is read-only, and your program cannot alter it. When set, this bit indicates that this terminal is remote.
14	TTLC$	When this bit is set, lower- and upper-case typing is enabled. When this bit is clear, the monitor converts all typed characters to upper-case. If this terminal is a private console for this job, the job can set bit 14 in the JSW. In a multiterminal application, the job can set bit 14 in either the JSW or in the TCB for the console terminal. In any case, setting bit 14 in one place (the TCB or the JSW) results in both bits being set. Be sure to issue a .MTRCTO request after changing any bits in the JSW.
15	BKSP$	When this bit is set, the monitor takes the appropriate action for a video terminal when the DELETE key is pressed. Your program can set this bit for a particular terminal through the .MTSET programmed request. The SET TT: SCOPE command sets this bit for the background console.

Table 4–5: Second Terminal Configuration Word, T.CNF2 (.TC2DF)

Bit	Name	Meaning
0-1	CHRLN$	These two bits indicate the length of a character. The DZ and DH interfaces can transmit characters that are five, six, seven, or eight bits long. The values are as follows:

Value	Character Length
00	5 bits
01	6 bits
10	7 bits
11	8 bits

These bits are unused for DL interfaces.

Table 4–5 (Cont.): Second Terminal Configuration Word, T.CNF2 (.TC2DF)

Bit	Name	Meaning
2	USTOP$	Unit stop bit. Depending on the speed, it indicates the number of stop bits to send. The values are as follows:

Value	Stop Bits
0	Send one stop bit
1	Send two stop bits (one and one-half stop bits if five-bit characters are used)

This bit is unused for DL interfaces.

3	PAREN$	The parity enable bit. When set, it enables parity checking.
4	ODDPR$	Indicates whether parity checking will be odd or even. The values are as follows:

Value	Parity
0	Even parity
1	Odd parity

This bit is unused for DL interfaces.

5-6		Reserved.
7	RPALL$	When set, this bit indicates *read pass-all* mode. In this mode, RT–11 passes (on input) all eight bits of each character without interpreting or echoing the characters. This feature is often referred to as *transparency.* For example, it passes CTRL/C as 203 in read pass-all mode if the terminal sets the high bit upon transmission. When RPALL$ is set, the terminal is implicitly in single-character mode.
8-14		Reserved.
15	WPALL$	When set, this bit indicates write pass-all mode. In this mode, RT–11 passes (on output) all eight bits of each character without interpreting the characters.

Table 4–6: Terminal Status Word, T.STAT (.TSTDF)

Bit	Name	Meaning When Set
0	FILL$	Indicates that a fill sequence is in progress.
1	CTRLU$	CTRL/U in progress.
2	DHOIP$	Output in progress on DH controller.
3	HANMT$	LUN owned by a device handler.
4	DTACH$	Indicates that a detach operation is in progress. Input from the terminal is ignored.
5	WRWT$	This is the TT handler synchronization bit.
6	INEXP$	Indicates that an output interrupt is expected.
7	PAGE$	Indicates that the terminal has sent XOFF to request suspension of output.
8	HANMC$	Indicates that the handler intends to control the modem signals (defined only for MTTY systems with HANMT$ set.) If HANMC$ is clear, MTTY monitor controls modem signals.
9		Reserved.
10	SHARE$	Indicates that this terminal is the shared console.
11	HNGUP$	Indicates that the remote terminal has hung up.
12	DZ11$	Indicates that the terminal interface is a DZ.
13	DH11$	Indicates that the terminal interface is a DH. If bits 12 and 13 both clear, indicates terminal interface is a DL. Bits 12 and 13 both set is reserved.
14	CTRLC$	Indicates that two CTRL/C characters were typed at this terminal. This bit is reset by .MTGET.
15	CONSL$	Indicates that this terminal is a console for some job. It can be shared or private.

4.7.1.2 Patching a TCB

You can use SIPP to make binary patches to the terminal control blocks in your monitor file, *monitr*.SYS. The TCBs are located in p-sect MTTY$, which you can find on your monitor link map. They appear in the same order in which SYSGEN assigned physical units to logical unit numbers at system generation time (see Section 4.2). The first TCB is for LUN 0; it starts at the label DLTCB::. The TCBs are all the same size; the value of T.CBSZ is their length.

4.7.2 Asynchronous Terminal Status (AST) Word

Support for the asynchronous terminal status (AST) word is a special feature that you can select at system generation time. If you select this feature, you can set aside space for one AST word per LUN in your own program. (You can check for AST support from within your program by checking if the CF3.AS bit (bit 13 in RMON fixed offset CONFG3) is set.) Then, when you issue the .MTATCH programmed request to attach a terminal to your job, you specify as an argument the address of the AST word for that terminal.

The purpose of the AST word is to monitor each terminal's line so that the program can obtain certain information without issuing a programmed request. RT–11 sets or clears bits in the AST word asynchronously, as significant events occur. The AST word contains information on whether:

- Input is available from the terminal
- The terminal's output ring buffer is empty
- Double CTRL/C was typed on the terminal
- A remote line just dialed in or just hung up

Table 4–7 shows the event flags in the AST word and their meaning. Unused bits are reserved for future use by Digital.

Table 4–7: Asynchronous Terminal Status (AST) Word (.TASDF)

Bit	Name	Bit Pattern	Meaning When Set
15	AS.CTC	100000	Double or multiple CTRL/C was typed on this terminal. You must reset this bit; the monitor never turns it off.
14	AS.INP	40000	Input is available from this terminal.
13	AS.OUT	20000	The output ring buffer is empty.
7	AS.CAR	200	Carrier is present (for remote lines only).
6	AS.HNG	100	This remote line just hung up and RT–11 dropped it.

The monitor sets bit 15, AS.CTC, whenever two or more consecutive CTRL/Cs are typed on any terminal. Typing two CTRL/Cs on a job's console terminal always aborts the job, unless the job already issued the .SCCA programmed request to intercept the characters. The job must reset this bit before it continues processing.

The monitor sets bit 14, AS.INP, when input is available from the terminal. It can be a line of characters in normal mode, or a single character in special mode. The monitor clears this bit when the program reads the characters.

The monitor sets bit 13, AS.OUT, when the terminal's output ring buffer is empty. This occurs after the last character in the ring buffer is printed on the terminal. The monitor clears this bit when there are characters in the ring buffer.

The monitor sets bit 7, AS.CAR, when it answers a remote line. It clears this bit when the remote line hangs up or drops carrier. *Carrier* is a tone transmitted over the remote line. It carries information through its modulation.

The monitor sets bit 6, AS.HNG, when it drops a remote line that just hung up.

4.8 Using the Multiterminal Programmed Requests

The routines in MTTEMT, which are part of RMON, dispatch the multiterminal programmed requests and process them. The multiterminal programmed requests are more fully described in the *RT–11 System Macro Library Manual*.

The dispatch routine accepts programmed requests that translate into EMT 375 instructions with a subcode of 37 and a function code in the range 0 through 10 octal. The dispatch routine first checks to see if the programmed request is a valid one. Then it verifies the logical unit number and makes sure that the terminal is installed. If the programmed request is for an attach operation, the dispatch routine verifies that the terminal is not already attached. For all other requests, the dispatch routine verifies that the terminal is attached to the calling program.

If the request passes all the checks in the dispatch routine, control passes to the EMT processing code for the individual request.

4.8.1 Attaching a Terminal: .MTATCH

Issue the .MTATCH programmed request to attach a terminal to your job. This permits your program to print characters on the terminal, get characters from it, and alter its characteristics.

The attach routine first checks to see if the terminal is the shared console, but not this job's console. If so, the routine issues error code 4. If the terminal is already attached to another job, the routine also issues error code 4. Finally, the routine checks if the terminal is attached to a handler. If the terminal (serial line) is attached to a device handler, the routine returns error code 6 and R0 contains the RAD50 handler name. The status information is always placed in the status block in your program even when an error is returned from this operation.

The routine attaches the terminal by setting up certain TCB offsets for this terminal. The routine always stores a pointer in the T.OWNR word to the owning job's impure area. In multi-job systems, it also stores the job number in the T.JOB byte.

If AST support is part of the system, the routine puts a pointer to the AST word in T.AST. In mapped systems, it also stores a value in T.AST + 2 to be used as a PAR1 value in mapping to the AST word.

The routine next moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies TTLC$ (bit 14 for lower case), TTSPC$ (bit 12 for special mode), and TCBIT$ (bit 6 for wait inhibit). If the terminal is the background console, T.TFIL is loaded from location 56.

When a job attaches a terminal, the terminal remains attached until the job issues a .MTDTCH request, or until the job exits or aborts.

- If the terminal is detached through the .MTDTCH request, the job is blocked until output in process for the terminal finishes and the monitor detaches the terminal.
- If the terminal is detached through a job exit, the actions taken are determined by the type of monitor and the type of terminal. A single-job monitor detaches a console terminal immediately (no wait for output to finish); with other terminal types, the monitor waits for output to finish.

  A multijob monitor detaches a shared terminal immediately (no wait for output to finish); with other terminal types, the monitor waits for output to finish.
- If the terminal is detached when the job aborts, the output terminates and the monitor detaches the terminal immediately.

4.8.2 Getting Terminal Status: .MTGET

Issue the .MTGET programmed request to obtain the status of a terminal. (The terminal need not be attached to your program in order to obtain the status.)

The .MTGET routine moves information from the TCB to the status block in your program. The following transfers occur:

T.CNFG to M.TSTS
T.CNF2 to M.TST2
T.TFIL to M.TFIL
T.FCNT to M.FCNT
T.WID to M.TWID
High byte of T.STAT to M.TSTW

Then, if the terminal is not attached to any job, the routine returns error code 1. If the terminal is attached, but not to this job, the routine returns error code 4 and R0 contains the job number of the terminal's owner. If the terminal is the shared console, but the job has its own private console, R0 contains the job's own job number. If the terminal (serial line) is attached to a device handler, the routine returns error code 6 and R0 contains the RAD50 handler name. The status information is always placed in the status block in your program even when an error is returned from this operation.

Finally, if no error was returned, the routine clears bit 14 (CTRL/C) in T.STAT.

4.8.3 Setting Terminal Characteristics: .MTSET

Issue the .MTSET programmed request to set the characteristics of a terminal. If the terminal is not attached to your program, the request returns error code 1.

The request moves the contents of M.TSTS to T.CNFG, except for REMOT$ (bit 13, the remote terminal bit), which is read only in T.CNFG. If the terminal is the job's console, the routine moves some bits from T.CNFG into the JSW. It copies TTLC$ (bit 14 for lower case), TTSPC$ (bit 12 for special mode), and TCBIT$ (bit 6 for wait inhibit).

Whether or not the terminal is the job's console, the routine moves the following information:

M.TST2 to T.CNF2
M.TFIL to T.TFIL
M.FCNT to T.FCNT
M.TWID to T.WID

If DH or DZ support is part of the system and if this terminal is on a DH or DZ interface, the routine waits for any characters to finish printing on this terminal, then sets up the appropriate DH or DZ line parameters.

NOTE

Always issue an .MTGET request before an .MTSET request. Change only the fields you are interested in. For a one-bit field, use a BIS or BIC instruction to set or clear it. For a multiple-bit field, clear it first with a BIC and then use BIS to load the field. Use MOVB or MOV instructions only for byte or word fields. Changing other bits can cause unusual terminal service errors. Finally, issue the .MTSET specifying the same status block that you used for the .MTGET request.

4.8.4 Getting Characters: .MTIN

Issue the .MTIN programmed request to get one or more characters from the terminal.

The routine moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies TTLC$ (bit 14 for lower case), TTSPC$ (bit 12 for special mode), and TCBIT$ (bit 6 for wait inhibit). If the terminal is the background console, T.TFIL is loaded from location 56.

The routine gets a character from the input ring buffer and adjusts the ring buffer pointers. If the terminal is the console, the routine uses the ring buffer in the job's impure area. If the terminal is not the console, the routine uses the ring buffer in the terminal's TCB.

If the input character is CTRL/C on a console terminal, and .SCCA is not in effect, the job aborts.

4.8.5 Printing Characters: .MTOUT

Issue the .MTOUT programmed request to print one or more characters on the terminal.

The routine moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies TTLC$ (bit 14 for lower case), TTSPC$ (bit 12 for special mode), and TCBIT$ (bit 6 for wait inhibit). If the terminal is the background console, T.TFIL is loaded from location 56.

The routine moves a character from the user buffer into the output ring buffer and adjusts the ring buffer pointers. If the terminal is the console, the routine uses the ring buffer in the job's impure area. If the terminal is not the console, the routine uses the ring buffer in the terminal's TCB.

4.8.6 Printing a Line: .MTPRNT

Issue the .MTPRNT programmed request to print a string of characters on the terminal. The string can end with a null byte (to print a carriage return and a line feed at its end) or a 200 octal byte, just as in the .PRINT programmed request.

The routine moves a line from the user buffer into the output ring buffer and adjusts the ring buffer pointers. If the terminal is the console, the routine uses the ring buffer in the job's impure area. If the terminal is not the console, the routine uses the ring buffer in the terminal's TCB. If there is no room in the output ring, the job is blocked until room is available, regardless of the value of TCBIT$ (bit 6) in T.CNFG.

4.8.7 Resetting CTRL/O: .MTRCTO

Issue the .MTRCTO programmed request to enable output on a terminal even though CTRL/O may have been typed.

This routine clears the CTRL/O flag in the TCB for the terminal and moves some bits from the JSW into T.CNFG if this terminal is the job's console. It copies TTLC$ (bit 14 for lower case), TTSPC$ (bit 12 for special mode), and TCBIT$ (bit 6 for wait inhibit). If the terminal is the background console, T.TFIL is loaded from location 56.

If you ever alter the contents of the JSW, Digital recommends that your program issue the .MTRCTO request immediately afterward so that the TCB and the JSW always have the same information. In particular, if you require lower-case input for a .GTLIN request, set TTLC$ (bit 14) in the JSW and issue .MTRCTO or .RCTRLO before using .GTLIN.

4.8.8 Getting System Status: .MTSTAT

Issue the .MTSTAT programmed request to obtain status information about the multiterminal system. This request returns the following four words of information to your program:

- The offset from the start of RMON to the first TCB
- The offset from the start of RMON to the TCB of the current console terminal for this job
- The value of the LUN associated with the last TCB.
- The size of each TCB in bytes. (Note that all TCBs on a particular monitor are the same size.)

Remember that the TCBs are located in RMON in the order in which you specified interface lines to the SYSGEN dialogue. That is, the TCBs for local DLs appear first, followed by remote DLs, local DZs, remote DZs, and local and remote DHs.

With the information returned to you by .MTSTAT you can find the TCB for any terminal in the system and examine its contents with the .GVAL request. Figure 4–3 and Table 4–3 describe the contents of each TCB.

4.8.9 Detaching a Terminal: .MTDTCH

Issue the .MTDTCH programmed request to detach a terminal from your job and make it available for use by another job.

The routine first sets the DTACH$ bit (bit 4) in T.STAT to indicate that a detach operation is in progress. This avoids any race conditions in the module MTTINT. (A race condition is a situation in which two or more processes attempt to modify the same data structure at the same time; as a result, the data structure is corrupted and the integrity of the processes is compromised.) It then forces XON if XOFF had been previously set. If the terminal is not a shared console, the output buffer is then flushed. The job is blocked until T.OCTR is clear.

The words T.OWNR and T.AST are set to zero to detach the terminal. DTACH$ is finally cleared to finish the operation.

Whenever a job aborts, terminals attached to it are detached without having their buffers flushed.

4.9 The Console as a Special Case

The console terminal is always a special case for I/O in multiterminal systems. Recall that each job has input and output ring buffers and pointers, both in its console's TCB and in its impure area. Whenever a job gets characters from its console terminal, or writes characters to it, the monitor uses the set of ring buffers located in the job's impure area. In this case, the console can be the background console, if this job is sharing it, or it can be a private console, if this job has one.

For all I/O requests involving the job's console, the monitor performs the request based on the characteristics indicated in the Job Status Word rather than in the terminal configuration word. However, the monitor aligns corresponding bits in the job's console terminal configuration word with the JSW the next time the job does any I/O or a reset CTRL/O request is issued for that terminal. (See Table 4–4). To force the alignment before any I/O or reset CTRL/O request is issued, Digital recommends that you issue the .MTRCTO request immediately after altering the JSW to make sure that the contents of the JSW are duplicated in the TCB for the terminal. Similarly, if you modify the terminal configuration word with .MTSET for a job's console, the monitor also modifies the JSW.

Note that a program must issue the .SCCA programmed request to inhibit CTRL/C on its console terminal.

4.10 Interrupt Service

Terminal service in multiterminal systems is centralized in the routines contained in MTTINT. This source file is assembled and linked together with other files to become part of RMON.

In general, RT–11 services terminals in one of two ways, depending on whether the terminal is connected through a local or a remote line.

4.10.1 Local Terminals

RT–11's interrupt service routine for multiterminal systems contains the following data structures:

- Receive CSR I/O page address
- Receive data buffer I/O page address
- Transmit CSR I/O page address
- Transmit data buffer I/O page address

RT–11's interrupt service is essentially simple. The bootstrap sets the input (or receiver) interrupt enable bit; the monitor leaves it set at all times. If a character is typed on a local terminal, an interrupt occurs and the monitor picks up the character. If the terminal is not attached to any job, the character is ignored. In multiterminal systems with time-out support, the monitor ensures that the interrupt enable bit for each DL is enabled once every 30 clock ticks (every half-second).

The monitor only sets the output interrupt enable bit when it is ready to print a character. It clears the bit after the output ring buffer is empty.

4.10.2 Remote Terminals

Remote terminals are connected to RT–11 through modems (also known as data sets) and telephone lines so that someone can call up the computer and ring its data phone. When this occurs, it causes an interrupt, which the monitor recognizes. If the unit is attached, the multiterminal service routine answers the phone call and sends out carrier in response. (Carrier is a tone transmitted over the remote line that carries information through its modulation.)

The remote terminal can communicate with RT–11 through an approved protocol. When a remote terminal connects to an RT–11 system, using a modem, the following protocol should be used:

1. The remote terminal asserts Data_Terminal_Ready (DTR) to the attached modem.
2. The user places the call to the RT–11 system.
3. The modem attached to the RT–11 system answers the phone and responds with a carrier.
4. RT–11 sets a 30 second timer.

   If the modem attached to the remote terminal responds with carrier within 30 seconds, the timer is cancelled and I/O can begin.

   If the 30 second timer expires and the modem attached to the remote terminal has not responded with carrier, RT–11 disconnects the line.

Once communication has begun, RT–11 never takes the initiative to terminate the connection. It always continues to send carrier. However, there are two situations in which RT–11 does hang up on the remote line. If the terminal stops sending carrier for any reason, RT–11 waits two seconds for it to resume. When the interval expires,

RT–11 hangs up on the remote line. In the other situation, the remote terminal hangs up. RT–11 detects loss of carrier and waits two seconds before disconnecting the remote line. Special requirements for customers in the United Kingdom are met through assemblies based on the U.K. conditional being set to 1.

Remote terminals require a DL11-E, DLV11-E (or equivalent), DH, or DZ interface. In addition to the data lines required for remote terminals, the following control lines must be connected (for dial-up operations):

- Data terminal ready
- Ring indicator
- Carrier detect

A local terminal can be connected to a remote terminal interface if it is identified during system generation as a local terminal. The control lines listed above are then ignored and you can leave them unconnected.

4.11 Polling Routines

RT–11's multiterminal support includes two polling routines, which the following sections describe. (The DH interface does not require polling.)

4.11.1 Time-Out Routine for DL Terminals

You can select the time-out polling routine as a special feature at system generation time. It is an example of the device time-out feature that is available to application programs through the .TIMIO programmed request. RT–11 executes this routine once every half second. Its purpose is to periodically reenable the I/O interrupt enable bits on DL interfaces so that noise on a line or local static electricity cannot seriously affect transmissions.

The polling routine examines each DL line on the system every half second. It turns on the line's input interrupt enable bit and, if the line is remote, its modem interrupt enable bit. Then, if output is pending with no output interrupt, it turns the output interrupt enable bit off and then on, to force an output interrupt on the line. (Depending on the hardware failure that caused the loss of the output interrupt, this may occasionally cause a character to be repeated.)

The last thing the time-out routine does is schedule itself to run again.

4.11.2 DZ Remote Line Polling Routine

The DZ polling routine polls the terminals connected to the system through DZ interfaces. It is necessary because these terminals do not interrupt when their status changes.

The remote line polling routine schedules a mark time request. It waits 30 seconds after the data set rings to detect carrier. If there is no carrier after the required amount of time, the routine disconnects the remote line. The routine takes similar action on line errors and lost carrier. This routine is automatically included in the multiterminal service for remote DZ lines.

4.12 Restrictions

The restrictions that apply to systems with the multiterminal special feature are listed in the *RT–11 System Release Notes*.

4.13 Debugging a Multiterminal Application

Use VDT, the Virtual Debugging Technique, to debug a multiterminal application. See the *RT–11 System Utilities Manual* for more information on VDT.

4.14 Multiterminal Example Program

Figure 4–4 shows a program that uses the multiterminal programmed requests.

Figure 4–4: Multiterminal Example Program

```
.TITLE MTYSET.MAC - Auto-baud and Initialize DEC Terminals
.IDENT /X05.01/

;                       COPYRIGHT 1989, 1990, 1991 BY
;           DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
;                         ALL RIGHTS RESERVED
;
;THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
;ONLY  IN  ACCORDANCE  WITH  THE TERMS  OF  SUCH  LICENSE AND WITH THE
;INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY OTHER
;COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
;OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS HEREBY
;TRANSFERRED.
;
;THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT NOTICE
;AND  SHOULD  NOT  BE  CONSTRUED AS  A COMMITMENT BY DIGITAL EQUIPMENT
;CORPORATION.
;
;DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF ITS
;SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
        .SBTTL  Abstract and Edit History

;+
;
; Auto-baud and Initialize DEC Terminals
;
; AUTHOR: LCP   -       10/79
;
; This program will attach all "known" terminals and
; if they are VT5x, VT1xx or LA1xx series it will determine
; at what baud rate they are set and put that information in
; their TCBs. ("Foreign" terminals will be assumed to
; be set at their correct baud rate).  As each terminal is
; "initialized", its screen will be cleared, a "sign-on"
; message will be displayed, and the terminal type and
; baud rate will be logged on the background console.
;
; Edit History:
;
; (001) 01-Jun-91 MBG   Source clean-up using macros from SYSTEM, add
;                       support for DH, fix numerous bugs.
;
;-
        .SBTTL  Macros & Definitions

        .LIBRARY "SRC:SYSTEM.MLB"

; RT-11 Programmed requests
```

Figure 4–4 (continued on next page)

Figure 4–4 (Cont.): Multiterminal Example Program

```
        .MCALL  .MTATC  .MTDTC  .MTGET
        .MCALL  .MTOUT  .MTIN   .MTPRN
        .MCALL  .MTSET  .MTSTA  .EXIT
        .MCALL  .MTRCT  .PRINT  .TTYOU
        .MCALL  .MRKT   .CMKT

        .MCALL  .ASSUM

; TCB offset and bit definitions

        .MCALL  .MGTDF  .MSTDF  .TCBDF
        .MCALL  .TCFDF  .TSTDF

        .MGTDF                          ;Define MTGET block offsets
        .MSTDF                          ;Define MTSTAT block offsets
        .TCBDF                          ;Define TCB offsets
        .TCFDF                          ;Define configuration word bits
        .TSTDF                          ;Define status word bits

; Characters

        C.LF    = 12                    ;Line feed
        C.CR    = 15                    ;Carriage return
        C.ESC   = 33                    ;Escape
        .SBTTL  Start of Program

        .ENABL  LSB

MTYSET: MOV     #STAT,R3                ;R3 -> 8-word status block for .MTSTAT
        .MTSTA  #AREA,R3                ;Get MTTY status
        MOV     MST.LU(R3),R2           ;R2 = # of highest LUN
        BEQ     MTEXIT                  ;If highest is zero (console), exit
        MOV     MST.CT(R3),R4           ;R4 = Offset to console TCB

        .Assume MST.1T EQ 0
        SUB     @R3,R4                  ;R4 = Diff from 1st TCB
        BEQ     1$                      ;No difference, so LUN 0 is console
        MOV     MST.ST(R3),R5           ;R5 = Size of TCB
        CLR     R1                      ;R1 = Quotient
DIV$:   INC     R1                      ;Divide diff by size of TCB
        SUB     R5,R4                   ; to get LUN of console
        BHI     DIV$                    ;Repeat until done...
        MOV     R1,(PC)+                ;Save console LUN...
CLUN:    .WORD  0                       ;for later reference

1$:     CMP     R2,CLUN                 ;Is this the Console?
        BEQ     4$                      ;Yes...already set up
        .MTATC  #AREA,#0,R2             ;Try to attach terminal
        BCS     MTERR1                  ;If carry set, can't!
        .MTGET  #AREA,R3,R2             ;Get terminal's status
        BCS     MTERR2                  ;Can't! (Very Bad!!!)
        BITB    #<DH11$!DZ11$>/400,M.TSTW(R3) ;Is line a DZ or DH?
        BEQ     6$                      ;Nope, assume it's a DL...

        .Assume M.TSTS EQ 0
        BIT     #<REMOT$>,@R3           ;Remote line?
        BEQ     2$                      ;Nope...
        BITB    #<HNGUP$>/400,M.TSTW(R3) ;Is it on-line?
        BNE     5$                      ;Branch if not
2$:     CALL    TSETUP                  ;Figure out baud rate
                                        ; and terminal type
3$:     .MTRCT  #AREA,R2                ;Reset CTRL/O
        .MTPRN  #AREA,#HELLO,R2         ;Clear screen (if CRT)
                                        ; and say hello...
        .MTDTC  #AREA,R2                ;Release it since we're done
        CALL    LOGLUN                  ;Log terminal ID on console
4$:     DEC     R2                      ;Are we finished?
        BPL     1$                      ;No...go do another LUN
MTEXIT: .EXIT                           ;We're done...exit
        .SBTTL  Terminal ID Log Routines, Error Routines

5$:     .PRINT  #OFFLIN                 ;Log terminal offline
        BR      8$                      ; and list LUN
```

Figure 4–4 (continued on next page)

Figure 4–4 (Cont.): Multiterminal Example Program

```
6$:
        .Assume M.TSTS EQ 0
        BIS     #<TTSPC$!TCBIT$>,@R3    ;DL11 - Set terminal special mode
                                        ; and inhibit terminal wait
        MOV     #ENDTBL,R4              ;Don't know speed...
        MOV     #32,LOTIM               ;Magic # for baud rate determination,
                                        ; reflects worse case situation,
                                        ; 50 baud rate on DL lines
        CALL    TERMID                  ;Attempt to determine terminal type
        CALL    RSET                    ;Set new status...
        BR      3$                      ;Merge...

LOGLUN: .PRINT  #ATMSG                  ;Print 1st part of log
        CALL    PRNLUN                  ;Print LUN
        .PRINT  R1                      ; the terminal ID,
        .PRINT  #TINIT                  ; and
        .PRINT  R4                      ; baud rate
        RETURN

PRNLUN: MOV     R2,R0                   ;Copy LUN into R0
        SWAB    R0                      ;Put it in high byte
7$:     ADD     #<-10.*400>+1,R0        ;Divide by 10 with
                                        ;repeated subtracts
        BPL     7$                      ;Q= Q-10, R= R+1 until
                                        ;overflow (V set)
        ADD     #'0*400+'0+<10.*400-1>,R0 ;Correct
                                        ;Q & R then ASCIIfy...
        .TTYOUT                         ;Print Q...
        SWAB    R0                      ;R to low byte...
        .TTYOUT                         ;Print it...
        RETURN

MTERR1: .PRINT  #MSG1                   ;Log attach error
        BR      8$                      ;Merge

MTERR2: .PRINT  #MSG2                   ;Log get status error
8$:     CALL    PRNLUN                  ;Include LUN
        .PRINT  #CRLF
        BR      4$                      ;Try next LUN
        .SBTTL  Main Terminal Setup Subroutine

TSETUP: MOV     #SPTABL-2,R4            ;R4 => Baud rate table

        .Assume M.TSTS EQ 0
        MOV     @R3,MSTAT               ;Save old status...

        .Assume M.TSTS EQ 0
        BIS     #<TTSPC$!TCBIT$>,@R3    ;Set special bits
10$:    TST     (R4)+                   ;R4 => Next table entry

        .Assume M.TSTS EQ 0
        BIC     #LINSP$,@R3             ;Clear baud rate mask
        MOV     (R4)+,R5                ;R5 = Baud from table

        .Assume M.TSTS EQ 0
        BIS     R5,@R3                  ;Set it in CONFG1
        CMP     #ENDTBL,R4              ;Are we thru table?
        BEQ     14$                     ;Yes...use as is
        MOV     #32,LOTIM               ;Magic # for .MRKT
        SWAB    R5                      ;Put mask in low byte
        SUB     R5,LOTIM                ;Subtract from magic #
                                        ; to get # ticks to wait
        CALL    TERMID                  ;Try to get terminal ID
        BCS     10$                     ;No dice...

RSET:
        .Assume M.TSTS EQ 0
        BIC     #<TTSPC$!TCBIT$>,@R3    ;Clear special bits

        .Assume M.TSTS EQ 0
        BIC     (R1)+,@R3               ;Turn off unwanted options
```

Figure 4–4 (continued on next page)

Figure 4–4 (Cont.): Multiterminal Example Program

```
        .Assume M.TSTS EQ 0
        BIS     (R1)+,@R3               ;Turn on desired options
                                        ;R1 => Terminal ID string
12$:    MOV     @R4,R4                  ;R4 => ASCII baud rate
13$:    .MTSET  #AREA,R3,R2             ;Store status
        RETURN                          ;Return to caller

14$:    CALL    GETSP                   ;Get ASCII of baud rate
        BR      13$                     ;Merge...

TERMID: .MTSET  #AREA,R3,R2             ;Set new status
        MOV     #TTLIST,R5              ;R5 => List of Terminals
15$:    MOV     (R5)+,R1                ;R1 => Terminal specific
                                        ; character sequence
        BEQ     18$                     ;End of table - leave!
        CALL    TOUT                    ;Try to communicate...
        BCS     15$                     ;Carry set = no dice
        ADD     OUTCT,R1                ;R1 => Expected response
        BIT     #1,R1                   ;Odd address?
        BEQ     16$                     ;No
        INC     R1                      ;YES! Make it even
16$:    CMP     MSGIN,(R1)+             ;Match?
        BNE     15$                     ;Nope...
        CMP     MSGIN+2,(R1)+           ;Still match?
        BNE     15$                     ;Nope...
        RETURN                          ;Return with R1 => options

18$:    MOV     #UNKTT,R1               ;R1 => "Unknown terminal"
        SEC                             ;Set carry...
        RETURN
        .SBTTL  Terminal I/O & Get Baud Rate Routines

TOUT:   MOVB    (R1)+,INCNT             ;Get 'what-are-you?' response length
        MOVB    (R1)+,OUTCT             ;Get 'what-are-you?' query length
        .MTOUT  #AREA,R1,R2,OUTCT       ;Send 'what-are-you?' query
        BCS     20$                     ;Output error
        CLRB    TFLG                    ;Clear flag
        CLR     MSGIN+2                 ;Init input buffer
        .MRKT   #AREA,#WAITM,#CRTNE,#1  ;Set time-out
19$:    TSTB    TFLG                    ;Did time-out occur?
        BEQ     19$                     ;Nope...
        .MTIN   #AREA,#MSGIN,R2,INCNT   ;Get response,
20$:    RETURN                          ; (with carry status)

GETSP:  MOV     #SPTABL,R4              ;R4 => baud rate table

        .Assume M.TSTS EQ 0
        MOV     @R3,R5                  ;R5 = TCB config word 1
        BIC     #^C<LINSP$>,R5          ;Clear all but baud rate
21$:    CMP     (R4)+,R5                ;Compare it with table
        BEQ     22$                     ;Branch if equal
        CMP     #UNKSP,(R4)+            ;End of table?
        BNE     21$                     ;Try another if not
22$:    MOV     @R4,R4                  ;R4 => ASCII baud rate
        RETURN                          ;Return to caller
        .SBTTL  Timeout Completion Routine

CRTNE:  INCB    TFLG                    ;Set time-out flag
        RETURN                          ; to mainline

; Argument blocks & working storage

INCNT:  .WORD   0                       ;Input byte count
OUTCT:  .WORD   0                       ;Output byte count
AREA:   .BLKW   5                       ;EMT argument block

WAITM:  .WORD   0                       ;Time-out argument (MINUTES)
LOTIM:  .WORD   0                       ;Lo order ticks

STAT:   .BLKW   8.                      ;Status block (8 words)
        .SBTTL  Baud Rate Mask & ASCII Baud Rate Tables

; Baud rate table - in "best guess" order
```

Figure 4–4 (continued on next page)

Figure 4–4 (Cont.): Multiterminal Example Program

```
SPTABL: .WORD   7000,B9600              ;9600  baud     ;Scopes
        .WORD   3400,B1200              ;1200  baud     ;LA120
        .WORD   2400,B300               ;300   baud     ;LA36
        .WORD   6000,B4800              ;4800  baud     ;Scopes
        .WORD   5000,B2400              ;2400  baud     ;Scopes
        .WORD   2000,B150               ;150   baud     ;LA36
        .WORD   1400,B134               ;134.5 baud     ;IBM
MSTAT:  .WORD   0                       ;Orig status
ENDTBL: .WORD   UNKSP                   ;End-of-Table

                                        ; => "Unknown baud"
MSGIN:  .BLKB   8.                      ;Response buffer
TFLG:   .BYTE   0                       ;Time-out flag
        .EVEN
        .NLIST  BEX

B134:   .ASCIZ  /134.5 Baud/
B150:   .ASCIZ  /150  Baud/
B300:   .ASCIZ  /300  Baud/
B1200:  .ASCIZ  /1200 Baud/
B2400:  .ASCIZ  /2400 Baud/
B4800:  .ASCIZ  /4800 Baud/
B9600:  .ASCIZ  /9600 Baud/
        .EVEN
        .SBTTL  Terminal ID Tables

TTLIST:                                 ;Terminal List...
        .WORD   VT100
        .WORD   VT52
        .WORD   LA120
        .WORD   LA34
        .WORD   VT55
        .WORD   0                       ;Table stopper

;  DEC terminal command sequences

VT100:  .BYTE   4,3,C.ESC,'[,'c         ;INCNT,OUTCNT,"W-A-Y" seq
        .EVEN
        .BYTE   C.ESC,'[,'?,'1          ;Response
        .WORD   CRLF$,<HWTAB$!BKSP$>    ;Undesired, Desired options
        .ASCII  / VT100/<200>           ;ASCII terminal ID
        .EVEN

VT52:   .BYTE   2,2,C.ESC,'Z
        .EVEN
        .BYTE   C.ESC,'/,0,0            ;VT52 response varies w/ model!
        .WORD   CRLF$,<HWTAB$!BKSP$>
        .ASCII / VT52 /<200>
        .EVEN

LA120:  .BYTE   4,3,C.ESC,'[,'c
        .EVEN
        .BYTE   C.ESC,'[,'?,'2
        .WORD   0,0
        .ASCII  / LA120/<200>
        .EVEN

LA34:   .BYTE   4,3,C.ESC,'[,'c
        .EVEN
        .BYTE   C.ESC,'[,'?,'3
        .WORD   0,0
        .ASCII  / LA34 /<200>
        .EVEN

VT55:   .BYTE   2,2,C.ESC,'Z
        .EVEN
        .BYTE   C.ESC,'E,0,0
        .WORD   CRLF$,<HWTAB$!BKSP$>
        .ASCII  / VT55 /<200>
        .EVEN
        .SBTTL  Message & Text Initialization String

        ;  Message Text...
```

Figure 4–4 (continued on next page)

Figure 4–4 (Cont.): Multiterminal Example Program

```
MSG1:   .ASCII  /?Cannot attach terminal LUN:/<200>
MSG2:   .ASCII  /?Status error - LUN:/<200>
ATMSG:  .ASCII  /Attaching LUN:/<200>
TINIT:  .ASCII  / initialized at /<200>
UNKSP:  .ASCIZ  /unknown baud rate/
UNKTT:  .ASCII  / unidentifiable/<200>
OFFLIN: .ASCII  /Terminal offline - LUN:/<200>
CRLF:   .ASCIZ  //

;  Clear screen & say hello character string...

HELLO:  .ASCII  <C.ESC>"[2J"            ;VT100 Erase screen
        .ASCII  <C.ESC>"\"              ;VT52 "Exit hold screen mode"
        .ASCII  <C.ESC>"H"<C.ESC>"J"    ;VT52 HOME & "Erase-to-End-of Screen
        .ASCII  <C.CR><C.LF>            ;CRLF (for hardcopy)
        .ASCIZ  /TERMINAL INITIALIZED/
        .EVEN

        .END    MTYSET                  ;End of program
```

4.15 Using Two or More Terminals Without the Multiterminal Feature

The following sections describe methods you can use to support two terminals on a system that has not been generated for multiterminal support. The methods pertain only to systems on which there is more than one DL controller and no more than one terminal is to be available at any one time. If your system includes one DL controller and either DH or DZ controllers, you must use a built multiterminal monitor, as described in Section 4.5.

There are several situations in which you may need to use more than one terminal, but you do not need any of the special features available through the multiterminal programmed requests. The following sections describe some of those situations and show how to arrange the terminals.

4.15.1 A Video Console Terminal and a Hardcopy Printing Terminal

A typical situation that arises in RT–11 applications is the case in which it is desirable to use a video terminal as the background console terminal and a hardcopy terminal as a line printer. The next two sections describe the procedures to use, depending on whether the video terminal or the hardcopy terminal is the boot-time console.

4.15.1.1 The Video Terminal Is the Boot-Time Console

If your video terminal is the boot-time console, it is simple to use a hardcopy printing terminal as a line printer. (Note that the hardcopy terminal must be on a DL interface to use this procedure.) You set up the vectors and CSR addresses for the hardcopy terminal in the LS device handler file (by using the SET LS: commands described in the *RT–11 Commands Manual*) and install LS. You can then simply assign LP to LS and proceed to use the hardcopy terminal as a line printer.

Under many circumstances, it may be desirable to have the hardcopy terminal become the console terminal. Use the procedure described in Section 4.15.2 to do this.

4.15.1.2 The Hardcopy Terminal Is the Boot-Time Console

You can use any of the following procedures to make the hardcopy terminal the line printer when the hardcopy terminal is also the boot-time console. Both the hardcopy terminal and the video terminal must be on a DL interface.

In *Procedure 1* you can perform a system generation (without including the multiterminal feature) to make the video terminal appear to be the boot-time console. Note that the hardcopy terminal remains the hardware console interface. That is, you must still type the name of the system device on the hardcopy terminal in response to the boot prompt. However, RT–11 does print its boot message on the video terminal. Once the system is bootstrapped, you can use the LS handler to access the hardcopy terminal as a line printer.

In *Procedure 2* you can change your system configuration so that the video terminal is the boot-time console, and the hardcopy terminal is on a local DL interface. Then, you can use the procedure outlined in Section 4.15.1.1.

In *Procedure 3* you can use a special program to switch the background console to the video terminal. Except that the default boot-time console defaults to the hardcopy terminal after each reboot, this is similar to procedure 1, above. You can use the LS handler to access the hardcopy terminal as a line printer. Section 4.15.2 shows the program you run to use Procedure 3.

Procedure 4 is similar to Procedure 3, except that you alter the monitor image on a mass storage device instead of in memory. This procedure is useful only in systems without the multiterminal feature and only for another DL-like interface. Figure 4–5 shows the patch for Procedure 4. You must supply the correct value for the vector, CSR, protection offset, and protection code (see Section 2.6.1.2) for your application.

Figure 4–5: Patch for Procedure 4

```
! Permanent modification of monitor using CSR and Vector addresses
! CSR = 175620-175626 / Vec = 310-316
.R SIPP [RET]
*monitr.SYS [RET]                          ! monitr represents the file name
Base?     ;S [RET]                         ! of the monitor file you are
Search for?     60 [RET]                   ! changing
Start?          5100 [RET]
End?            5200 [RET]
Found at nnnnnn
Base?     nnnnnn [RET]
Offset?      [RET]
  Base       Offset      Old  New?
nnnnnn       000000   000060  310 [RET]  ! New vector
nnnnnn       000002   xxxxxx  [CTRL/Z]  [RET]
Offset?   6 [RET]
  Base       Offset      Old  New?
nnnnnn       000006   000064  314 [RET]  ! New vector plus 4
nnnnnn       000010   xxxxxx  [CTRL/Z]  [RET]
```

Figure 4–5 (continued on next page)

Figure 4–5 (Cont.): Patch for Procedure 4

```
Offset?    CTRL/Z  RET
Base?      $RMON RET              ! Find the value of $RMON on your
Offset?    304  RET               ! link map.
  Base       Offset        Old   New?
$RMON         000304     177560   175620  RET        ! New CSR
$RMON         000306     177562   175622  RET        ! New CSR
$RMON         000310     177564   175624  RET        ! New CSR
$RMON         000312     177566   175626  RET        ! New CSR
$RMON         000314     177777   CTRL/Z  RET
Offset?    342  RET                        ! Offest for protection byte
  Base       Offset        Old   New?
$RMON         000342     000000   17  RET   ! Enable protection
$RMON         000344     000000   CTRL/Y
* CTRL/C
.
```

4.15.2 Switching the Console Terminal

RT–11 distributes a program called CONSOL that you can use to switch the console terminal to another terminal in a system without the multiterminal special feature. Edit the source file to supply values for the CSR and vector for the new console; use the symbols CSRAD and VEC. To switch the console back and forth between two terminals, maintain two copies of the program, one for each terminal. The terminal interfaces must be DL11s; the program will not work on DH11 or DZ11 interfaces.

To switch the console terminal with DH11 or DZ11 interfaces, you must perform a system generation for multiterminal support and request handler hooks support for LS. See Section 4.4.2 for information.

26omment>(edited 19-sep-91)

Chapter 5
Interrupt Service Routines

This chapter describes the ways a program (as opposed to a device handler) can transfer data between memory and a peripheral device. First it covers noninterrupt programmed I/O; next it introduces the concept of using interrupts to handle device I/O by comparing the advantages and disadvantages of in-line interrupt service routines and device handlers. After these general points have been discussed, the chapter continues with a description of the structure of an interrupt service routine, and shows in detail how to organize and write one. A skeleton example of a foreground program that contains an interrupt service routine ends this discussion of applications. The discussion is followed by a final section dealing with the considerations involved in using interrupt service routines in a single-mapped, extended memory environment.

5.1 Noninterrupt Programmed I/O

One way to move data between memory and a peripheral device is to use noninterrupt programmed I/O. According to this method, your program operates with the device interrupts disabled and uses flags to coordinate the data transfer. Your program checks the ready bit in the status register for a particular device, moves the data when appropriate, and then either waits in a tight loop for another ready signal or does other processing and polls the device occasionally. Programmed I/O is device-specific and does not make use of operating system features designed for I/O processes. In addition, it ties up system resources until the I/O transfer is complete.

However, programmed I/O is sometimes the best method to use. For example, the Resident Monitor uses programmed I/O to print its *?MON–F–System halt* error message. It first performs a RESET to stop all active I/O. Then it waits in a tight loop for the console terminal to print the error message, one character at a time. Clearly in such a situation, where the monitor itself may be corrupted, no other job or data transfer could be running, and the console terminal is the only desirable output device. Also, the monitor .PRINT routine may have been corrupted and should not be used. Given these requirements, programmed I/O is the best method to use for printing this error message.

In an application program, noninterrupt programmed I/O can provide the quickest response to an external event with interrupts disabled.

The following lines of code from RMON demonstrate noninterrupt programmed I/O:

```
;
; Note that R1 points to the message text, terminated by a NUL
; TTPS is a word in memory containing the address of
; the terminal printer status register;
; its ready flag is the high-order bit of the low byte.
; TTPB is a word in memory containing the address of
; the terminal printer buffer.
; Moving a character to the printer buffer resets
; the busy flag in the status register.
;
5$:     TSTB    @TTPS       ;Test for tt busy
        BPL     5$          ;If yes, test again
        MOVB    (R1)+,@TTPB ;If no, print a character
        BNE     5$          ;Branch back if more to print
```

The device handler for the single-density diskette, DX, provides another example of programmed I/O. Reading data from the diskette one sector at a time, the handler first requests a read of one sector. The diskette completes the read operation, places the data in an internal silo, and issues an interrupt. The handler then disables diskette interrupts and uses programmed I/O to move data from the silo into memory. When it is ready to read another sector, the handler enables interrupts again.

The following lines of code are from a DX handler:

```
;
; Note that R4 points to the diskette status register;
; R5 points to the silo;
; R2 points to the data buffer in memory.
;
TRBYT:  TSTB    @R4     ;Wait for transfer ready
        BPL     TRBYT   ;Branch if tr not up
EFBUF:  MOVB    @R5,(R2)+ ;Transfer a character
        DEC     @SP     ;Check for count done
        BGT     TRBYT   ;Transfer more
```

Refer to the *PDP–11 Processor Handbook* for your computer for more information on noninterrupt programmed I/O.

5.2 Interrupt-Driven I/O

Although programmed I/O is useful in a few situations, generally the best way to handle device I/O is through interrupt processing. According to this method, a program starts an I/O transfer but continues processing. When the transfer completes, the device issues an interrupt. An interrupt service routine then determines whether the transfer is incomplete, complete, or has encountered an error. It takes the appropriate action (restarting the transfer, returning to the program, or possibly retrying the transfer in case of error). The advantages of using interrupt-driven I/O are that it enables two or more processes to run concurrently and it does not monopolize system resources.

5.2.1 How an Interrupt Works

An interrupt is a forced transfer of program execution that occurs because of some external event, such as the completion of an I/O transfer. The state of the processor prior to the interrupt is saved on the stack so that processing can continue smoothly after the return from the interrupt. The processor saves the Processor Status word, or PS, which reflects the current machine state, and the Program Counter, or PC, which indicates the return address.

Next, the processor loads new contents for the PC and PS from two preassigned locations in low memory, called an *interrupt vector*. These words contain the address of the interrupt service routine and the new PS, which indicates the new processor priority. When the interrupt service routine completes, it executes an RTI instruction, which restores the old PS and PC from the stack, and execution resumes at the interrupted point in the original program.

5.2.2 Device and Processor Priorities

Interrupt processing is closely related to device and processor priorities. Figure 5–1 illustrates the RT–11 priority structure. Each device on the system has a priority assigned to it and devices that must be serviced as soon as possible after they interrupt have the highest priority. Disks typically have priority 5; terminals and other character-oriented devices usually have priority 4. You can control the ordering of devices with the same priority. For these devices, the one closest to the CPU on the bus is serviced before other devices when interrupts occur simultaneously.

Figure 5–1: RT–11 Priority Structure

The central processor operates at any one of eight levels of priority, from 0 to 7. (Some older processors are an exception; they operate at either 0 or 7.) When the CPU is operating at priority 7, no device can interrupt it with a request for service. When the CPU is operating at a lower priority, only a device with a higher priority can cause an interrupt. You can adjust the processor's priority from within an interrupt service routine by modifying the Processor Status word. In an RT–11 system, software tools are provided to do this for you, so you never directly modify the PS yourself. The tools include the .MTPS and .MFPS programmed requests, and the .INTEN and .FORK macros.

The interrupt system allows the processor to continually compare its own priority with that of any interrupting devices and to acknowledge the device with the highest level above the processor's. This system can be nested—that is, the servicing of one interrupt can be left in order to service an interrupt with a higher priority. Service continues for the lower priority device when the higher priority device is finished.

See the *PDP–11 Processor Handbook* for your computer for more information on priorities and interrupts.

5.2.3 Processor Status (PS) Word

The Processor Status (PS) word occupies the highest address on the I/O page. (Again, some older processors are an exception; their PS is not addressable on the I/O page. The monitor refers to the PS by using the MTPS and MFPS instructions.) It contains information on the current status of the machine. This information includes the current processor priority, current and previous operational modes, the condition codes describing the results of the last instruction, and an indicator to cause the execution of an instruction to be trapped (used for program debugging).

Figure 5–2 illustrates the bits in the PS. Bits 5 through 7 determine the current processor priority. (In some older systems, only bit 7 determines the priority; priority is either 0 or 7.) By changing bits, you alter the CPU's priority. You can change the priority to 7, for example, to prevent any more interrupts from occurring. When you are servicing a particular interrupt, you can change the processor priority to the priority of that device so that only devices with a higher priority will interrupt that service routine. (Specifically, the device you are servicing cannot interrupt.) In general, you need not access the PS yourself; use the macros provided in RT–11, such as .INTEN and .FORK, to change the processor priority.

5.3 In-line Interrupt Service Routines Versus Device Handlers

Because both noninterrupt programmed I/O and interrupt-driven I/O are valid processes in an RT–11 system, when you need to interface a new device to your system—one that is not already supported by RT–11 – your first decision must be whether to use in-line interrupt service or to write a device handler for it. Whatever your decision, both interrupt service routines and device handlers can include noninterrupt programmed I/O sections as well as interrupt-driven code. The normal RT–11 interface between the monitor and a peripheral device is a device handler, which exists as a memory image file on a mass storage device, and resides in memory when it is needed to perform device I/O (see Chapter 1). A device handler usually includes an interrupt service routine within it.

If you choose to use an interrupt service routine, you must place the routine within your program so that your program directly changes the status and buffer registers for a specific device, and it can service the interrupts within its own code. This means, of course, that the interrupt service code must always be resident in memory.

Figure 5–2: Processor Status (PS) Word

On the other hand, if you choose to use a device handler, the interrupt service code is contained within the handler, not in your program. You issue .READ and .WRITE programmed requests from your main program, and the monitor and the handler together initiate the data transfer, service the interrupts, and notify your program when the transaction is done. In a single-job system, or for a background job in a multi-job system, the handler must be resident only when your program actually needs it to perform I/O. (That is, the handler must be resident whenever a file or channel is open.) For foreground jobs and system jobs in a multi-job system, you must load the handler (by using the monitor LOAD command) before you execute your program, so that the handler is always resident.

How you decide which method is more suitable for your new device depends largely on how you want the device to appear to system and application programs. In general, you might use in-line interrupt service for sensor or control devices, such as analog-to-digital converters. You should service devices that appear to be block-replaceable, file-structured mass storage devices, such as disks and diskettes, through device handlers. You can service most communications hardware by either method; the decision rests on other criteria.

The two major advantages of in-line interrupt service routines are their speed and the amount of control information they provide. Because there is no monitor overhead involved in a data transfer, an in-line routine can often handle interrupts faster than a device handler can. If the speed of servicing interrupts is crucial to

your application, you may choose to write an in-line interrupt service routine even if the device is a disk.

An in-line routine has access to all the device control and status registers for a device, as well as its data buffer registers. (Of course, a device handler has access to all the same registers, but the program using the handler does not.) It can pass a lot of information to the program. This provides a great deal of flexibility in the way the program calls the interrupt service routine, and in the amount of information the routine returns to it.

The three major advantages of using device handlers are that they provide device independence for your programs, they can share processor time with other processes, and they are simple to use. Device handlers have a standard protocol for interfacing to the RT–11 monitor. There is also a standard protocol for the interface between the monitor and a program, so that any program that conforms to the monitor standards can use the handler. This includes application programs, system utility programs, and RT–11 language processors such as MACRO–11, FORTRAN, BASIC–PLUS, and PDP–11 C. Thus, the device handler makes a new device available to a large number of programs without any special modification. (In addition, a device handler for a random-access device makes the RT–11 file system available on the device at no extra cost.) In contrast, an in-line interrupt service routine makes the new device available to just one application program.

Device handlers are easy to use. Because they are the standard RT–11 means of handling device I/O, the procedure for writing them and using them is clear and straightforward. This procedure is simplified further by the fact that RT–11 provides macros to write a handler; there are also keyboard monitor commands that install handlers into the monitor device tables and load them into memory. In addition, a device handler permits you to take advantage of the monitor programmed requests for performing data transfers. Finally, a device handler is the only way you can interface a device to a virtual job in a mapped system.

Figure 5–3 highlights some differences between in-line interrupt service routines and device handlers.

If you decide that your new device requires an in-line interrupt service routine, read the rest of this chapter to learn how to plan and write one. If you decide that a device handler is more suitable, read the rest of this chapter and then go on to *RT–11 Device Handlers Manual* to learn how to plan, write, and debug a handler.

5.4 How to Plan an Interrupt Service Routine

The most important part of writing an in-line interrupt service routine is taking the time to plan carefully. Follow these guidelines:

- Get to know your device
- Study the structure of an interrupt service routine
- Study the skeleton interrupt service routine

Figure 5–3: In-line Interrupt Service Routines and Device Handlers

- Think about the requirements of your program
- Prepare a flowchart of your program

- Write the code
- Test and debug the program

5.4.1 Get to Know Your Device

Getting to know your new device is crucial to writing an interrupt service routine that works correctly. If your device is a Digital peripheral, consult the hardware reference manual for that device. If your device is not from Digital, study the documentation for it carefully. Regardless of the format of the documentation (whether it is a manual, a brochure, or a set of engineering prints), it should contain the vital information you need to support it on a PDP–11 system. Be sure you obtain this information.

In any case, you must understand how the device operates: what it needs from you, and how it handles data transfers. Use the following checklist to make sure you have enough device-specific information to write the service routine. Do not attempt to write any code until you have considered each question.

Some of the following questions do not apply to all types of devices. Some are for mass storage devices, some are more appropriate for sensor devices or communications devices. Consider each question carefully, though, to see if it applies to your device.

- What is the interrupt vector (or vectors) for the device?

  Decide what the interrupt vector should be. Consider both conflicts with existing RT–11-supported devices and also conflicts with devices supported by other PDP–11 operating systems, if you use those systems. Once you decide on the vector, make sure the device is installed properly and that the hardware is jumpered for that address. RT–11 requires all vectors to be below location 500 and some low-memory locations are not available for use as vectors. Chapter 1 lists the current PDP–11 vector assignments.

- What are the control and status registers?

  Learn where these registers are located and what the bits in each mean.

- What is the priority for the device?

- Is the device DMA (Direct Memory Access) or programmed transfer (word- or character-oriented)?

- What are the data buffer registers?

  Learn where these registers are located and what the bits in each mean.

- What are the op codes for typical operations?

  Learn how to initiate the various operations by manipulating the bits in the device registers.

- When does the device interrupt?

  Some devices interrupt for each character; others are word-oriented, block-oriented, or packet-oriented. Some devices interrupt twice for certain operations,

such as seek or drive reset. Find out if your device does this, and plan now to take this information into account later.

- What is the basic unit for data transfers?

  This relates to the previous question, of course, but you must determine whether to send I/O requests to the device as byte, word, or block counts. If, for example, your program deals in terms of words and the device is character-oriented, you may have to convert the word count to a byte count in the service routine.

- Does the device want a positive or negative byte count?

  Some devices require a negative byte or word count. If your device is one of those, you may need to negate the count in the service routine.

- What is the device structure, or geometry?

  If the device is a disk, find out how the cylinders, tracks, and sectors are structured. Determine their size. Find out if the device requires interleaving, and, if so, learn how to optimize for speed. (*Interleaving* describes the process for writing data to a spinning device that requires program intervention between sectors. The disk is constantly moving; data is written into one sector, the program intervenes as the adjacent sector spins past, then more data is written into the next available sector.)

- What is the buffering arrangement?

  Some devices transfer data to your program one character at a time. Others buffer data internally in a silo, or send it in packets. Decide how to buffer the data in your program. Make sure the buffer space you allocate is large enough.

- How do you calculate the address of the data on the device?

  This relates to the device's structure. Study the device now and determine how to find the data you want on it. Note that RT–11 block numbers must be converted to device-specific addresses. Note also that some processors have no multiply or divide instructions.

- What ''housekeeping'' operations does the device require?

  Some devices require a drive reset before a retry. Others require that the device be selected or that a disk pack be acknowledged before you can perform any operations on it. You might have to do a drive reset after a seek incomplete or a drive error, for example.

- How will you handle errors and exception conditions?

  First you must decide which errors are hard and will abort the transfer, and which errors are soft and will retry the transfer. Some typical soft errors include checksum errors, data late errors, and timing errors. Decide how many times you will retry the transfer for soft errors, and how you will handle a hard error condition.

- What are the abort considerations?

Consider whether the device is relatively fast or slow. Keep in mind that you do not want to issue a controller reset if only one unit of a two-unit controller is affected by a program's abort, because this can interfere with the operation of the second unit. Similar considerations may apply to dual-ported devices.

5.4.2 Study the Structure of an Interrupt Service Routine

Section 5.5 describes the structure of an interrupt service routine. Read this section carefully.

5.4.3 Study the Skeleton Interrupt Service Routine

Section 5.6 contains a skeleton outline of a foreground job with an in-line interrupt service routine. Study this outline to be sure you understand the flow of execution.

5.4.4 Think About the Requirements of Your Program

Remember that the interrupt service routine is part of your program and decide where to place it in the program. Review the material in Chapter 1 on swapping the USR. If you plan to execute your program in a mapped environment, read Section 5.7 for mapping considerations.

5.4.5 Prepare a Flowchart of Your Program

Many experienced programmers prepare flowcharts after all their programs are written, or they omit them entirely. However, flowcharting a system with the complexities of interrupt service can help you find loose ends and point out errors in your logic. Flowcharts are not much help, unfortunately, in pointing out potential race conditions. (A race condition is a situation in which two or more processes attempt to modify the same data structure at the same time; as a result, the data structure is corrupted and the integrity of the processes is compromised. It may be caused by a device interrupting while its interrupt service routine is running, due to improper processor priority.) When you design your program, examine every step carefully; keep in mind what would happen if an interrupt occurred at each instruction. This kind of planning can help you avoid race conditions later.

Spend enough time to design a clean and straightforward way of handling error conditions; if your program can handle error conditions well, you will probably find that the rest of your program design works well too.

5.4.6 Write the Code

If you have followed the recommended steps so far, writing the code for the interrupt service routine itself should be relatively simple. You can borrow as much code as possible from other interrupt service routines you have studied. Start with a general outline, then add details to reflect the specifics of your particular device. When you are satisfied with the code, have checked it thoroughly for logic errors, and it assembles properly, you are ready to test and debug it.

5.4.7 Test and Debug the Program

The only way to test a program with in-line interrupt service is to try executing it. If the program is operating correctly, it should be able to read or write data accurately, should not lose any data, and should handle error conditions properly. Try executing the program in a test situation with data you have prepared. If you find errors, use the DBG–11 symbolic debugger to step through your program. If for some reason you cannot use DBG–11, link the program with ODT (not VDT) and try running it step by step. Make coding corrections, reassemble the program, and retry it as necessary.

5.5 Structure of an Interrupt Service Routine

The following sections outline the general structure of an in-line interrupt service routine. Read them carefully and determine which items apply to your own situation.

5.5.1 Protecting Vectors: .PROTECT

In systems where more than one job can be running, you should use the .PROTECT programmed request to protect an interrupt vector before you move a value to it. This process makes sure that the vector does not already belong to the monitor or to another job. It gives ownership of the vector to your job, and protects it from interference from another job by setting bits in the monitor bitmap. (Chapter 2 describes the low-memory bitmap in detail.) Your job should abort immediately if the .PROTECT request fails; your job must not access a vector that is already in use. See Sections 5.5.2 and 5.6 for examples of how to use .PROTECT.

See the *RT–11 System Macro Library Manual* for the format of the .PROTECT programmed request.

Even though the .PROTECT request has no meaning in a single-job system, it is advisable to use it in your program. The request takes no action, returning immediately to your program, yet is useful later if your program needs to run in a multi-job environment.

5.5.2 Setting up the Interrupt Vector

Your program must take care of moving the address of your interrupt service routine to the first word of the interrupt vector. RT–11 requires all interrupts to raise the processor priority to 7, so your program must fill in the second word of the interrupt vector with 7 as the new priority. The following lines of code show a typical way for a program to set up the two-word interrupt vector. Note that a program should not set up a vector until the vector is protected. For this example, assume the device name is XX, and the interrupt vector is at 220 and 222.

```
        .MCALL  .DEVICE .GVAL   .INTEN  .PROTECT
        .MCALL  .RSUM   .SPND   .SYNCH

        .LIBRARY "SRC:SYSTEM.MLB"

        .MCALL  .FIXDF
        .FIXDF

; ** MAIN PROGRAM **
```

```
        xxVEC  = vvv            ; The device vector
        PR7    = 340            ; Priority 7
        DEVPRI = 5              ; Device priority = 5 (0-7, not 000-340)
        xxCSR  = nnnnnn         ; The device control register
        IENABL = 100            ; Interrupt enable bit

START:  .PROTECT #AREA,#xxVEC   ; Protect the vector
        BCS     ERROR           ; Handle .PROTECT error
        .GVAL   #AREA,#$JOBNU   ; Get our job number in R0
        BCS     ERROR           ; Handle .GVAL error
        MOV     R0,SYNBLK+2     ; Store our job number in synch block
        MOV     #ISREP,@#xxVEC  ; Set up first word of vector
        MOV     #PR7,@#xxVEC+2  ; Set up second word of vector
        .DEVICE #AREA,#DEVLST   ; Disable device on exit or abort

; Insert lines of code here to initialize input buffers in the
; service routine and to initialize other pointers and flags

SPND:   BIS     #IENABL,@#xxCSR ; Enable interrupts
        .SPND                   ; Wait until there is some data

; Insert lines of code here to store the data and reset some flags

        BR      SPND            ; Wait for more data

DEVLST: .WORD   xxCSR           ; List for .DEVICE
        .WORD   0               ; Clear xxCSR on .EXIT or abort
        .WORD   0               ; Terminate .DEVICE list
AREA:   .BLKW   3               ; Programmed request argument block

ERROR:      .                   ; Routines to handle errors
            .
            .

; ** INTERRUPT SERVICE ROUTINE **

ISREP:      .                   ; The interrupt entry point
            .                   ;  (priority is 7)
            .
        .INTEN  DEVPRI          ; NOTE:  not ".INTEN #DEVPRI"
                                ;  Lower to device priority
                                ;   and enter system state
                                ;   with R4 and R5 available
;
; If there is more data to collect:
;
        BR      RET
;
; If there is no more data to collect:
;
        .SYNCH  #SYNBLK         ; Go back to main program to process data
         BR     SYNERR          ; .SYNCH returns here on error
        .RSUM                   ; Wake up main program
RET:    RETURN                  ; Wait for another interrupt

SYNBLK: .WORD   0,0,0,0,0,-1,0  ; Job number is filled in by code at START

SYNERR:                         ; Process .SYNCH error
```

5.5.3 Stopping Cleanly: .DEVICE

The .DEVICE programmed request turns off a device (by clearing its interrupt enable bit) if its associated program is aborted or when the program exits. (See the *RT–11 System Macro Library Manual* for the format of the .DEVICE programmed request. See Section 5.6 of this manual for an example using .DEVICE.)

This request is not required in a single-job environment. However, even though the request has no meaning in a single-job system, it is advisable to use it in your program. The request takes no action, returning immediately to your program, yet is useful later if your program needs to run in an multi-job environment.

5.5.4 Lowering Processor Priority: .INTEN

When an interrupt occurs, control passes to your interrupt service routine entry point the address you supplied as the first word of the interrupt vector. At this point, the processor priority is 7, and all other interrupts are prohibited. If you need to do anything with all interrupts disabled, this is where the code belongs. It should be as short and efficient as possible and should not destroy the contents of any registers. If this code needs to use registers, it must save them and restore them before issuing the .INTEN call. If the code executed at priority 7 is too long, system interrupt latency (a measure of how quickly the system can respond to an interrupt) will suffer. A good guideline is to spend no more than 50 microseconds at priority 7.

You should lower the processor priority to that of the device as soon as possible. This means that only devices with a higher priority than this one will be able to interrupt its service routine. To lower the priority, use the .INTEN programmed request. The stack pointer and general registers R0 through R5 must contain the same values when your interrupt service routine issues the .INTEN request as they did at the interrupt entry point. If your interrupt service routine is not written in Position-Independent Code (PIC), use the following format:

```
.INTEN prio
```

The .INTEN call generates the following code:

```
JSR     R5,@54
.WORD   ^C<PRIO*40>&340
```

If your interrupt service routine is written in PIC, use the .INTEN call with a second argument, *PIC*.

```
.INTEN  prio,PIC
```

The second format generates Position-Independent Code:

```
MOV     @#54,-(SP)
JSR     R5,@(SP)+
.WORD   ^C<PRIO*40>&340
```

Both formats cause a JSR to the monitor's INTEN routine, which lowers the processor priority and switches to system state. The monitor then calls the interrupt service routine back as a co-routine. R4 and R5 are available for use on return from the call. You must not destroy the contents of any other registers. If you need R0 through R3, save them on the stack or in memory and restore them before you exit.

If you need to preserve values across the .INTEN request, you must save them in memory before the call and restore them after it. Likewise, if the contents of the PS are important, such as the values of the condition bits, you should save them before issuing the .INTEN call.

In unmapped systems, a .INTEN causes a switch to the system stack, so you should avoid using the stack excessively once you are in your interrupt service routine. In mapped systems, the stack is switched to the kernel stack upon entering the ISR.

Save and restore registers and the PS, as necessary, by using memory locations instead of the stack.

NOTE

Saving values in memory locations may prevent your interrupt routine from being reentrant. If you intend to use the routine for multiple devices, be careful about reentrancy when you design it.

(See the *RT–11 System Macro Library Manual* for more information on .INTEN. See Section 5.6 of this chapter for an example using .INTEN. See Section 5.5.7 for a summary of the interrupt service routine macro calls.)

5.5.5 Issuing Programmed Requests: .SYNCH

The .SYNCH call is useful mainly in the multi-job environments. Its purpose is to make sure that the correct job is running when an interrupt service routine executes a programmed request. Even though the .SYNCH call has no meaning in a single-job system, it is advisable to use it in your program. The request takes no action, returning immediately to your program, yet is useful later if your program needs to run in a multi-job environment.

For the format and a complete expansion of this macro, see the listing of the system macro library in the *RT–11 System Macro Library Manual*.

If you need to issue one or more RT–11 programmed requests from the interrupt service routine, you must first issue the .SYNCH call. Remember that the .INTEN call switched execution to system state, and programmed requests can only be made in user state. The .SYNCH call itself handles the switch back to user state. Note that you should never issue programmed requests requiring the USR from within an interrupt service routine, even after using .SYNCH. You can also issue .SYNCH after .FORK, which is covered in Section 5.5.6. When you issue the .SYNCH call, R0 through R3 and the stack pointer must contain the same values as they did when the .INTEN request returned to you.

Table 5–1 illustrates the format of the synch block, which acts like a completion queue element. The information in the seven-word synch block is placed at the head of the appropriate job's completion queue. Therefore, the code following the .SYNCH request executes as a completion routine, in user state, at priority 0. Because of this, your program must either disable interrupts before the .SYNCH call, or it must be prepared for the device to interrupt again before the .SYNCH code executes. The synch block is available for reuse when QS.CMP (offset 14_8) is 0. You can test the

synch block easily by issuing another .SYNCH. If control passes to the error return (the word following the .SYNCH call), the block is still in use.

Table 5–1: Synch Block (.QSYDF)

Offset	Name	Agent	Contents
0	QS.LNK	—	Pointer to next queue element or 0 if none.
2	QS.JOB	User	Job number.
4	—-	——	Reserved.
6	—-	——	Reserved.
10	QS.ID	User	Argument to pass in R0.
12	QS.SYN	Monitor	–1.
14	QS.CMP	User	Initialize as 0; after you issue a .SYNCH, the monitor maintains the contents of this word.

In general, a long time can elapse between the .SYNCH call and the return. First, the monitor switches to user state, and a scheduling pass is required to determine whether or not a context switch is also necessary. Then a background completion routine may have to wait for a compute-bound foreground job to become blocked. So, it may take a considerable amount of time before the code following the .SYNCH actually executes.

In the code following the .SYNCH call, R0 and R1 are free for use, as they are in any completion routine. However, you must preserve R2 through R5 if your .SYNCH routine uses them. This poses a problem for R4 and R5, which are not preserved across the call. If their contents are important, save them in memory before the .SYNCH call. You can use QS.ID in the synch block to pass a value into R0 for the synch routine.

The .SYNCH call has an unusual error return. The first word after .SYNCH is the return address on error; the second word after .SYNCH is the return on success. Routines following .SYNCH calls (and, in fact, completion routines in general) are serialized.

See Section 5.6 for an example using .SYNCH. See Section 5.5.7 for a summary of the interrupt service routine macro calls.

5.5.6 Running at Fork Level: .FORK

The .FORK programmed request gives you another way to lower the processor priority. (See the *RT–11 System Macro Library Manual* for the format of the .FORK programmed request. For the complete expansion of this macro, see the listing of the system macro library in that manual.)

When you issue a .FORK call, the fork block is added to a fork queue, which is a first-in, first-out list. Fork routines (all the code following a .FORK call) execute in system state at priority 0, after all interrupts have been serviced, but before the monitor switches to user state. Context switching is inhibited as well during the

time fork routines are executing. (See Figure 5–1 for a review of RT–11 priority levels.)

R4 and R5 are preserved across the .FORK call. In addition, R0 through R3 are free for use after the call. Like .SYNCH, the .FORK call assumes you have not changed R0 through R3 or the stack since the .INTEN call returned to you. See Section 5.5.7 for a summary of the interrupt service routine macro calls. Note that you cannot issue .FORK without a prior .INTEN call.

You must provide a four-word block of memory for the fork queue element, the last three words of which will contain the return PC, R5, and R4. The first word is a link word, which must be 0 when you issue the .FORK request. Because a .FORK routine should not be reentrant, make sure that the device cannot interrupt between the time you issue the .FORK call and the time the .FORK routine (the code following the call) begins to execute.

You cannot reuse a fork block until the fork routine has been entered. It is safe to assume that the fork block is free when the call that used it returns. See Table 5–2 for an illustration of the fork block.

Table 5–2: Fork Block

Offset	Name	Agent	Contents
0	F.BLNK	Monitor	Link word.
2	F.BADR	Monitor	Address of FORK routine.
4	F.BR5	Monitor	R5 save area.
6	F.BR4	Monitor	R4 save area.

Generally, .FORK is used in device handlers. To use it in an interrupt service routine, you must first set up a pointer called $FKPTR. The recommended way to do this in a main program is as follows:

```
        MOV     @#$SYPTR,R4
        ADD     $FORK(R4),R4
        MOV     R4,$FKPTR
        .
        .
        .
$FKPTR: .WORD   0
XXFBLK: .WORD   0,0,0,0
```

Then, in the interrupt service routine, you can use the normal form of the .FORK macro:

```
        .FORK   XXFBLK
```

The .FORK macro expands as follows:

```
        JSR     R5,@$FKPTR
        .WORD   XXFBLK-.
```

Note that in your interrupt service routine, no registers are free for use before the .INTEN call. After the .INTEN, you can safely use R4 and R5. See Section 5.5.7 for a summary of the interrupt service routine macro calls.

The .FORK request has several applications in a real-time environment because it permits lengthy but noncritical interrupt processing to be postponed until all other interrupts are dismissed. Further, the .FORK request (unlike .SYNCH) does not cause a context switch, which is a lengthy process.

The .FORK request returns at priority 0, but only when all other interrupts have been dismissed and before control is returned to the interrupted user program. (Note that you dismiss an interrupt when you leave interrupt level, by any one of several means.)

5.5.7 Summary of .INTEN, .FORK, and .SYNCH Action

Table 5–3 summarizes the effects of the .INTEN, .FORK, and .SYNCH macro calls. Figure 5–4 describes the status of the registers for each call.

Table 5–3: Summary of Interrupt Service Routine Macro Calls

Macro Call	New Priority	New Stack[1]	Registers Available to Use After Call	Your Data Preserved Across Call In:
.INTEN	Device's	System	R4, R5	None
.FORK	0	System	R0–R5	R4, R5
.SYNCH	0	User	R0, R1	R0

[1]In mapped systems, all are kernel stack; in unmapped systems, are as shown.

5.5.8 Exiting from Interrupt Service

The .INTEN request causes the monitor to call your interrupt service routine as a co-routine. At the end of your routine, when it is time to exit, use a RETURN instruction. This returns control to the monitor, which restores R4 and R5 and then executes an RTI instruction.

You also exit from .FORK and .SYNCH routines with a RETURN instruction. Be sure that the stack is the same as it was upon entry, and that any registers that must be preserved have their original contents.

5.6 Skeleton Outline of an Interrupt Service Routine

Figure 5–5 shows a foreground main program that contains an in-line interrupt service routine. The foreground program performs some initialization tasks and then suspends itself. When data is available from a peripheral device, the interrupt service routine collects it. When all the data is gathered, the interrupt service routine resumes the main program, which can then process the new information

Figure 5–4: Summary of Registers in Interrupt Service Routine Macro Calls

before suspending itself again. The main program's processing could involve some manipulation of the new data or it could be writing the data to a file shared by a background data analysis job.

For this example, *xx* represents the device name.

Figure 5–5: Skeleton Interrupt Service Routine

```
        .MCALL  .DEVICE .GVAL   .INTEN  .PROTECT
        .MCALL  .RSUM   .SPND   .SYNCH

        .LIBRARY "SRC:SYSTEM.MLB"

        .MCALL  .FIXDF
        .FIXDF

; ** MAIN PROGRAM **

        xxVEC  = vvv            ; The device vector
        PR7    = 340            ; Priority 7
        DEVPRI = 5              ; Device priority = 5 (0-7, not 000-340)
        xxCSR  = nnnnnn         ; The device control register
        IENABL = 100            ; Interrupt enable bit

START:  .PROTECT #AREA,#xxVEC   ; Protect the vector
        BCS     ERROR           ; Handle .PROTECT error
        .GVAL   #AREA,#$JOBNU   ; Get our job number in R0
        BCS     ERROR           ; Handle .GVAL error
        MOV     R0,SYNBLK+2     ; Store our job number in synch block
        MOV     #ISREP,@#xxVEC  ; Set up first word of vector
        MOV     #PR7,@#xxVEC+2  ; Set up second word of vector
        .DEVICE #AREA,#DEVLST   ; Disable device on exit or abort

; Insert lines of code here to initialize input buffers in the
; service routine and to initialize other pointers and flags

SPND:   BIS     #IENABL,@#xxCSR ; Enable interrupts
        .SPND                   ; Wait until there is some data

; Insert lines of code here to store the data and reset some flags

        BR      SPND            ; Wait for more data

DEVLST: .WORD   xxCSR           ; List for .DEVICE
        .WORD   0               ; Clear xxCSR on .EXIT or abort
        .WORD   0               ; Terminate .DEVICE list
AREA:   .BLKW   3               ; Programmed request argument block

ERROR:      .                   ; Routines to handle errors
            .
            .

; ** INTERRUPT SERVICE ROUTINE **

ISREP:      .                   ; The interrupt entry point
            .                   ;  (priority is 7)
            .
        .INTEN  DEVPRI          ; NOTE:  not ".INTEN #DEVPRI"
                                ;  Lower to device priority
                                ;   and enter system state
                                ;   with R4 and R5 available
;
; If there is more data to collect:
;
        BR      RET
;
```

Figure 5–5 (continued on next page)

Figure 5–5 (Cont.): Skeleton Interrupt Service Routine

```
; If there is no more data to collect:
;
        .SYNCH  #SYNBLK         ; Go back to main program to process data
         BR     SYNERR          ; .SYNCH returns here on error
        .RSUM                   ; Wake up main program
RET:    RETURN                  ; Wait for another interrupt

SYNBLK: .WORD   0,0,0,0,0,-1,0  ; Job number is filled in by code at START

SYNERR:                         ; Process .SYNCH error
```

5.7 Interrupt Service Routines in Mapped Systems

If you are not planning to execute your program in a mapped environment, you need not read this section.

Of the two kinds of jobs in a mapped environment, virtual jobs and privileged jobs, virtual jobs cannot contain in-line interrupt service routines (see Chapter 3). Virtual jobs cannot directly access the real vectors but instead use virtual vectors.

If a job containing an in-line interrupt service routine must run in a mapped environment, it must run as a privileged job. Privileged mapping makes the low 28K words of memory and the I/O page available to the program and permits the program to map portions of the user virtual address space into extended physical memory if the program requires it.

In order to understand the restrictions that mapping imposes on interrupt service routines, you must understand that when an interrupt occurs in a mapped system, its service routine executes with kernel, not user, mapping. This means that whether or not the program has mapped some of its virtual address space into extended memory, the interrupt service routine executes with the default kernel mapping to the low 28K words of memory plus the I/O page. It makes sense, therefore, that the first restriction demands that the mapping for your interrupt service routine plus any data it uses must be identical to kernel mapping at any time that an interrupt could occur.

Figure 5–6 shows the default kernel mapping scheme, which provides access to the low 28K words of memory plus the I/O page. This is also the mapping scheme for a privileged job when it first begins execution. And, this is the mapping scheme that takes effect whenever an interrupt is serviced. (The shaded areas in the figure represent memory that the user job cannot access.) In Figure 5–6, the interrupt vector at 200 and 202 contains the entry point, called ISREP:, of the interrupt service routine, and the value 340, which represents the new PS. When an interrupt occurs, the system uses kernel mapping to locate the interrupt service routine. In this example, it should start at address 120000. Since privileged mapping and kernel mapping are identical in this diagram, the interrupt service routine is located in physical memory exactly where the kernel mapping points, so it can execute correctly.

Figure 5–6: Kernel and Privileged Mapping

Figure 5–7 shows a privileged job that changes the user virtual address mapping. (The shaded areas in the figure represent memory that the user job cannot access.) You can see from the example that the interrupt service routine cannot execute correctly when an interrupt occurs because the interrupt service routine is not located in physical memory where it should be. The memory area pointed to by the kernel mapping contains random data or instructions.

The second restriction for interrupt service routines relates to the way the monitor uses Page Address Register (PAR) 1 with kernel mapping. PAR1 controls the mapping for virtual addresses 20000 through 37776. When XM is first bootstrapped with kernel mapping, the virtual addresses map directly to the same physical addresses. However, the monitor itself uses PAR1 to map to EMT area blocks and to user data buffers. So, whenever the system is running, the kernel virtual addresses in the PAR1 range can be mapped just about anywhere in physical memory and you have no way of controlling it. You must be sure that your interrupt service routine and any data it needs are not located in the virtual address range mapped by PAR1. Figure 5–8 illustrates this restriction. Valid locations for interrupt service routines,

Figure 5–7: Interrupt Service Routine Mapping Error

assuming that privileged mapping is identical to kernel mapping at the time of the interrupt, are marked on the diagram as ‘‘OK’’.

If your interrupt service routine needs a window into memory, it can borrow PAR1 the same way the monitor does. It must save the contents, set the value it needs, and restore the original contents before exiting. It can do this at .INTEN or fork level, but not at synch level.

NOTE

If your system uses the MQ handler to communicate among system jobs and you have defined the conditional assembly parameter MQH$P2=1 during system generation, all the restrictions for PAR1 also apply to PAR2 – the range of addresses from 40000 through 57777.

One final piece of information is important if you use .SYNCH in your interrupt service routine. The lines of code following .SYNCH execute almost like a completion

Figure 5–8: PAR1 Restriction for Interrupt Service Routines

routine. Completion routines in mapped environments execute with the user stack and with user mapping. But, since the code following .SYNCH is still part of an interrupt service routine, it executes in user context, but with *kernel* mapping. So, the code following a .SYNCH call in mapped environments must observe the same restriction as the main body of the service routine: its mapping must be identical to kernel mapping at any time that an interrupt could occur, or any time the completion routine could be executing. Of course, it must observe the PAR1 and PAR2 restrictions as well.

Appendix A
The System Definition Library (SYSTEM.MLB)

A.1 Introduction

The macro library file SYSTEM.MLB contains macros that define symbolic names, values, and offsets for the data structures of the RT–11 system. For example, the macro .FIXDF in SYSTEM.MLB defines symbolic names for all the monitor fixed offsets. Digital suggests that you use the values and symbolic names defined by SYSTEM.MLB whenever you refer to data structures in RT–11.

The macros in SYSTEM.MLB are of three general classes:

- ..%%%%
- .%%%DF
- Miscellaneous

The ..%%%% macros define symbolic offsets and values for RT–11's programmed requests. Examples of this form include ..GTIM, ..READ, and ..WRIT.

The .%%%DF macros define data areas in the monitor and data areas used or created by programmed requests. Examples of this form include .DATDF and .FIXDF. Some macros of the .%%%DF form existed in SYSMAC.SML prior to the creation of SYSTEM.MLB. These include .QELDF, .RDBDF, and .WDBDF. Those macros still reside in SYSMAC.SML, and a new one, .CMPDF, has been added.

The miscellaneous macros perform various functions that do not fall into the other categories.

You can get a macro expansion listing of any macro in SYSTEM.MLB if you want to see what symbols that macro defines. For example, to see the expansion of the ..WRIT macro, create a simple MACRO–11 program called, say, TEST.MAC:

```
.MCALL  ..WRIT
..WRIT LIST=YES
.END
```

Assemble TEST.MAC and include SYSTEM.MLB as a macro library on the command line. SYSTEM.MLB must reside on logical device SRC. Assign the volume on which it resides (*dd* to SRC.

```
.ASSIGN dd SRC RET
.MACRO TEST,SYSTEM.MLB/LIBRARY/LIST RET
```

If you receive an insufficient memory error message, use the V command:

```
.V MACRO RET
*TEST,TEST=TEST,SRC:SYSTEM.MLB/M RET
* CTRL/C
```

The results should look similar to the following:

```
.MAIN. MACRO V05.05  Saturday 01-Jun-91 05:25  Page 1

      1
      2
      3
      4                                         .mcall  ..write
      5
      6 000000                                  ..write list=yes
                000375                  ...WRI  =: ^o375
                000000                  A.CHAN  =: 0
                000001                  A.CODE  =: 1
                000011                  .WRIT   =: ^o11
                000002                  A.BLK   =: 2
                000004                  A.BUF   =: 4
                000006                  A.WCNT  =: 6
                000010                  A.CRTN  =: 10
                000012                  L.WRIT  =: 12
                000010                  A.TYPE  =: A.CRTN
                000001                  ..ISIO  =: 1
                000000                  ..WTIO  =: 0
                000203                  ..EMIM  =: ^o203
                000003                  ..EMIO  =: ^o003
                000377                  ..DRIO  =: ^o377
                000001                  ..ISIO  =: 1
                000000                  ..WTIO  =: 0
                000000                  ..USER  =: 0
                000004                  ..SUPY  =: ^o04
                000010                  ..CURR  =: ^o10
                000000                  ..DSPA  =: 0
                000020                  ..ISPA  =: ^o20
                000010                  A.TYPE  =: A.CRTN
                000012                  A.SRTN  =: ^o12
                000001                  ..CSUP  =: 1
                000014                  L.WRIU  =: ^o14
      7
      8         000001                          .end
                .
                .
                .
```

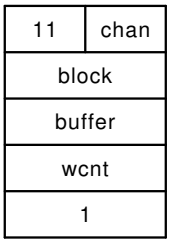
The following sections describe the macros in SYSTEM.MLB in more detail.

A.2 Macros That Define EMT Request Block Structures

Macros in SYSTEM.MLB with names of the form ..%%%% define the structure of EMT request blocks. For example, the ..WRIT macro defines offsets, values, and symbols for the request block used by the .WRITE, .WRITC, and .WRITW programmed requests.

For ordinary programming, you do not need to use these macros. Just use the programmed requests (such as .WRITE) as your program requires them. The programmed requests build their EMT request blocks before issuing the EMT instruction to call the monitor; you do not need to worry about the details. If you must optimize your program for speed or space, however, you may choose to build programmed request blocks yourself or to reuse an old request block after modifying one or two words in it. If that is the case, you can use the SYSTEM.MLB macros to help you. Let's continue with the example of .WRITE.

The .WRITE programmed request builds an EMT request block, pointed to by R0, that looks like this:



(Refer to the *RT–11 System Macro Library Manual* for more detail about .WRITE.) The ..WRIT macro defines symbolic values for these entries. The following table describes them in more detail.

Symbol	Value	Comments
...WRI	^o375	EMT code for .WRITE requests
A.CHAN	0	Offset for byte to hold channel number
A.CODE	1	Offset for byte to hold EMT subcode value
.WRIT	11_8	EMT subcode value for .WRITE request
A.BLK	2	Offset for word to hold block number
A.BUF	4	Offset for word to hold buffer address
A.WCNT	6	Offset for word to hold word count
A.CRTN	10	Offset for word to hold completion routine
L.WRIT	12	Length of EMT request block (bytes)
A.TYPE	A.CRTN	Offset for word identifying type of I/O
..ISIO	1	Value to put in A.TYPE for .WRITE request
..WTIO	0	Value to put in A.TYPE for .WRITC request
..EMIM	203_8	Mask for testing ..EMIO
..EMIO	003_8	Fixed bits for extra mapping

Symbol	Value	Comments
..DRIO	377_8	BUF normal mapping, SRTN maybe supy
..ISIO	1	Issue I/O
..WTIO	0	Wait on I/O
..USER	0	I/O to User mode
..SUPY	04_8	I/O to Supervisor mode (value)
..CURR	10_8	I/O to Current mode (value)
..DSPA	0	I/O to D space (bit)
..ISPA	20_8	I/O to I space (bit)
A.TYPE	A.CRTN	Completion I/O
A.SRTN	12_8	A.CRTN value if A.TYPE has 2 lowest bits set
..CSUP	1	Low bit A.SRTN completion to Supervisor mode
L.WRIU	14_8	Area length of A.TYPE has lowest bits set

Using these definitions, you can build your own EMT request block for doing a .WRITE request. Building a dynamic request block (using normal .WRITE macro) takes about twice the code and data space of one you statically build. Variables in lowercase are program-dependent values that you supply:

```
.LIBRARY "SRC:SYSTEM"

.MCALL  ..WRIT
        ..WRIT

        MOV     #pwrite,R0              ;point to filled in request block
        EMT     ...WRI                  ;and issue .WRITE EMT
        .
        .
        .
pwrite:                                 ;statically build request block
.=pwrite+A.CHAN
        .byte   chanl                   ;channel number
.=pwrite+A.CODE
        .byte   .WRIT                   ;EMT subcode
.=pwrite+A.BLK
        .WORD   blknum                  ;block number
.=pwrite+A.BUF
        .WORD   bufadr                  ;buffer address
.=pwrite+A.WCNT
        .WORD   wrdcnt                  ;word count
.=pwrite+A.TYPE
        .WORD   ..ISIO                  ;issue I/O type
```

A.3 Macros That Define Data Areas

SYSTEM.MLB defines many data area definition macros, such as .JSWDF, which assigns names to bits in the JSW. As with the ..%%%% macros, you can get a listing of any .%%%DF macro expansion by creating a simple MACRO–11 program calling the macro with LIST=YES. For .JSWDF, the program would look like this:

```
.MCALL  .JSWDF

.JSWDF  LIST=YES

.END
```

The following is the expansion of the .JSWDF macro:

```
.MAIN. MACRO V05.05  Saturday 01-Jun-91 06:02  Page 1

      1
      2                                          .MCALL  .JSWDF
      3 000000                                   .JSWDF  LIST=YES
                000004                 VRUNV$  =: ^o4
                000010                 GTLIN$  =: ^o10
                000020                 EDIT$   =: ^o20
                000040                 SPXIT$  =: ^o40
                000100                 TCBIT$  =: ^o100
                000200                 VBGEX$  =: ^o200
                000200                 HLTER$  =: VBGEX$
                000400                 CHAIN$  =: ^o400
                001000                 OVLY$   =: ^o1000
                002000                 VIRT$   =: ^o2000
                004000                 CHNIF$  =: ^o4000
                010000                 TTSPC$  =: ^o10000
                020000                 RSTRT$  =: ^o20000
                040000                 TTLC$   =: ^o40000
                100000                 USWAP$  =: ^o100000
      4         000001                          .END
                .
                .
                .
```

A.4 Miscellaneous Macros

SYSTEM.MLB contains some general utility macros that do not fall into the ..%%%% or .%%%DF categories. Unlike the ..%%%% and .%%%DF macros, these miscellaneous macros do not support the LIST=YES functionality to enable you to get a listing of the macro's results.

The following table lists the miscellaneous macros in SYSTEM.MLB.

Macro	Function
.LIST.	Listing control for .DS, .BS, .EQU, .LB

Macro	Function
.BSECT	Initialize definition of bits
.BS	Define 1 or more bits
.LB	Find low bit in bit pattern
.DSECT	Initialize definition of storage area
.DS	Define 1 or more storage locations
.EQU	Generic (=, =:, ==!, ==:) equate

A.5 Macros That Are Available in SYSTEM.MLB

Table A–1 lists all macros in SYSTEM.MLB, alphabetized by macro name.

Table A–1: SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
...CMA	.SPFUN	Support macro for areas for EMT375 requests
...CMT	.MTxxx	Support macro for areas for .MT* macros
..%%%%		Generic information about definition macros
..ABTI	.ABTIO	EMT request layout / values
..ASTX	.ASTX	EMT request layout / values
..CALL	.CALLK	EMT request layout / values
..CDFN	.CDFN	EMT request layout / values
..CHAI	.CHAIN	EMT request layout / values
..CHCO	.CHCOPY	EMT request layout / values
..CLOS	.CLOSE	EMT request layout / values
..CLOZ	.CLOSZ	EMT request layout / values
..CMAP	.CMAP/.GCMAP/.MSDS	EMT request layout / values
..CMKT	.CMKT	EMT request layout / values
..CNTX	.CNTXSW	EMT request layout / values
..CRAW	.CRAW	EMT request layout / values
..CRRG	.CRRG	EMT request layout / values
..CSIG	.CSIGEN	EMT request layout / values
..CSIS	.CSISPC	EMT request layout / values

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
..CSTA	.CSTAT	EMT request layout / values
..DATE	.DATE	EMT request layout / values
..DELE	.DELETE	EMT request layout / values
..DEVI	.DEVICE	EMT request layout / values
..DSTA	.DSTATUS	EMT request layout / values
..ELAW	.ELAW	EMT request layout / values
..ELRG	.ELRG	EMT request layout / values
..ENTE	.ENTER	EMT request layout / values
..EXIT	.EXIT	EMT request layout / values
..FETC	.FETCH	EMT request layout / values
..FPRO	.FPROT	EMT request layout / values
..GFDA	.GFDAT	EMT request layout / values
..GFIN	.GFINF	EMT request layout / values
..GFST	.GFSTA	EMT request layout / values
..GMCX	.GMCX	EMT request layout / values
..GTIM	.GTIM	EMT request layout / values
..GTJB	.GTJB	EMT request layout / values
..GTLI	.GTLIN	EMT request layout / values
..GVAL	.GVAL	EMT request layout / values
..HERR	.HERR	EMT request layout / values
..HRES	.HRESET	EMT request layout / values
..LIMI	.LIMIT	Assembler/linker directive
..LOCK	.LOCK	EMT request layout / values
..LOOK	.LOOKUP	EMT request layout / values
..MAP	.MAP	EMT request layout / values
..MRKT	.MRKT	EMT request layout / values
..MTAT	.MTATCH	EMT request layout / values
..MTDT	.MTDTCH	EMT request layout / values
..MTGE	.MTGET	EMT request layout / values
..MTIN	.MTIN	EMT request layout / values

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
..MTOU	.MTOUT	EMT request layout / values
..MTPR	.MTPRNT	EMT request layout / values
..MTRC	.MTRCTO	EMT request layout / values
..MTSE	.MTSET	EMT request layout / values
..MTST	.MTSTAT	EMT request layout / values
..MWAI	.MWAIT	EMT request layout / values
..PEEK	.PEEK	EMT request layout / values
..POKE	.POKE	EMT request layout / values
..PRIN	.PRINT	EMT request layout / values
..PROT	.PROTECT	EMT request layout / values
..PURG	.PURGE	EMT request layout / values
..PVAL	.PVAL	EMT request layout / values
..QSET	.QSET	EMT request layout / values
..RCTR	.RCTRLO	EMT request layout / values
..RCVD	.RCVD .RCVDC .RCVDW	EMT request layout / values
..READ	.READ .READC .READW	EMT request layout / values
..RELE	.RELEASE	EMT request layout / values
..RENA	.RENAME	EMT request layout / values
..REOP	.REOPEN	EMT request layout / values
..RSUM	.RSUM	EMT request layout / values
..SAVE	.SAVESTATUS	EMT request layout / values
..SCCA	.SCCA	EMT request layout / values
..SDAT	.SDAT .SDATC .SDATW	EMT request layout / values
..SDTT	.SDTTM	EMT request layout / values
..SERR	.SERR	EMT request layout / values
..SETT	.SETTOP	EMT request layout / values

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
..SFDA	.SFDAT	EMT request layout / values
..SFIN	.SFINF	EMT request layout / values
..SFPA	.SFPA	EMT request layout / values
..SFST	.SFSTA	EMT request layout / values
..SPCP	.SPCPS	EMT request layout / values
..SPFU	.SPFUN	EMT request layout / values
..SPND	.SPND	EMT request layout / values
..SRES	.SRESET	EMT request layout / values
..TLOC	.TLOCK	EMT request layout / values
..TRPS	.TRPSET	EMT request layout / values
..TTIN	.TTYIN/.TTINR	EMT request layout / values
..TTOU	.TTYOUT/.TTOUTR	EMT request layout / values
..TWAI	.TWAIT	EMT request layout / values
..UNLO	.UNLOCK	EMT request layout / values
..UNMA	.UNMAP	EMT request layout / values
..UNPR	.UNPROTECT	EMT request layout / values
..WAIT	.WAIT	EMT request layout / values
..WRIT	.WRITE .WRITC .WRITEW	EMT request layout / values
.%%%DF	–	Generic information about definition macros
.BBRDF	–	Bad Block Replacement table (in Home Block)
.BBSDF	–	Boot Block "Standard" info
.BOTDF	–	Memory usage in boot block
.BSRDF	–	Boot service routine structure
.BUPDF	–	BUP home block information
.CCLDF	–	CCL command string layout
.CF1DF	–	System configuration word
.CF2DF	–	Second configuration word
.CF3DF	–	Third configuration word

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
.CHADF	.CHAIN	Define chain calling arguments
.CHNDF	–	Define channel block layout
.CLDDF	–	CLI/IND structure
.CLIDF	–	Command Line Interpreter bytes
.CPUHD	–	CPU registers
.CSIDF	.CSIGEN .CSISPC	Define layout of CSI returned info
.CSTDF	.CSTAT	Define CSTAT return area
.CSWDF	.CSTAT	CSW bit definitions
.CVHDF	–	Overlay handler config word
.D2BDF	–	DUP to BSTRAP communications
.DATDF	–	RT–11 date format
.DBKDF	–	Define offsets in dblock
.DEVDF	–	Define handler IDs for dstat
.DFXDF	–	Default extension list for CSI%%%
.DIEDF	–	Directory entry definition
.DIHDF	–	Directory header definition
.DSCDF	–	Define handler codes for dstat
.DSEDF	.DRSET	DRSET generated entries
.DSKDF	–	Disk layout (by block)
.DSPDF	.DRSPF	.DRSPF data format
.DSTDF	.DSTATUS	.DSTATUS return area definition
.DTMDF	.SDTTM	Date / time setting block
.DUSDF	.DRUSE .DRTAB	DRUSE (and DRTAB) entry definition
.DVCDF	–	Device class codes
.DVLDF	.DEVICE	.DEVICE list format
.DVMDF	–	Device mode bits
.DVTDF	.DRVTB	.DRVTB layout
.DWHDF	–	DW area in home block

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
.DWTDF	–	Layout of DWTYPE in FIXED area
.E16DF	–	EMT16 list layout
.ELDDF	–	Error log file Device Error record format
.ELIDF	–	Error log file record types
.ELMDF	–	Error log file Memory Error record format
.ELSDF	–	Error log file header's device statistics record
.EMTDF	–	Define EMT codes and subcodes
.EPTDF	–	Object library entry point table entry
.ERMDF	.SPFUN	Define magtape handler SPFUN error/status codes
.ERNDF	.SPFUN	Define Ethernet handler SPFUN error codes /subcodes
.ERRDF	All	Define error codes in ERRBYT
.ERSDF	.SPFUN	Define DU/DM handler .SPFUN error codes
.ESBDF	.SPFUN	Define magtape SPFUN error/status block format
.FIXDF	.GVAL .PVAL	Define RMON fixed area
.FLBDF	–	Form library header
.FMTDF	–	Format argument block
.FPBDF	–	File prefix block
.FRKDF	.FORK	Fork element definition
.GRBDF	–	Global Region control block
.GTJDF	.GTJB	Define layout of GTJB returned info
.HANDF	–	Handler block 0 definition
.HBFDF	–	Handler Block 1 flag word bits (H1.FLG)
.HBGDF	.DRBEG	Define ".DRBEG" table layout
.HF2DF	–	H1.FG2 flag word bit definitions
.HINDF	–	Init/Restore area in home block
.HOMDF	–	Device home block def
.HNPDF	–	Handler "NOP" flag bits

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
.HS2DF	–	Second handler status word
.HSRDF	–	Handler service routines codes
.HSVDF	–	Home block system version values
.HUMDF	–	Define bits in H.64UM word
.HUSDF	–	User area of home block
.IBKDF	–	Define bits for I.BLOK
.IMPDF	–	Impure area layout
.INDDF	–	IND status byte bit definitions
.IOBDF	–	I/O Block (used for system I/O requests)
.ISTDF	–	Define bits for I.STAT
.JSWDF	–	JSW bits
.JSXDF	–	JSX bits
.LDADF	–	Absolute Binary File Format
.LOWDF	–	Low memory vector protection bitmap
.MCADF	–	Mapping Context Area (MCA)
.MEMDF	–	Memory info and routines
.MGTDF	.MTGET	Define .MTGET returned info
.MLBDF	–	MACRO library file
.MNTDF	–	Macro library name table entry
.MODDF	.MODULE	Define .MODULE area
.MONDF	–	Define offsets in system image files
.MSTDF	.MTSTAT	Define .MTSTAT returned info
.NALDF	.SPFUN	Define Ethernet SF.NAL SPFUN argument format
.NMUDF	.SPFUN	Define Ethernet SF.NMU SPFUN argument format
.NPMDF	.SPFUN	Define Ethernet SF.NPM SPFUN argument format
.NPRDF	.SPFUN	Define Ethernet SF.NPR SPFUN argument format
.NRWDF	.SPFUN	Define Ethernet SF.NRD/SF.NWR SPFUN argument format

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
.NSTDF	.SPFUN	Define Ethernet SPFUN status word format
.OBJDF	–	OBJ file entry formats
.OLBDF	–	Object library file
.OPTDF	.CSISPC	CSISPC option value return layout
.OTBDF	–	Disk overlay table definition
.OTJDF	–	Overlay jump table layout
.OVRDF	–	Overlay data area layout
.OWNDF	–	Ownership table layout
.P1XDF	–	Define area -> by P1EXT
.PGMDF	–	Code for default programs (edit/ftn)
.PRODF	–	PRO GET{VEC|CSR|ID} vector
.QCMDF	–	Completion queue element definition
.QFKDF	.FORK	Fork queue element definition
.QHKDF	–	$QHOOKS structure
.QSYDF	.SYNCH	Synch queue element definition
.QTIDF	.TWAIT	Timer queue element definition
.RCBDF	–	Region control block
.RELDF	–	Block 0 .REL image definition
.RGTDF	–	Replacement Geometry Table
.RTSDF	–	Return instruction et. al.
.SAVDF	–	Block 0 .SAV image definition
.SF%DF	.SPFUN	SPFUN function byte definition
.SFBDF	.SPFUN	Argument format/values for SF.BYP and SF.OBP (obsolete)
.SFCDF	.SPFUN	Argument values associated with SF.CTL
.SFDDF	.SPFUN	Disk SPFUN function byte definition
.SFKDF	.SPFUN	KXJ11–CA SPFUN function byte definition
.SFMDF	.SPFUN	Magtape SPFUN function byte definition
.SFNDF	.SPFUN	Ethernet SPFUN function byte definition
.SFODF	.SPFUN	Other SPFUN function byte definition

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
.SFSDF	.SPFUN	Argument values for SF.MST SPFUN
.SFTDF	.SPFUN	Argument values for SF.TAB and SF.MTB SPFUNs
.SFUDF	.SPFUN	Argument format for SF.USR SPFUN
.SFXDF	.SPFUN	X(C,S,L) SPFUN function byte definition
.SFZDF	.SPFUN	Arguments for SF.SIZ request
.SGNDF	–	System generation options
.SPCDF	.SPCPS	SPCPS block
.SPLDF	–	Spooler word bit definitions
.STWDF	–	DCL and IND state word
.SWIDF	–	Switch parsing table (ICSI et. al.)
.SYCDF	–	Syscom area
.TASDF	.MTATCH	Asynchronous status word supplied by user
.TC2DF	–	Terminal status word 2
.TCBDF	–	Terminal Control block Definition
.TCFDF	–	Terminal configuration bits
.THKDF	–	Multiterminal hooks for handlers
.TIMDF	.GTIM	Time word pair
.TSTDF	–	Terminal Status word
.TTCDF	–	TTCNFG area definitions
.UBVDF	–	Entry vector in UBX.SYS
.UEBDF	–	User error byte codes
.USSDF	–	Codes for special directory requests
.VERDF	–	Version and release area
.VTBDF	–	Virtual overlay table definition
.WCBDF	–	Window control block
.X%%DF	.CHAIN	Chain to %%* arguments layout (except .XITDF)
.XBADF	.CHAIN	Chain to BATCH argument format
.XCRDF	.CHAIN	Chain to CREF argument format

Table A–1 (Cont.): SYSTEM.MLB Macros and Their Related Programmed Requests

SYSTEM.MLB Macro	Related Programmed Request	Comments
.XEDDF	.CHAIN	Chain to EDIT argument format
.XHEDF	.CHAIN	Chain to HELP argument format
.XITDF	.CHAIN	Define EXIT argument area
.XKEDF	.CHAIN	Chain to (and from) KED argument format
.XLDDF	.CHAIN	Chain to (and from) LD argument format
.XQUDF	.CHAIN	Chain to QUEMAN argument format
.XSPDF	.CHAIN	Chain to SPLIT argument format
.XTEDF	.CHAIN	Chain to TECO argument format
.XXXDF	.CHAIN	Chain to (and from) "standard" argument format
.Y%%DF	–	Structure used by %%* and other components
.YCRDF	–	Entry for CREF file
.YQUDF	–	Message format for QUEMAN / QUEUE
.YSPDF	–	SP / SPOOL communications area
.Z%%DF	.SPFUN	Layout of table for SPFUN SF.TAB for %% handler
.ZDUDF	.SPFUN	DU table definition
.ZLDDF	.SPFUN	LD table definition
.ZMUDF	.SPFUN	MU table definition
.ZUBDF	.SPFUN	UB table definition

Appendix B
Software Simulation Of The Console Terminal Hardware

B.1 Introduction

RT–11 provides an interface into the terminal handler that lets you simulate the console hardware under all monitors.

The console hardware can be simulated under single terminal and multiterminal monitors. The program can check the MTTY$ bit in $SYSGE to determine if the system uses multiterminal support.

The data and instructions in the simulation code must reside in kernel memory and must not be in PAR1 as they are accessed by the monitor from interrupt code.

B.2 Single Terminal Software Simulation

Simulating the console terminal in a single terminal system requires redirecting either the terminal input or the output registers or both. Then, an interrupt to a redirected hardware register is directed to the software emulation.

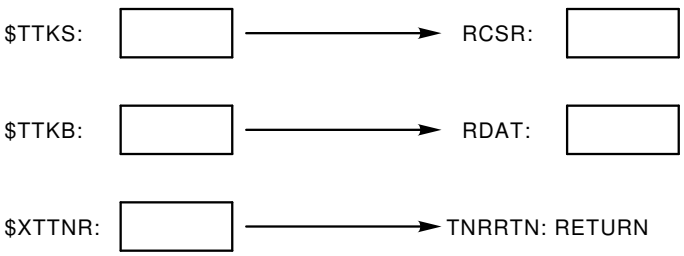
B.2.1 Data Structures

The data structures required to support console terminal software simulation are found in the following RMON fixed area offsets:

Offset	Name	Contents
304	$TTKS	Pointer to input CSR (RCSR)
306	$TTKB	Pointer to input DATA register (RDAT)
310	$TTPS	Pointer to output CSR (XCSR)
312	$TTPB	Pointer to output DATA register (XDAT)
372	$SYSGE	MTTY$ bit indicating nonmultiterminal or multiterminal support
470	$XTTNR	Pointer to a routine to be called when there is no room for input characters in the terminal input buffer.
476	$XTTPS	Pointer to routine called before reading @TTKS and after changing @TTKS
500	$XTTPB	Pointer to routine called after changing @TTKB

B.2.2 Setting Up the Terminal Input Software Simulation

The following figure illustrates the structure of the hardware terminal input data structures before software simulation.



Use the following procedure to replace the hardware terminal input with the software simulation:

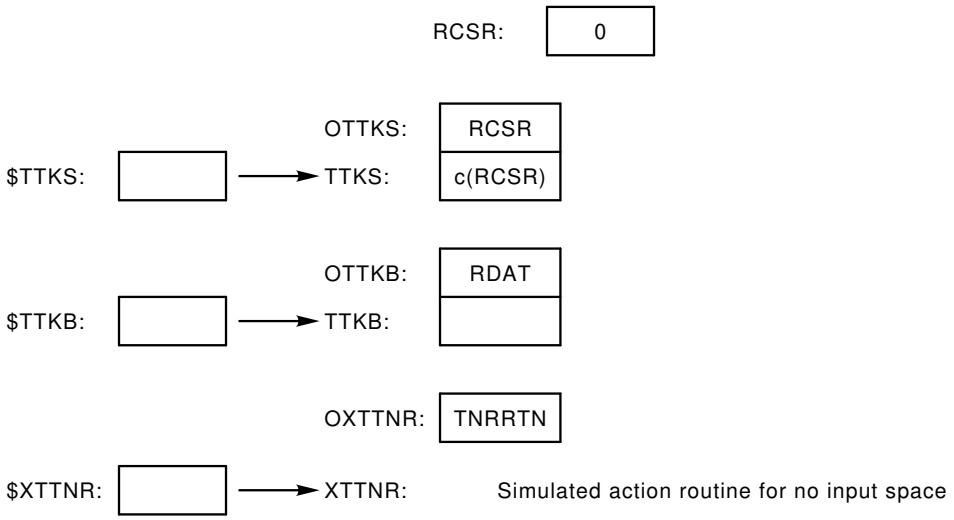
1. Raise the processor priority to prevent interrupts and other interference.
2. Save the contents of RMON fixed offsets $TTKS and $TTKB, preserving the addresses of the hardware registers RCSR and RDAT:

   Copy $TTKS to location OTTKS.

   Copy $TTKB to location OTTKB.
3. Change the contents of those offsets to point to locations in memory that serve as the software simulated RCSR and RDAT. You could call those locations TTKS and TTKB, respectively.

   The monitor then accesses TTKS and TTKB for terminal input, rather than the actual hardware registers.
4. Save the contents of RMON fixed offset $XTTNR by copying it to location OXTTNR.
5. Change the contents of $XTTNR to point to a routine in your code that simulates the hardware console's XOFF 'bell' when there is no room for a character in the monitor's terminal input buffer.
6. Lower the processor priority.

The following figure illustrates the simulated terminal input structure after performing those steps.



B.2.2.1 Providing an Interrupt Source

The software must simulate the action of a DL11, by using interrupts to notify the monitor of input characters, as follows:

- Hook into the line time clock interrupt vector and use it as the interrupt source.

  Save the current value in location 100.

- Replace the contents of 100 with the address of an interrupt routine in the simulation code.
- The interrupt routine determines if a console input interrupt is needed. If so, the routine can use the PIRQ hardware (if necessary) to post such an interrupt.
- The interrupt routine then jumps to the original contents of 100 to process the normal clock code.

B.2.2.2 Operation

The following list is a guide to operating the simulated terminal input.

- Entry from interrupt source

  When the simulation code is entered by the interrupt source:

  1. Check if there is a character in RECHAR or an active buffer with a character available.

     If there is a character in RECHAR, place it in location TTKB and clear RECHAR.

     If there is an active buffer with a character available, place it in TTKB and in LCHAR and update the buffer pointer and count.

     Otherwise, just jump out, done.

2. Set the DONE bit in TTKS.
3. If the IENABL bit is set in TTKS, then post an interrupt for the console input vector (60).

- Entry from XTTNR

  If the simulation code is entered from XTTNR (as a result of a full monitor input buffer):

  1. Move the character in LCHAR to RECHAR, setting up a retry for this character.
  2. Pop the top of the stack and do a RETURN (to return to the caller's caller) which will suppress sending a BELL char in response buffer overflow.

- Posting interrupts

  The input simulation cannot directly simulate an interrupt because the monitor may be using a priority level to synchronize operations. Instead of direct simulation, use the PIRQ hardware to post and deliver interrupts.

  Interrupt dispatching for input or output (but not both) simulation is fairly simple. The interrupt dispatching for both input and output simulation is more complicated; PIRQ interrupts must be sorted out separately for input and output simulation.

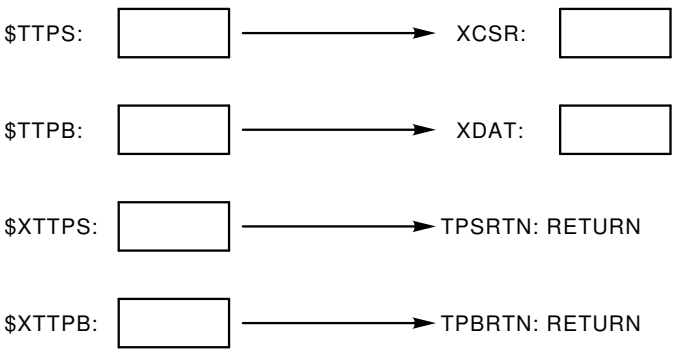
- Unloading the input simulation software

  Unload the input simulation software by restoring the structures to their premodified state, using the values you stored when loading the software.

  Restore $TTKS from OTTKS.
  Restore the hardware RCSR from TTKS.
  Restore $TTKB from OTTKB.
  Restore $XTTNR from OXTTNR.

B.2.3 Setting Up the Terminal Output Software Simulation

The following figure illustrates the structure of the hardware terminal output data structures before software simulation:



Use the following procedure to replace the hardware terminal output with the software simulation:

1. Raise the processor priority to prevent interrupts and other interference.
2. Save the contents of RMON fixed offset $TTPS, preserving the address of the hardware register XCSR.

   Copy $TTPS to OTTPS.

   Copy @$TTPS to PTTPS to create a second copy of XCSR. PTTPS is discussed later.
3. Clear @$TTPS to prevent interrupts from the actual hardware.
4. Change the contents of $TTPS to point to a location in memory that serves as the software simulated XCSR; you could name that location TTPS.

   The monitor then accesses TTPS for terminal output, rather than the actual hardware register.
5. Save the contents of RMON fixed offset $TTPB, preserving the address of the hardware register XDAT:

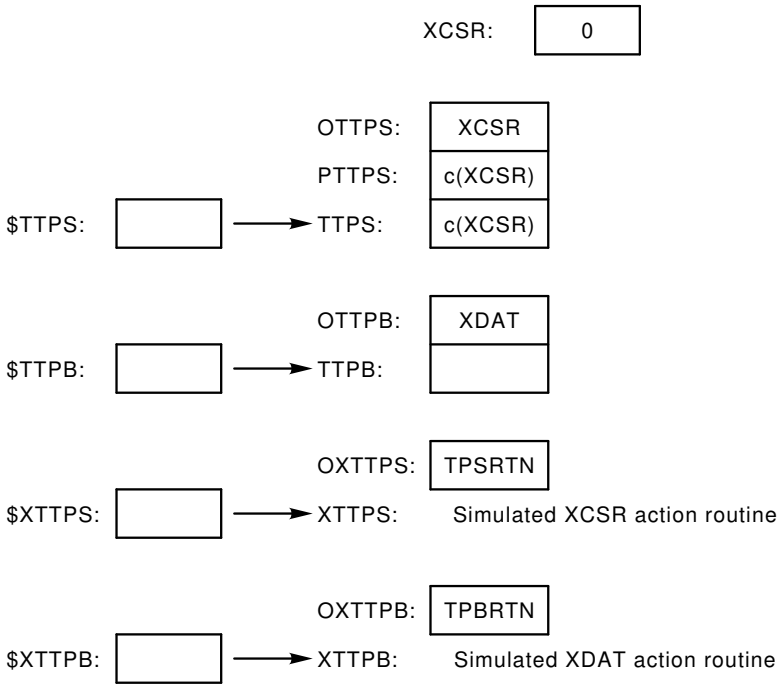
   Copy $TTPB to OTTPB.
6. Change the contents of $TTPB to point to a location in memory that serves as the software simulated XDAT; you could name that location TTPB.

   The monitor then accesses TTPB for terminal output, rather than the actual hardware register.
7. Save the contents of RMON fixed offset $XTTPS, by copying it to location OXTTPS.

   Change the contents of $XTTPS to point to XTTPS, the simulated XCSR action routine.
8. Save the contents of RMON fixed offset $XTTPB, by copying it to location OXTTPB.

   Change the contents of $XTTPB to point to XTTPB, the simulated XDAT action routine.
9. Lower the processor priority.

The following figure illustrates the simulated terminal output structure after performing those steps.



B.2.3.1 Operation

The following list is a guide to programming the simulated terminal output.

- Calling the XTTPS simulated action routine.

  The monitor calls XTTPS before it reads or after it modifies TTPB. If the monitor modifies and then reads TTPB, it calls XTTPS only once, between the modify and the read.

  Do the following when calling XTTPS:

  1. Save the current condition codes and any work registers.
  2. Copy the DONE bit (200) setting from PTTPS to TTPS. If the DONE bit is SET, check the IENABLE bit (100) in PTTPS.

     If the IENABLE bit in PTTPS is clear and the IENABLE bit in TTPS is set, post an interrupt for the console output interrupt vector (64).
  3. Copy TTPS to PTTPS.
  4. Restore the saved condition codes and any work registers.

- Calling the XTTPB simulated action routine.

  The monitor calls XTTPB after it modifies TTPB.

  Do the following when calling XTTPB:

  1. Save the current condition codes and any work registers.

2. Pick up the low byte of TTPB as the next output character.
3. Set the DONE bit in PTTPS and TTPS.
4. If the IENABLE bit (100) in TTPS is set, post an interrupt for the console output terminal vector (64).
5. Restore the saved condition code and any work registers.

- Posting interrupts

  The output simulation cannot directly simulate an interrupt because the monitor may be using a priority level to synchronize operations. Instead of direct simulation, use the PIRQ hardware to post and deliver interrupts.

  Interrupt dispatching for input or output (but not both) simulation is fairly simple. The interrupt dispatching for both input and output simulation is more complicated; PIRQ interrupts must be sorted out separately for input and output simulation.

- Unloading the output simulation software

  Unload the output simulation software by restoring the structures to their premodified state, using the values you stored when loading the software.

  Restore $TTPS from OTTPS.
  Restore the hardware XCSR from TTPS.
  Restore $TTPB from OTTPB.
  Restore $XTTPS from OXTTPS.
  Restore $XTTPB from OXTTPB.

B.3 Multiterminal Software Simulation

Simulating the console terminal in a multiterminal system requires redirecting a single pointer, T.CSR, rather than the four separate pointers in a single terminal configuration. The input and output CSR and data registers are offset from the address pointed to by T.CSR. This is the major difference between simulating the hardware console terminal with a multiterminal monitor, as opposed to the single terminal monitor.

B.3.1 Data Structures

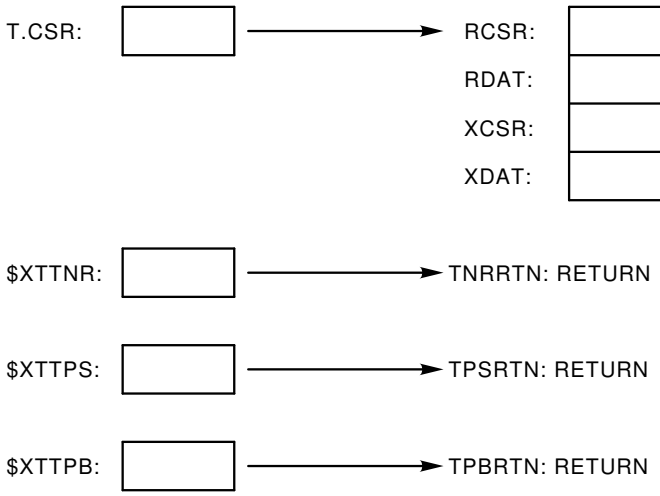
Besides the data structures described in Section B.2.1, support for multiterminal console terminal simulation requires:

Location	Offset	Name	Contents
RMON Fixed	446	$IMPLO	Pointer to background impure area.

Location	Offset	Name	Contents
Impure Area	016	I.CNSL	Pointer to TCB.
TCB	014	T.STAT	DZ11$ and DH11$ bits; if either set, then not a DL11 interface (and operation is invalid).
	016	T.CSR	Pointer to input CSR; registers are addressed as offsets from this value.
RMON Fixed	466	$CNFG3	Contains CF3.1S bit; if set, indicates support for one DL11 interface under multiterminal monitor.

B.3.2 Setting Up the Multiterminal Software Simulation

The following figure illustrates the structure of the multiterminal monitor console terminal data structures before loading the software simulation. Notice that the CSR and data registers are offset from the address pointed to by T.CSR.



Use the following procedure to replace the hardware multiterminal console terminal with the software simulation:

1. Raise the processor priority to prevent interrupts and other interference.
2. Search the table just before $IMPLO in the RMON fixed offset area for the address of the background impure area.

   If handler load code is being run, then the current job is the background and the value in offset $CNTXT can be used.
3. Get the address for the job's console TCB from the I.CNSL word in the TCB.

4. Check the T.STAT word in the TCB to make sure both DZ11$ and DH11$ bits are clear. If either is set, software simulation cannot be used.
5. Save T.CSR by copying it to OTCSR.
6. Save the address pointed to by T.CSR by copying it to TTKS, the simulated RCSR.
7. Clear the hardware RCSR (pointed to by T.CSR) to prevent interrupts from the actual hardware.
8. Copy the contents of the hardware XCSR (pointed to by T.CSR+4) to TTPS, the simulated XCSR.

   Also copy the contents of the hardware XCSR to PTTPS, creating a second copy of XCSR at PTTPS.

   Then, clear TTPS to prevent interrupts from the actual hardware.
9. Change the contents of T.CSR to point to TTKS.

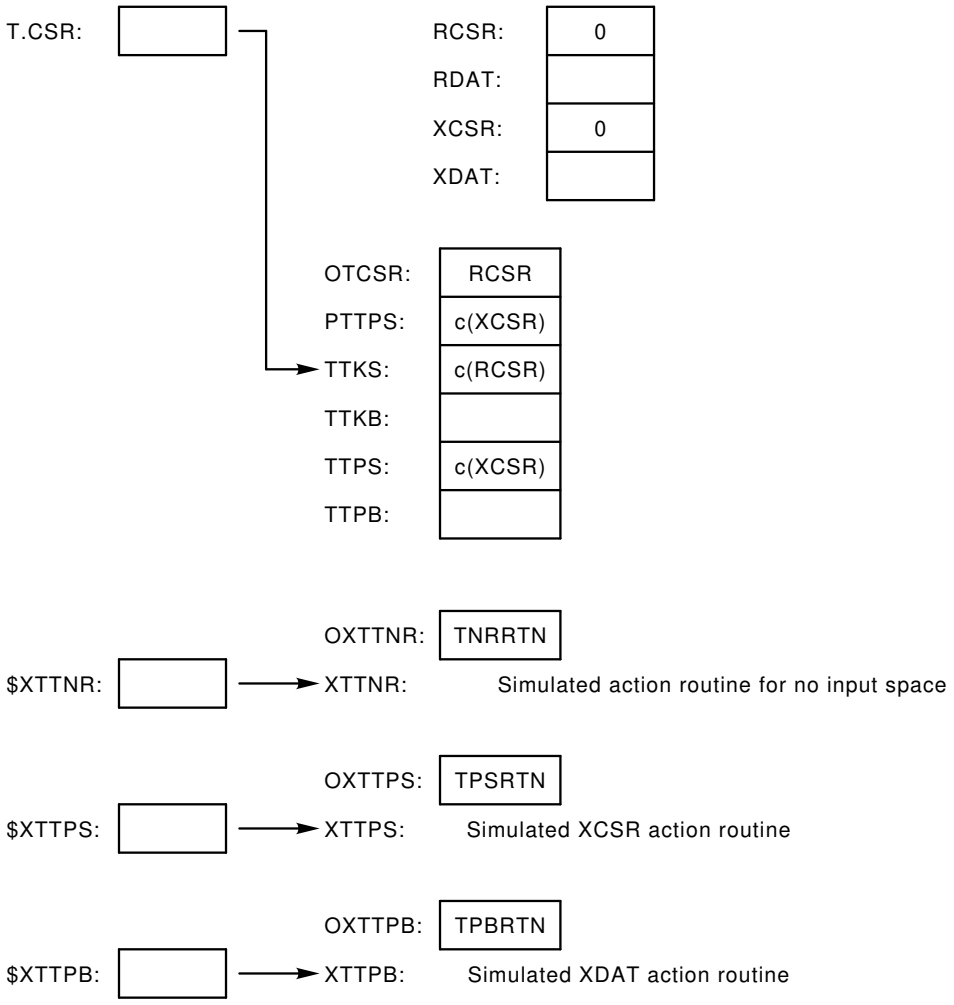
   The three other simulated registers, TTKB, TTPS, and TTPB, are located after TTKS in memory.
10. Copy $XTTNR to OXTTNR.

    Then, change the contents of $XTTNR to point to XTTNR, the routine to be called when there is no room for input characters at the terminal input buffer.
11. Copy $XTTPS to OXTTPS.

    Then, change the contents of $XTTPS to point to XTTPS, the simulated XCSR action routine.
12. Copy $XTTPB to OXTTPB.

    Then, change the contents of $XTTPB to point to XTTPB, the simulated XDAT action routine.
13. Lower the processor priority.

The following figure illustrates the simulated multiterminal monitor input/output terminal structure after performing those steps.



B.3.2.1 Identify the Console Terminal

The need to identify the console terminal arises when there is more than one DL11 interface on the system. Check CF3.1S; if set, there is one DL11 interface and no further identification is necessary.

If CF3.1S is clear, the console replacement code must determine if calls to the action routines ($XXTNR, $XTTPS, $XTTPB) are for the supplanted terminal or another. When an action routine is called, the contents of R4 should be compared with the saved value OTCSR. If they are the same, the action routine is directed at the correct terminal.

B.3.2.2 Operation

The software simulation of the hardware console terminal is the same under multiterminal and single terminal monitors. See Section B.2.2.1 and Sections B.2.2.2 for the input simulation procedure and Section B.2.3.1 for the output simulation procedure.

Unload the input simulation software by restoring the structures to their premodified state, using the values you stored when loading the software.

Restore T.CSR from OXTTPB.
Restore RCSR from TTKS.
Restore XCSR from TTPS.
Restore $XTTNR from OXTTNR.
Restore $XTTPS from OXTTPS.
Restore $XTTPB from OXTTPB.