RSTS/E
MONITOR INTERNALS

Michael Mayfield

May 1983

This manual contains detailed information about the internal format and operation of the RSTS/E monitor, its disk structures, resident libraries and runtime systems. It is intended for the advanced system programmer.

OPERATING SYSTEM AND VERSION: RSTS/E V8.0

SUPERSESSION/UPDATE INFORMATION: This printing supersedes and replaces the November 1982 printing.

NORTHWEST DIGITAL SOFTWARE, INC. BOX 2-743 NEWPORT, WA 99156 (509) 447-2620

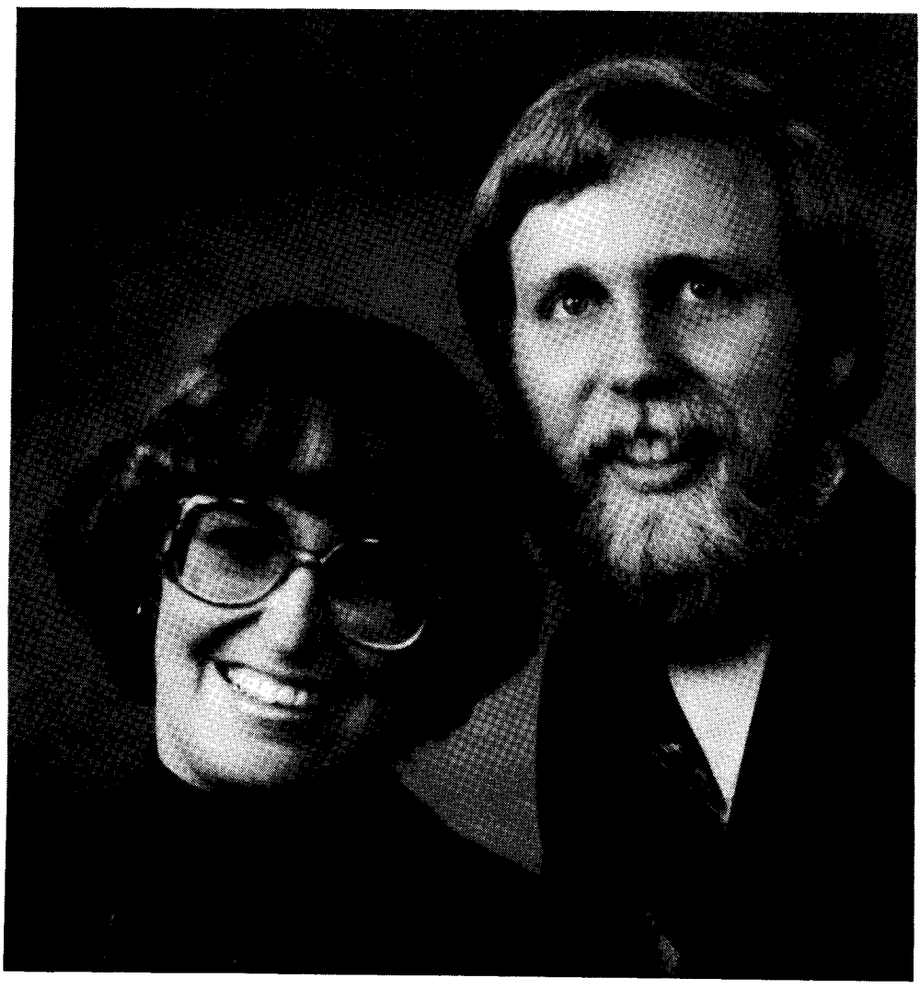
Published and sponsored by M-Systems, Inc., publishers of the VAX/RSTS Professional, DEC Professional, and Personal and Professional magazines. This document was produced entirely by computer typesetting on RSTS/E by Data & Staff, Inc., Seattle, Washington.

DEDICATION

This document is dedicated to the members of the RSTS/E development group. Over a period of more than ten years, these dedicated professionals have consistently produced a product that is unsurpassed in capabilities, performance and ease of use. From all of us who make our livings with RSTS/E and Digital products, I say "Thank you, and keep up the good work".

This document is also dedicated to my wife, Barbara, for the love and support she has shown through the years. Without her help and understanding during the endless overnight sessions and the months away from home, I never would have been able to provide a document such as this one.

ABOUT THE AUTHOR



Michael Mayfield started in RSTS over 10 years ago and has been going strong ever since. Few people know as much about RSTS/E and how to get the most from it as Mike. During his career he has been involved in the design and development of many successful software products for RSTS/E, including RPM, WS-11, CT*OS, WORD-11/DECWORD, and SAVER.

Mr. Mayfield presently heads Northwest Digital Software, Inc., a RSTS/E development and consulting firm located in Newport, Washington. As such, he is involved in the development of an array of quality products for the RSTS/E environment. He is also involved in providing consulting services in the area of RSTS/E performance optimization, product design and RSTS/E monitor extensions such as device drivers and custom modifications.

Mr. Mayfield received his B.S. in Information and Computer Science from the University of California at Irvine, graduating with honors. From there he worked with Digital Equipment Corporation and Data Processing Design, Inc. for several years before starting his own organization.

TABLE OF CONTENTS

Preface x

Introduction xi

Summary of Changes xii

Chapter 1: DISK STRUCTURES 1-1
1.1 TERMINOLOGY 1-2
1.1.1 Logical Block Number(LBN) 1-2
1.1.2 Cluster 1-2
1.1.3 Device Cluster Size(DCS) 1-2
1.1.4 Device Cluster Number(DCN) 1-2
1.1.5 FIP Block Number(FBN) 1-2
1.1.6 Pack Cluster Size 1-2
1.1.7 File Cluster Size 1-3
1.1.8 MFD Cluster Size 1-3
1.1.9 GFD Cluster Size 1-3
1.1.10 UFD Cluster Size 1-3
1.1.11 FIP Block Sub-block(FBB) 1-3
1.1.12 Directory Link 1-3
1.2 RDS0 DISK DIRECTORY STRUCTURES 1-4
1.2.1 MFD Label Entry 1-5
1.2.1.1 MFD Pack Status 1-5
1.2.2 MFD Name Entry 1-6
1.2.2.1 USTAT - UFD Status 1-7
1.2.3 MFD Accounting Entry 1-7
1.2.4 MFD Cluster Map 1-8
1.2.5 User File Directory (UFD) 1-8
1.3 RDS1 DISK DIRECTORY STRUCTURES 1-9
1.3.1 Pack Label 1-10
1.3.1.1 Pack Status 1-11
1.3.2 Master File Directory (MFD) 1-11
1.3.2.1 Label Block 1-11
1.3.2.2 GFD Pointer Block 1-13
1.3.2.3 Other Blocks 1-13
1.3.3 Group File Directory (GFD) 1-13
1.3.3.1 Label Block 1-14
1.3.3.2 UFD Pointer Block 1-15
1.3.3.3 Name Entry Pointer Block 1-15
1.3.3.4 Other Blocks 1-16
1.3.3.4.1 Name Entry 1-16
1.3.3.4.2 Accounting Entry 1-17
1.3.3.4.3 Attribute Entries 1-18
1.3.4 User File Directory (UFD) 1-19
1.3.4.1 UFD Label Entry 1-19
1.3.4.2 UFD Name Entry 1-20
1.3.4.2.1 USTAT - File Status 1-21
1.3.4.3 UFD Accounting Entry 1-21
1.3.4.4 UFD Attributes Entry 1-22
1.3.4.4.1 RMS-11 Attribute Values 1-22
1.3.4.5 UFD Retrieval Entry 1-23
1.3.4.6 UFD Cluster Map 1-24
1.3.4.7 Unused entries 1-24

1.3.5 Directory Links 1-25
1.4 STORAGE ALLOCATION TABLE (SAT) 1-26
1.5 SAVE IMAGE LIBRARY (SIL) 1-27
1.5.1 SIL Index Block 1-27
1.5.2 SIL Module Entry 1-27
1.5.3 SIL Symbol Table Entry 1-29
1.5.4 SIL Overlay Descriptor Entry 1-29
1.6 BOOTSTRAP BLOCK 1-30
1.7 BAD BLOCK FILE 1-33

Chapter 2: MONITOR TABLES 2-1
2.1 JOB CONTROL 2-2
2.1.1 JOBTBL - Job Table 2-2
2.1.2 JDB - Primary Job Data Block 2-3
2.1.2.1 JDFLG - Primary Job Status Flags 2-4
2.1.2.2 JDFLG2 - Secondary Job Status Flags 2-5
2.1.3 JDB2 - Secondary Job Data Block 2-6
2.1.4 IOB - I/O Block 2-7
2.1.5 WRK - Work Block 2-8
2.1.6 JBWAIT,JBSTAT - Job Status Tables 2-8
2.1.7 JOBCLK - Job Sleep Time Table 2-9
2.2 LEVEL THREE QUEUE 2-10
2.3 MEMORY CONTROL 2-12
2.3.1 MCB - Memory Control Sub-Block 2-12
2.3.1.1 M.CTRL - Memory Status Information 2-13
2.3.2 MEMLST - Resident Memory List 2-13
2.3.2.1 Root Memory Control Sub-Block 2-13
2.3.2.2 Tail Memory Control Sub-Block 2-14
2.3.3 RESLST - Desired Residency List 2-14
2.3.4 RTS - Runtime System Descriptor Block 2-15
2.3.4.1 R.FLAG - Runtime System Characteristics 2-16
2.3.4.2 NULRTS - Disappearing RSX Runtime System 2-16
2.3.4.3 RTSLST - Runtime System List 2-17
2.3.5 LIB - Resident Library Descriptor Block 2-17
2.3.5.1 R.FLAG - Resident Library Characteristics 2-19
2.3.6 WDB - Window Descriptor Block 2-19
2.3.6.1 W.WIN? - Address Windows 2-20
2.3.6.1.1 W$NSTS - Window Status 2-21
2.3.6.2 Extended Window Descriptor Blocks 2-21
2.3.6.3 Resident Library Linkages 2-22
2.4 FILE CONTROL 2-23
2.4.1 File Control Block 2-24
2.4.1.1 F$STAT - Status Flags 2-25
2.4.1.2 FBB - FIP Block Sub-Block 2-25
2.4.2 WCB - Window Control Block 2-25
2.4.2.1 W$STS - Status Flags 2-27
2.4.2.2 W$FLAG - Flag Bits 2-27
2.4.2.3 W$WCB - File Flags and Link to WCB 2-28
2.4.2.4 Extended WCB 2-28
2.5 DEVICE CONTROL 2-29
2.5.1 DDB - Device Data Block 2-29
2.5.1.1 DDSTS - Device Characteristics Flags 2-30
2.5.1.2 DDCNT - Device Flags 2-31
2.5.1.3 Small Buffer Control Block 2-31
2.5.2 DSQ - Disk I/O Queue Entry 2-32

2.5.3 Logical Device Tables 2-34
2.5.3.1 DEVNAM - Device Name Table 2-34
2.5.3.2 DEVCNT - Device Unit Count Table 2-35
2.5.3.3 DEVPTR - Device Information Pointer Table 2-35
2.5.3.4 UNTCNT - Disk Unit Status Table 2-36
2.5.3.5 DEVTBL - DDB Pointer Table 2-36
2.5.3.6 LOGNAM - Logical Name Table 2-36
2.5.4 Device Driver Dispatch Tables 2-38
2.5.5 Device Driver Support Tables 2-38
2.5.5.1 FLGTBL - Device Characteristics Flags 2-38
2.5.5.2 SIZTBL - Line Width 2-39
2.5.5.3 BUFTBL - Default I/O Buffer Size 2-39
2.5.5.4 JSBTBL - JS.xx Bit for Device 2-40
2.5.5.5 DVRAP5 - APR5 Pointers 2-40
2.5.5.6 CSRTBL - CSR Addresses 2-40
2.5.5.7 TIMTBL - Timeout Counters 2-40
2.5.6 Disk Control Tables 2-41
2.5.6.1 DEVCLU/CLUFAC - Device Cluster Size, Cluster Ratio 2-41
2.5.6.2 UNTCLU/UNTERR - Pack Cluster Size, Error Count 2-41
2.5.6.3 UNTLVL/UNTREV - Disk unit RDS Level/Revision 2-41
2.5.6.4 UNTSIZ - Disk Size 2-41
2.5.6.5 UNTLIB - [1,2] Starting Cluster 2-42
2.5.6.6 MFDPTR - MFD Pointers 2-42
2.5.6.7 UNTOWN/UNTOPT - Unit Owner, Unit Options 2-42
2.5.6.8 DSKMAP - FUN to Disk Index 2-42
2.5.7 SAT Tables 2-43
2.5.7.1 SATCTL, SATCTM - Count of Free Blocks 2-43
2.5.7.2 SATPTR - DCN of Last Allocated Cluster 2-43
2.5.7.3 SATEND - Ending PCN 2-43
2.5.7.4 SATSTL, SATSTM - Starting FBN of SATT.SYS 2-43
2.5.8 DECNET Device Control Tables 2-43
2.5.8.1 DDCTBL - Number of DECNET Controllers, UCTTBL Bias 2-44
2.5.8.2 UCTTBL - Number of Units per Controller 2-44
2.6 SEND-RECEIVE 2-45
2.6.1 RIB - Receiver ID Block 2-45
2.6.1.1 S.OBJT - Object Types 2-46
2.6.1.2 S.ACCS - Access Control Bits 2-47
2.6.2 PMB - Pending Message Block 2-47
2.6.2.1 P$TYPE - Message Type 2-48
2.6.2.2 Buffer Format 2-49
2.7 CCL - CONCISE COMMAND LANGUAGE BLOCK 2-50
2.8 FIXED MEMORY LOCATIONS 2-51

Chapter 3: DEVICE DRIVERS 3-1
3.1 GENERAL STRUCTURE 3-2
3.1.1 PSECT Usage 3-3
3.1.1.1 xxDVR - Driver Read-Only Code Section 3-3
3.1.1.2 xxDINT - Interrupt Service Dispatch Section 3-3
3.1.1.3 xxDCTL - Read-Write Data Section 3-4
3.1.1.4 xxDTBL - Read-Only Data Section 3-4
3.2 ENTRY POINTS 3-5
3.2.1 ASN$xx - Assign 3-5
3.2.2 DEA$xx - Deassign 3-5
3.2.3 OPN$xx - Open 3-6
3.2.4 CLS$xx - Close 3-6

3.2.5 SER$xx - I/O Service 3-7
3.2.6 SPC$xx - Special Service 3-8
3.2.7 INT$xx - Interrupt Service 3-8
3.2.8 TMO$xx - Timeout 3-9
3.2.9 ERL$xx- Error Logging 3-10
3.2.10 SLP$xx - Sleep Check 3-10
3.2.11 UMR$xx - Unibus Mapping Register Available 3-11
3.2.12 L3Q$xx - Level Three Queue Reentry 3-11
3.3 SYMBOLIC VALUES 3-12
3.3.1 STS.xx - DDB Status Byte 3-12
3.3.2 FLG.xx - Device Dependent Flags 3-12
3.3.3 SIZ.xx - Line Width 3-13
3.3.4 BUF.xx - I/O Buffer Size 3-13
3.3.5 CNT.xx - Number of Units for Device 3-13
3.3.6 DDS.xx - DDB Size 3-13
3.3.7 CCC.xx - !C Flag 3-14
3.3.8 BFQ.xx - Buffer Quota 3-14
3.3.9 HOR.xx - Horizontal Line Width 3-14
3.3.10 SLP.xx - Check Before Sleeping Flag 3-15
3.3.11 UMR.xx - Notify Driver When UMR is Available 3-15
3.3.12 ALT.xx - Alternate Device Name 3-15
3.3.13 TIM.xx - Timeout Clock Setting 3-16
3.3.14 IDX.xx - Driver Index 3-16
3.3.15 JS.xx - JBWAIT/JBSTAT Status Bit 3-16
3.3.16 CSR.xx - Pointer to CSR Address 3-17
3.3.17 DEV.xx - Pointer to DDB for Unit n 3-17
3.3.18 xxDDDB - Address of Unit 0 DDB 3-17
3.3.19 LOG$xx - Enter Error Logging 3-18
3.3.20 WAIT2T - Reenter After Two Clock Ticks 3-18
3.4 SYSTEM MACROS 3-19
3.4.1 .BR - Branch to Following Location 3-19
3.4.2 .CALLR - Call Following Routine and Return 3-19
3.4.3 PUSH - Push a Value on the Stack 3-19
3.4.4 POP - Pop a Value from the Stack 3-20
3.4.5 .ASSUME - Verify Assumption 3-20
3.4.6 REGSAV - Save Registers R0-R5 3-20
3.4.7 REGSCR - Save Registers Co-Routine 3-21
3.4.8 REGRES - Restore Registers 3-21
3.4.9 DEVICE - Define Device Driver Information 3-21
3.4.10 BUFFER - Allocate/Deallocate Small Buffers 3-22
3.4.11 GETUSR - Get Byte from User Buffer 3-23
3.4.12 PUTUSR - Store Byte in User Buffer 3-23
3.4.13 SETERR - Post Error Code 3-24
3.4.14 ERROR - Post Error Code and Exit 3-24
3.4.15 L3QSET - Set Bit in L3Q 3-24
3.4.16 MAP - Access Memory Management Registers 3-25
3.4.17 SPL - Set Processor Priority 3-25
3.4.18 SPLC - Set Program Status Word 3-26
3.4.19 CRASH - Crash the System 3-26
3.5 MONITOR SUBROUTINES 3-27
3.5.1 FREBUF - Check Small Buffer Availability 3-27
3.5.2 STORE - Store Character in Small Buffer 3-27
3.5.3 FETCH - Get Character from Small Buffer 3-28
3.5.4 CLRBUF - Return All Small Buffers 3-28
3.5.5 BUFFER - Allocate a Large Buffer 3-28

3.5.6 RETCHN - Return All Large Buffers in a Chain 3-29
3.5.7 INTSAV - Enter Interrupt Service Routine 3-29
3.5.8 INTSVX - Enter Secondary Interrupt Service Routine 3-30
3.5.9 IOFINI - I/O Finished 3-30
3.5.10 IOFINC - I/O Conditionally Finished 3-31
3.5.11 IOREDO - Stall for I/O Redo 3-31
3.5.12 RTI3 - Exit and Check Level Three Queue 3-31
3.5.13 RETDEV - Return an Open Device 3-31
3.5.14 ERLDVR - Enter Error Log Entry 3-32
3.5.15 QUEUER - Enter Item in Queue 3-34
3.5.16 FNDJOB - Force a Job Into Memory 3-34
3.5.17 UNLOCK- Unlock a Job's Memory 3-35
3.5.18 LRRSQ - Clear Residency Quantum 3-35
3.5.19 DMPJOB - Dump the Current Job 3-35
3.5.20 MAPBUF - Map a Buffer 3-36
3.5.21 GETUMR - Get a Unibus Mapping Register 3-36
3.5.22 HMADR - Initialize Unibus Mapping for RH70 or RH11 3-37
3.5.23 RELUMR - Return a Unibus Mapping Register 3-37
3.6 CSR AND VECTOR ASSIGNMENT 3-38
3.7 INSTALLING THE DRIVER 3-40
3.8 DEBUGGING A DEVICE DRIVER 3-42
3.8.1 Notation 3-42
3.8.2 Control Registers 3-43
3.8.3 Control Tables 3-44
3.8.4 Commands 3-45

Chapter 4: RESIDENT LIBRARIES AND RUNTIME SYSTEMS 4-1
4.1 ADDRESS SPACE USAGE 4-3
4.1.1 Address Requirements for a Resident Library 4-3
4.1.2 Address Requirements for a Runtime System 4-3
4.1.3 Writing Position Independent Code 4-5
4.2 SPECIAL LIBRARIES AND RUNTIME SYSTEMS 4-7
4.2.1 Memory Resident Overlays Within a Library 4-7
4.2.2 Common Data Areas 4-7
4.2.3 Coordinating Runtime Systems 4-9
4.3 BUILDING A RESIDENT LIBRARY 4-10
4.3.1 Building a Position Dependent Library 4-10
4.3.2 Building a Position Independent Library 4-10
4.3.3 Building a Read-Write Resident Library 4-11
4.4 BUILDING A RUNTIME SYSTEM 4-12
4.5 DEBUGGING A RUNTIME SYSTEM 4-13

Appendix A: QUICK REFERENCE CHARTS
Appendix B: EXAMPLE PEEK SEQUENCES
Appendix C: EXAMPLE DEVICE DRIVER
Appendix D: EXAMPLE RUNTIME SYSTEM

References
Index

PREFACE

This document describes the internal workings and data structures of the RSTS/E monitor, as of version 8.0.

The information in this document will be useful for accessing information contained in tables within the monitor, the disk directory structure, and other monitor tables and structures. It provides the information required to write a custom device driver and add it to the monitor. It also provide information on writing resident libraries and runtime systems for custom applications.

The first chapter describes the structures used by the monitor that are resident on disk. These include the directory structure, disk allocation tables, save image library formats, bootstrap formats and bad block mapping.

The second chapter describes the tables used within the monitor to control system resources and provide program services. These tables provide job, memory, file and device control, as well as providing program services such as interjob communication.

The third chapter provides the information needed to write and install a custom device driver. It describes the entry points and information the driver must provide to the monitor as well as the services the monitor provides for the driver.

The fourth chapter provides information that enhances information already provided by Digital on writing custom resident libraries and runtime systems. It concentrates mainly on non-standard uses of resident libraries and runtime systems.

Appendix A provides several fold out sheets which give a quick reference to most of the information contained in this document. Appendix B provides examples of the peek sequences required to access most of the monitor tables. Appendices C and D provide examples of a user-written device driver and runtime system, respectively.

The reader should have a basic understanding of operating system theory and system programming and should be familiar with the MACRO-11 language. With the exception of offsets, all values used in tables and examples are in decimal, unless specified otherwise. All values used in MACRO program examples are in octal unless specified otherwise. All offset values are in octal.

The symbols "[" and "]" specify that the enclosed text is optional. The symbols "{" and "}" specify that one of the enclosed choices of text must be used.

INTRODUCTION

The RSTS community has repeatedly asked for a manual describing the internal workings of RSTS/E. In each instance, Digital's response has been that, if they provided information about internal structures they would no longer be able to make changes to these structures if they were needed for a new release of the monitor.

Their point is valid in many respects, but it still doesn't answer the need for this information by a user who recognizes that the information is subject to change and will not necessarily be upward compatible with new versions of RSTS/E.

The best solution to this delemma appears to be a manual produced by the user community. Michael Mayfield has taken the experience he has gained in over 10 years with RSTS/E and written a manual detailing the internals of the RSTS/E operating system.

This document describes, in detail, the internal workings and data structures of the RSTS/E monitor. It includes information on disk structures, monitor tables, device drivers, resident libraries and runtime systems.

Bear in mind, however, that information on the internal workings of any program under continual development is subject to change. As new versions of the monitor become available, information provided in this document may change. Design any programs you write using this information with this possibility in mind.

Under no circumstances should it be assumed that Digital Equipment Corporation has any responsibility to keep the internal workings of the RSTS/E operating system compatible between releases.

SUMMARY OF CHANGES

Several changes and additions were made to the RSTS/E structures in the V8.0 release. The following is a summary of the changes that effect this document:

1. Addition of RSTS Disk Structure Level 1 (RDS1) format directory structure.
2. Addition of UNTLVL table to record disk level for all mounted disks.
3. Addition of MFDPTR table to point to the beginning of the Master File Directory (MFD) for RDS1 format disks.
4. Replacement of bootstrap with new bootstrap format.
5. Changes to Secondary Job Descriptor Block (JDB2) required for removal from small buffer pool, EMT logging, extended spawn features and optional mini-systat.
6. Removal of small file system structures.
7. Changes to Unit Options table (UNTOPT) to support dual ported disks and disks that require that system I/O be stalled during dismount.
8. Addition of new send/receive object types.

In addition to the changes required for V8.0, this document has been redesigned to make it easier to use. All offsets are now in octal. All bit values now have a quick reference chart to make their use easier.

Changes from the previous release of this document are noted by a change bar in the outside margin. Deletions are noted by a bullet in the outside margin.

Chapter 1

DISK STRUCTURES

Disks are divided into individual files. Each of these files can be used as if it were a disk unto itself. Files are created, extended and deleted under the control of the RSTS/E monitor, specifically by the File Processor (FIP).

In order to catalog all the files on a disk, a directory structure is needed. RSTS V8.0 supports two different directory structurs; RDS0 and RDS1. The RDS0 directory structure is a combination of a master file directory and some number of user file directories. The RDS1 directory structure uses a combination of master file directory, (MFD), several group file directories, (GFDs), and several user file directories, (UFDs). Section 1.2 describes the format of the RDS0 directory structure. Section 1.3 describes the RDS1 directory structure.

Allocation and deallocation of disk space is handled automatically by FIP using the Storage Allocation Table (SAT) contained on each disk. The SAT contains a bit map that specifies which disk blocks are available for allocation and which are already in use. Section 1.4 describes the structure of the SAT.

The RSTS/E monitor consists of several independant program modules. These modules are contained in a specially indexed file structure known as a Save Image Library (SIL). When the system is started, the INIT program is brought into memory using a bootstrap contained in the first block on the system disk. The INIT program then loads the monitor from a SIL file and starts timesharing. Sections 1.5 and 1.6 describe the format of a SIL and the bootstrap block.

In order to provide for the proper handling of bad blocks on each disk, a special file is provided which maps these bad blocks and keeps them from being used in normal operations. Section 1.7 describes this bad block file.

1.1 TERMINOLOGY

A basic understanding of the terms used in disk structures is needed for the discussions which follow in this chapter.

1.1.1 Logical Block Number (LBN)

Each disk is divided into a number of 512 byte logical blocks. Logical blocks are numbered sequentially, starting at zero. The total number of logical blocks on a disk depends on the size of the disk. See the RSTS/E System Generation Manual for more information on disk sizes.

1.1.2 Cluster

A cluster is a group of contiguous logical blocks. The number of logical blocks in a cluster is always a power of two, in the range 1-256.

1.1.3 Device Cluster Size (DCS)

Large disks contain more blocks than can be specified in a 16 bit word. To circumvent this problem, RSTS/E divides large disks into device clusters. The size of the cluster is calculated such that all the clusters on the disk can be specified in a 16 bit word. See the RSTS/E System Generation Manual for a list of the device cluster sizes for all standard disk types.

1.1.4 Device Cluster Number (DCN)

Each device cluster on a disk is assigned a unique device cluster number. These numbers start at zero and continue sequentially through all device clusters.

1.1.5 FIP Block Number (FBN)

FIP and the common disk subsystem convert the 16 bit device cluster numbers to 23 bit FIP block numbers (FBNs), which are used to actually access the disk. FBN 1 corresponds to the logical block number of the first block in DCN 1. The following FBNs correspond to the remaining logical blocks numbers.

A special case is FBN 0. FBN 0 always corresponds to logical block 0. If the device cluster size of a disk is greater than one, the remaining blocks in device cluster 0 do not have a corresponding FBN and are, therefore, inaccessable.

1.1.6 Pack Cluster Size

When a disk pack in initialized, it is assigned a pack cluster size. This pack cluster size is used as the default cluster size for all files created on that pack. The pack cluster size is a power of two, in the range 1-16. The pack cluster size is always greater than or equal to the device cluster size.

1.1.7 File Cluster Size

The blocks of a file are grouped together in clusters. The directory structure contains the information necessary to access the first block of each cluster. Since the blocks of a cluster are contiguous, any block within the cluster can be accessed if the location of the first block is known. The file cluster size is always a power of two, in the range 1-256, but not less than the pack cluster size.

1.1.8 MFD Cluster Size

The Master File Directory (MFD) consists of up to seven clusters. The MFD cluster size can be 1,2,4,8, or 16 in RDS0 and in RDS1. Since the current version of the RDS1 MFD only uses three blocks, specifying a clustersize greater than four for an RDS1 MFD is of no value in most cases.

1.1.9 GFD Cluster Size

Each Group File Directory (GFD) consists of up to seven clusters. The clustersize of the GFD determines the number of user accounts that can be defined for a particular group. The GFD cluster size can be 4, 8 or 16.

1.1.10 UFD Cluster Size

Each User File Directory (UFD) consists of up to seven clusters. The cluster size of a UFD determines the number files and the amount of file retrieval information that can be contained in a particular account. The UFD cluster size is always a power of two, in the range 1-16, but not less than the pack cluster size.

1.1.11 FIP Block Sub-Block (FBB)

A FIP Block Sub-Block is used to identify a particular block on a disk. It is used in File Control Blocks (see section 2.4.2) and Window Control Blocks (see section 2.4.3) to access an entry in a directory.

A FIP Block Sub-Block consists of four bytes. The first byte is the FIP Unit Number of the specified disk. The next byte contains the most significant byte of the FIP block number. The next two bytes (as a word) contain the least significant bytes of the FIP block number.

1.1.12 Directory Link

The entries in a User File directory (UFD) are linked together in a linked list. The elements of this list are connected together by special directory links.

A directory link is a word that specifies the cluster number of the directory, the block within the cluster and the offset within the block to access a particular directory entry. See section 1.3.5 for a description of directory links.

1.2 RDS0 DISK DIRECTORY STRUCTURES

The RSTS/E Disk Structure Level 0 (RDS0) file system is based on a two level hierarchy: a Master File Directory to store account information and User File Directories to store file information. The Master File Directory (MFD) is used to retrieve information about each account on the system. Information about the files belonging to each account are stored in User File Directories (UFDs).

The Master File Directory (MFD) contains information about every account on the disk. Each disk contains exactly one MFD. Using the information in the MFD, the monitor can find the User File Directory (UFD) for each account. The MFD also contains information about each account's resource usage. It also contains information necessary to mount and use the disk pack. Since the MFD also doubles as the UFD for account [1,1], it may contain UFD entries if files are present in account [1,1].

Since all information about file and UFD placement is based on the directory structure, the root of this structure, the MFD, must be located at a known location on the disk. The MFD is the only portion of an RDS0 disk except for the bootstrap, that always resides at a fixed location on the disk. The first cluster of the MFD is located at disk cluster number (DCN) one. The remaining clusters are found using information in the MFD structure. They may be located anywhere on the disk.

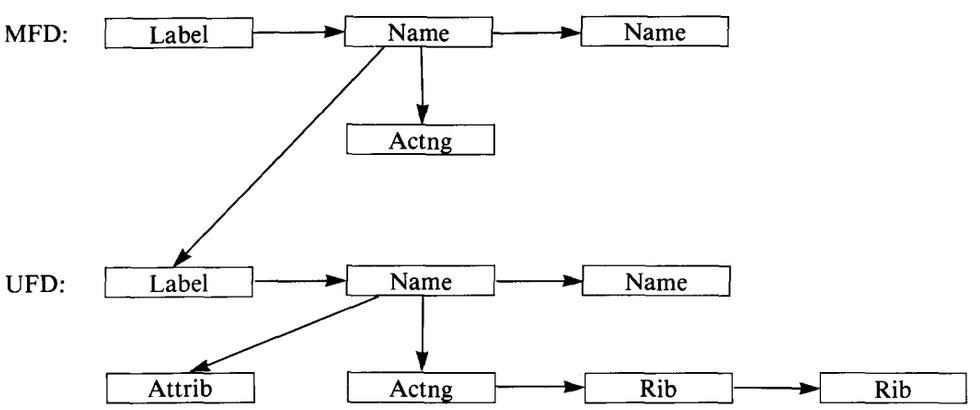
The MFD consists of up to six different types of entries: the Label Entry, Name Entries, Accounting Entries, Attributes Entries (optional), Cluster Maps and Unused Entries.

The User File Directory (UFD) catalogs a user's files. Each file in the associated account has an entry in the UFD which completely describes the file, including its protection requirements, current status and information needed to access the contents of the file. The format of the UFD is identical on RDS0 and RDS1 disks. See section 1.3.4 for more information on the UFD.

The UFD is almost identical in format to the RDS0 MFD. It contains a Label Entry, Name Entries, Accounting Entries, Attributes Entries (optional), Cluster Maps and Unused Entries. In addition, it contains Retrieval Entries which are used to determine the actual location on disk of each block of a data file.

The MFD Name Entry and Accounting Entry for each account are created when the account is created. No space is allocated for the UFD until the first file is created in the associated account.

The following figure provides a graphic representation of the RDS0 directory structure:



UFD information is kept in a linked list. In most cases, new files are added to the end of the list. However, it is possible to add a file to the beginning of the list using a special open mode or by specifying "new files first" when initializing the disk pack (see sections 1.2.1.1 and 1.3.1.1). Files are added to the beginning of the list on a logical basis only and are not necessarily physically first unless optimized by the REORDR utility program or a similar utlity.

Improper arrangement of files in the UFD can have a dramatic effect on the efficiency of disk I/O. If properly done, disk I/O requires no additional seeks for directory overhead. But, improperly done (alas, the obvious way), directory links can cross block boundaries dozens, if not hundreds, of times. This can require many overhead seeks for every disk access request in your program.

Optimal organization of the UFD requires that (1) the order of information in the UFD be optimized using REORDR or a similar utility, (2) files use optimal clustersizes, (3) contiguous files be used whenever possible.

1.2.1 MFD Label Entry

The MFD Label Entry contains the information needed to mount and allow access to the disk pack. It is always the first entry in MFD cluster 0. It is created by the DSKINT option of INIT or by the DSKINT utility program during timesharing.

Symbol	*Offset*		*Offset*	*Symbol*
		Link to first Name Entry in MFD	0	ULNK
		-1	2	
		0	4	
		0	6	
		Pack cluster size	10	
		Pack status	12	
		Pack ID (in RAD50)	14 16	

Offset	*Symbol*	*Description*
0	ULNK	This word contains the link to the first Name Entry in the MFD. This link will never be zero since there is always at least one Name Entry in every MFD, the entry for account [1,1].
2		This word contains a -1 to show that the entry is in use.
4		Unused
6		Unused
10		This word specifies the pack cluster size.
12		This word contains a set of bits that describe the characteristics of this disk pack as well as its current status (see section 1.2.1.1).
14		These two words contain the logical pack ID in RAD50. The pack ID is used when the disk is mounted and as an alternate name for the pack once it has been mounted.

1.2.1.1 MFD Pack Status

The pack status bits contained in offset 12 of the MFD Label Entry have the following meaning when set:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UC.MNT	UC.PRI			UC.DLW		UC.TOP									

Bit	*Symbol*	*Description*
<0:8>		Unused
<9>	UC.TOP	Link new files to the top of the UFD for files on this disk. This is the "New Files First" characteristic.
<10>		Unused.
<11>	UC.DLW	The access date for files on this disk should only be updated when they are written into, not when they are accessed for reading. This is the "Date Last Written" characteristic.
<12:13>		Unused.
<14>	UC.PRI	This disk is a private pack (as opposed to a disk in the public structure). System disks are defined as private packs. When used as the system disk they are treated as part of the public structure. When explicitly mounted, they are treated as private packs.
<15>	UC.MNT	This disk is currently mounted. If this bit is already set when an attempt is made to mount the disk, it was not dismounted properly after its last use and should be "rebuilt" before being used.

1.2.2 MFD Name Entry

The MFD Name Entry is used to catalog all the accounts on the system. Each account on the system disk has a Name Entry associated with it. Name Entries and their associated User File Directories are created automatically on other disks in the public structure as files are created on those disks. Name Entries are created on private packs only when the associated account is created using the REACT program or the UFD creation SYS call.

The Name Entry contains all the information necessary to identify a desired account and to verify login access to the account. Name Entries are linked together in the order in which they were created.

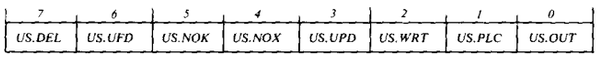
Symbol	*Offset*		*Offset*	*Symbol*
	Link to next Name Entry in MFD		0	ULNK
	PPN Project #	PPN Programmer #	2	UNAM
	Password (in RAD50)		4 6	
	Unused	Status	10	USTAT
	Access count		12	UACNT
	Link to accounting entry		14	UAA
	DCN of 1st UFD cluster		16	UAR

Offset	*Symbol*	*Description*
0	ULNK	This word contains a link to the next MFD Name Entry. If this is the last MFD Name Entry, this word will be zero.
2	UNAM	This word contains the project programmer number (PPN) of the account associated with this Name Entry. The programmer number is in the low byte. The project number is in the high byte. The PPN cannot be [*,255], [255,*] or [0,*], except for account [0,1].
4		These two words are the account password in RAD50. A password of ?????? specifies that the account exists but cannot be logged into. Password verification is the responsibility of the LOGIN program. It is not checked by the monitor.
10	USTAT	This byte contains a set of bits describing the status and restrictions of the UFD or file associated with this Name Entry (see section 1.2.2.1).
11		This byte is unused. It corresponds to the protection code in a UFD Name Entry. It always contains a value of 60 in case UFD protection is provided in a future release of RSTS/E.
12	UACNT	This word is used as a pair of bytes to count current accesses to the UFD. The low byte (offset 12) is currently unused. The high byte (offset 13) is the current login count for this account.

14	UAA	This word is the directory link to the Accounting Entry for this account.
16	UAR	This word is the device cluster number (DCN) of the first cluster of the UFD for this account. If no files have been created for this account on this disk since the account was created or zeroed, this word will be 0. Disk space for the UFD is not actually allocated until the first file is created.

1.2.2.1 USTAT - UFD Status

The bits in the account status byte, USTAT, in the MFD have the following meaning when set:



7	6	5	4	3	2	1	0
US.DEL	US.UFD	US.NOK	US.NOX	US.UPD	US.WRT	US.PLC	US.OUT

Bit	*Symbol*	*Description*
<0>	US.OUT	Always 0.
<1>	US.PLC	The UFD was placed at a specific location on the disk.
<2>	US.WRT	Write privileges for explicit opens of the UFD have already been given out.
<3>	US.UPD	Always 0.
<4>	US.NOX	Always 1.
<5>	US.NOK	Always 1.
<6>	US.UFD	This bit is always 1 to specify that this is an MFD Name Entry. This allows UFD Name Entries for [1,1] to be intermixed with MFD Name Entries since the MFD is also used as the [1,1] UFD.
<7>	US.DEL	Always 0.

1.2.3 MFD Accounting Entry

Every account has an MFD Accounting Entry. The Accounting Entry stores accumulated resource usage counts for the associated account. Each time a user logs out, his current resource usage (maintained by the monitor) is added to the existing values in the Accounting Entry for that user.

Symbol	*Offset*			*Offset*	*Symbol*
		Link to Attributes Entry		0	ULNK
		Accumulated CPU time (LSB)		2	MCPU
		Accumulated connect time		4	MCON
		Accumulated kilo-core-ticks		6	MKCT
		Accumulated device time		10	MDEV
	13	CPU time (MSB)	KCT (MSB)	12	MMSB
		Logout disk quota		14	MDPER
		UFD cluster size		16	UCLUS

Offset	*Symbol*	*Description*
0	ULNK	This word is available for a link to an Attributes Entry. While RSTS/E doesn't currently use attributes on an Accounting Entry, a user written program could add attributes to an Accounting Entry for its own use. (See section 1.3.4.4 for a discussion of the Attributes Entry.)
2	MCPU	This word contains the least significant 16 bits of the accumulated CPU time (in tenths of seconds) used by this account. The most significant 6 bits of CPU time are stored at offset MMSB. The resulting 22 bit number can hold a total of 116.5 hours of CPU time.

4	MCON	This word records the accumulated connect time (in minutes) used by this account. This word can record up to approximately 45.5 days of connect time.
6	MKCT	A kilo-core-tick (KCT) is a combined measurement of CPU and memory usage. KCTs are calculated at the end of each run burst by multiplying the CPU time used (in tenths of a second) times the size of the job (in K-words). MKCT contains the least significant 16 bits of the accumulated kilo-core-ticks for this account. The most significant 10 bits are stored at offset MMSB. The resulting 26 bit number can hold a total of 67,108,863, KCTs, or the equivalent of 116.5 CPU hours at 16K-words.
10	MDEV	Device usage time is recorded in device-minutes. A device-minute is equivalent to having one device assigned (either explicitly or implicitly) for one minute. Having two devices assigned for one minute is two device-minutes. MDEV contains the accumulated device usage time for this account, in device-minutes. A maximum of 45.5 device days may be recorded.
12	MMSB	This word contains the most significant 10 bits of the accumulated kilo-core-ticks in bits 0 through 9 (see MKCT) and the most significant 6 bits of the accumulated CPU time in bits 10 through 15 (see MCPU).
14	MDPER	This word contains the logout disk quota. It is specified when the account is created and changed with a SYS call. While the RSTS/E monitor does not enforce the logout quota itself, the LOGOUT program checks this value and will not allow a job to logout if the quota is exceeded.
16	UCLUS	This word specifies the UFD cluster size. Given this information, and the cluster map at the end of each block in the UFD, each block of the UFD can be located.

1.2.4 MFD Cluster Map

The MFD cluster map contains pointers (device cluster numbers) to each cluster in the MFD. There is a cluster map in every block of the MFD, at offset 760_8. Each cluster map in the MFD is identical. When a new cluster is allocated to extend the MFD, the cluster map in each block of the MFD is updated to show the change.

Symbol *Offset*		*Offset* *Symbol*
	MFD cluster size	0
	DCN of MFD cluster 0	2
	DCN of MFD cluster 1	4
	DCN of MFD cluster 2	6
	DCN of MFD cluster 3	10
	DCN of MFD cluster 4	12
	DCN of MFD cluster 5	14
	DCN of MFD cluster 6	16

Offset	*Symbol*	*Description*
0		This word contains the cluster size of the MFD.
2-16		The following seven words contain the device cluster number (DCN) for each corresponding cluster of the MFD. If less than seven clusters have been allocated to the MFD, the unused words will be 0.

1.2.5 User File Directory (UFD)

The structures in the User File Directory (UFD) are identical in RDS0 and RDS1. See section 1.3.4 for a description of the UFD.

1.3 RDS1 DISK DIRECTORY STRUCTURES

Version 8.0 of RSTS/E introduced a new directory structure which replaces the Master File Directory (MFD) structures used in previous releases. This new directory structure is identified as RSTS/E Disk Structure Level 1, or RDS1. The previous directory structure is identified as RDS0.

In RDS1 the old MFD structure has been replaced by a label block, a new format Master File Directory (MFD) and a Group File Directory (GFD). The MFD and GFD are used together to form a two level tree structure with the MFD as the root.

The label block contains information necessary to mount and use the disk pack. It also contains a pointer to the MFD. This allows the MFD to be placed close to the most used GFDs and files to minimize disk seek overhead.

The MFD and GFDs contain information about every account on the disk. Each disk contains one MFD for the entire disk, plus one GFD for every group number. Using information in the MFD and GFDs, the monitor can find the User File Directory (UFD) for each account. The GFD's also contain information about each account's resource usage.

The MFD contains a pointer to the Group Fil Director, GFD, for each group number used on the system. (The group number is the first number in an account number.)

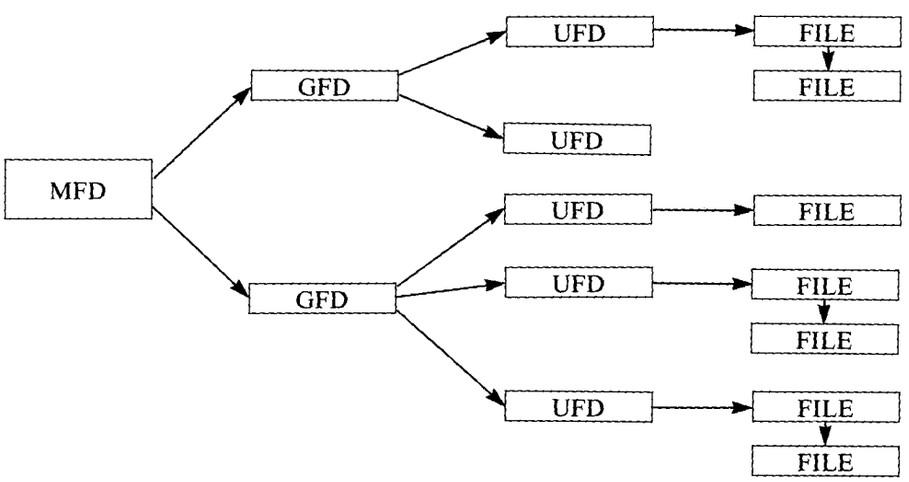
The GFD contains a pointer to the User File Directory (UFD) for each user number within the associated group. It also contains information required to log a user onto the system and to account for their resource usage.

The UFD contains information about the associated account's files. Each file in the associated account has an entry in the UFD which completely describes the file, including its protection requirements, current status and information needed to access the contents of the file.

The following procedure is used when opening a file in a specified account:

1. Read the MFD block to get a pointer to the GFD.
2. Read the GFD block to get a pointer to the UFD.
3. Read the first block of the UFD and follow the linked list of name entries until the specified file is found or the end of the directory is reached.

The following figure provides a graphic representation of the RDS1 directory structure:



Each GFD is created as the first account in the group is created. Name, accounting and attribute entries are created for each account as the account is created. No space is allocated for the UFD until the first file is created in the associated account.

UFD information is kept in a linked list. In most cases, new files are added to the end of the list. However, it is possible to add a file to the beginning of the list using a special open mode or by specifying "new files first" when initializing the disk pack (see section 1.3.1.1). Files are added to the beginning of the list on a logical basis only, and are not necessarily physically first unless optimized by the REORDR program or a similar utility.

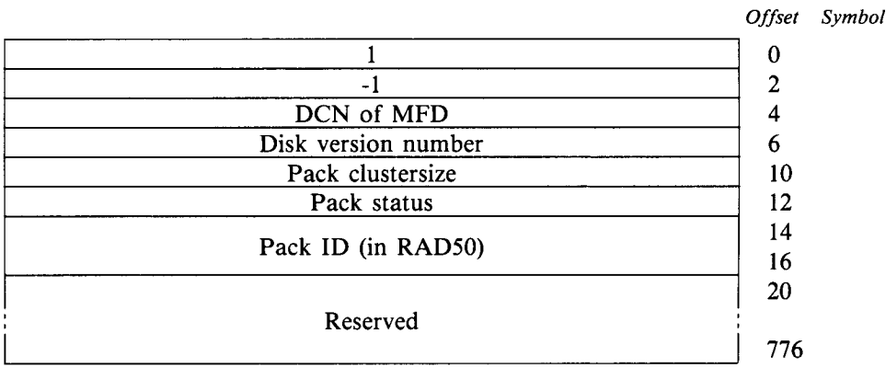
Improper arrangement of files in the UFD can have a dramatic effect on the efficiency of disk I/O. If properly done, disk I/O requires no additional seeks for directory overhead. But, improperly done (alas, the obvious way), directory links can cross block boundaries dozens, if not hundreds, of times. This can require many overhead seeks for every disk access request in your program.

Optimal organization of the UFD requires that (1) the order of information in the UFD be optimized using REORDR or a similar utility, (2) files use optimal clustersizes, (3) contiguous files be used whenever possible.

1.3.1 Pack Label

The pack label contains the information needed to mount and allow access to the disk pack. The pack label is the base of the entire directory system on the disk. It points to the Master File Directory (MFD) which, in turn, points to the Group File Directories (GFDs) which point to the User File Directories.

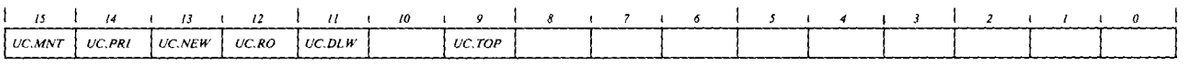
The pack label is always located at Disk Cluster Number (DCN) 1 on the disk. It is the only structure other than the bootstrap that must be located at a specific block number on an RDS1 disk.



Offset	*Symbol*	*Description*
0		This word always contains a 1.
2		This word always contains a -1.
4		This word is a pointer to the MFD. It specifies the Disk Cluster Number (DCN) of the first block of the MFD. This pointer is stored in MFDPTR (see section 2.5.6.6) when the disk is mounted.
6		This word identifies the RDS version and edit level. The low order byte (offset 6) is always a 1 to specify RDS1 format. The high order byte (offset 7) is currently a 1 to specify edit level one.
10		This word specifies the pack cluster size.
12		This word contains a set of bits that describe the characteristics of this disk pack, as well as its current status (see section 1.3.1.1)
14	PCKID	These two words contain the logical pack ID in RAD50. The pack ID is used when the disk is mounted and as an alternate name for the pack once its has been mounted.
20-776		These words are reserved for future pack attributes. They are currently unused and will contain zeroes.

1.3.1.1 Pack Status

The pack status bits contained in offset 12 of the pack label have the following meaning when set:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UC.MNT	UC.PRI	UC.NEW	UC.RO	UC.DLW		UC.TOP									

BIT	*Symbol*	*Description*
<0:8>		Unused
<9>	UC.TOP	Link new files to the top of the UFD for files on this disk. This is the "New Files First" characteristic.
<10>		Unused.
<11>	UC.DLW	The access date for files on this disk should only be updated when they are written into, not when they are accessed for reading. This is the "Date Last Written" characteristic.
<12>	UC.RO	This pack is inherently read-only.
<13>	UC.NEW	This is an RDS1 format pack.
<14>	UC.PRI	This disk is a private pack (as opposed to a disk in the public structure). System disks are defined as private packs. When used as the system disk they are treated as part of the public structure. When explicitly mounted, they are treated as private packs.
<15>	UC.MNT	This disk is currently mounted. If this bit is already set when an attempt is made to mount the disk, it was not dismounted properly after its last use and should be "rebuilt" before being used.

1.3.2 Master File Directory (MFD)

The Master File Directory (MFD) is the root of the directory structure used for accessing files. Its main purpose is to point to the Group File Directory (GFD) (see section 1.3.3) of the account group. In addition to these pointers, the MFD contains a label block, and blocks that may be used for group or pack attributes in future releases, but that are currently unused.

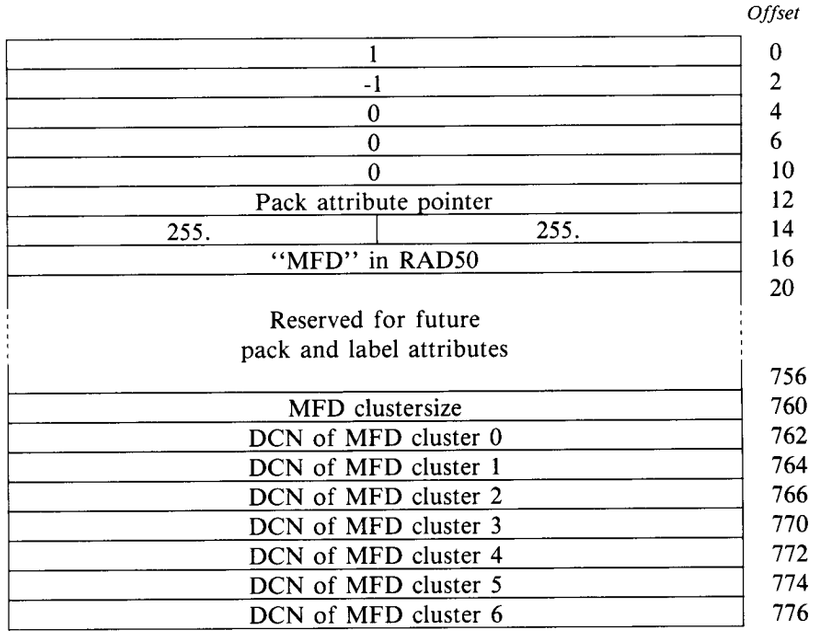
The blocks within the MFD have the following usages (see section 1.3.2.1 through 1.3.2.3):

Block	*Usage*
0	Label block.
1	GFD pointer block.
2	Group attribute pointer block.
3-n	Reserved for future attribute entries.

1.3.2.1 Label Block

The first block of the MFD (block 0) is a label block. Its purpose is to identify the following group of blocks as an MFD and to provide information that relates to the MFD as a whole.

This block is typically only used when the pack is mounted. All other accesses to the MFD are made to block 1 (the second block).

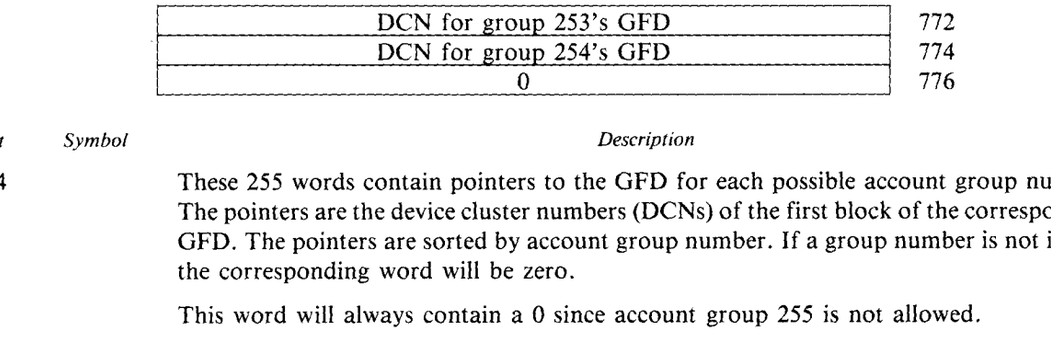


Offset	*Symbol*	*Description*
0		This word always contains a 1.
2		This word always contains a -1.
4		Unused.
6		Unused.
10		Unused.
12		This word is reserved for possible pack attribute pointer in a future release of RSTS. It is currently unused.
14		This word is used to signify that this is an MFD. It contains a pair of bytes with a value of 255 each, which signifies an invalid account number used to mark an MFD.
16		This word contains the characters "MFD" in RAD50.
20-756		Reserved for future pack attribute or label attributes. Currently unused.
760		This word specifies the MFD clustersize. The most significant bit of this word is always set. This signifies that this is an RDS1 format cluster map.
762-776		These seven words specify the device cluster number (DCN) for each corresponding cluster of the MFD. If less than seven clusters have been allocated to the MFD, the unused words will be 0.

1.3.2.2 GFD Pointer Block

The second block of the MFD (block 1) contains a table of pointers to Group File Directories (GFDs) corresponding to each possible account group number. The account group number is the first number in the PPN. For example, the group number for account [2,10] is 2.

The pointers in this block are used as a first level index to access the GFD entry for each account. By accessing this pointer block and then accessing the GFD pointed by this block, the User File Directory (UFD) for the desired account can be found in two disk accesses. If files are frequently opened using a specified account number, the GFD pointer block will typically remain in the directory cache at all times, reducing the number of physical disk accesses to a maximum of one.



Offset	*Symbol*	*Description*
0-774		These 255 words contain pointers to the GFD for each possible account group number. The pointers are the device cluster numbers (DCNs) of the first block of the corresponding GFD. The pointers are sorted by account group number. If a group number is not in use, the corresponding word will be zero.
776		This word will always contain a 0 since account group 255 is not allowed.

1.3.2.3 Other Blocks

The third block of the MFD (block 2) is reserved for future group attribute pointers. This block is currently unused.

The fourth and following blocks of the MFD are reserved for future group or pack attribute entries. These blocks are currently unused.

1.3.3 Group File Directory (GFD)

The Group File Directory (GFD) is the second level of the directory structure used for accessing files. Its main purpose is to point to the User File Directory (see section 1.3.4) of the account group. In addition to these pointers, the GFD contains a label block, a name entry pointer block and blocks that are used for UFD name, accounting, and attribute entries.

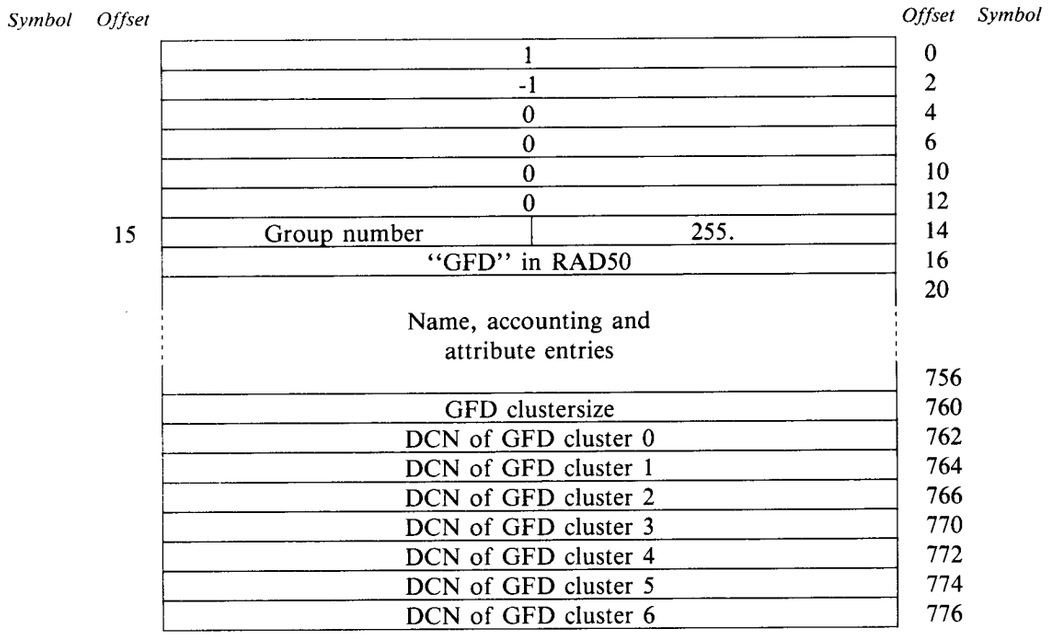
The blocks of the GFD have the following usage:

Block	*Usage*
0	Label block UFD name, accounting and attribute entries.
1	UFD pointer block.
2	Name entry pointer block.
3-n	UFD name, accounting and attribute entries.

1.3.3.1 Label block

The first block of the GFD is a label block. Its purpose is to identify the following group of blocks as a specific GFD and to provide information that relates to the GFD as a whole.

This block is typically only accessed for name, accounting and attribute entries. The first 16_8 bytes are not typically used by the monitor. They are present for compatability with MFD and UFD label formats and to allow GFDs to be identified in case of corruption of directory data.

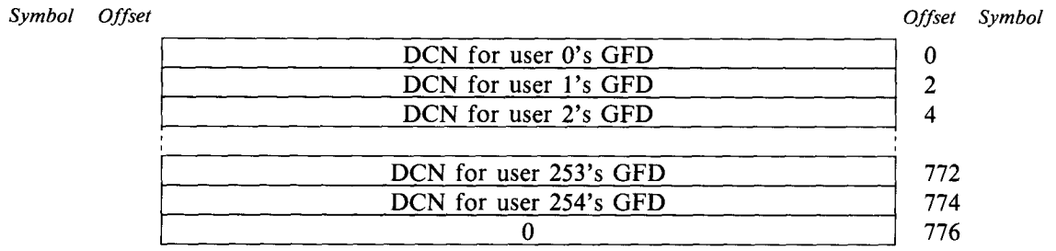


Offset	*Symbol*	*Description*
0		This word always contains a 1.
2		This word always contains a -1.
4		Unused.
6		Unused.
10		Unused.
12		Unused.
14		This byte is used to signify that this is a GFD. It contains a byte value of 255, which signifies an invalid account number used to mark a GFD.
15		This byte specifies the group number for this GFD.
16		This word contains the characters "GFD" in RAD50. GFDs that have been lost through corruption of the MFD can be found by searching for this value at offset 16_8 in each free cluster on the disk.
20-756		These 236 words are used for name, accounting and attribute entries.
760		This word specifies the GFD clustersize. The most significant bit of this word is always set. This signifies that this is an RDS1 format cluster map.
762-776		These seven words specify the device cluster number (DCN) for each corresponding cluster of the GFD. If less than seven clusters have been allocated to the GFD, the unused words will be 0.

1.3.3.2 UFD Pointer Block

The second block of the GFD contains a table of pointers to User File Directories (UFDs) corresponding to each possible account user number within this account group.The account user number is the second number in the PPN. For example, the user number for account [2,10] is 10.

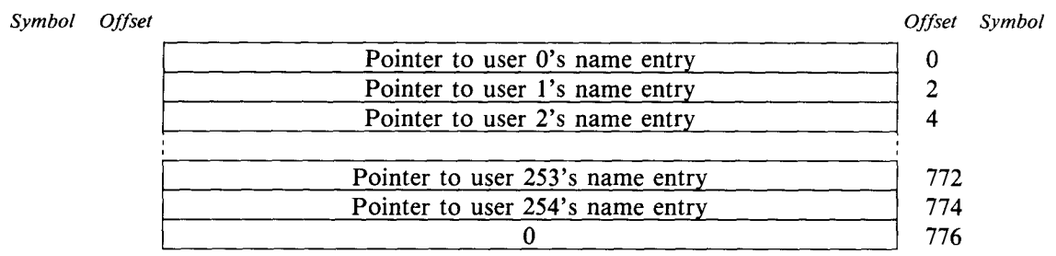
The pointers in this block are used as a second level index to access the UFD entry that contains file information for the desired account. By accessing the MFD pointer block and then accessing this pointer block in the GFD, the User File Directory (UFD) for the desired account can be found in two disk accesses.



Offset	*Symbol*	*Description*
0-774		These 255 words contain pointers to the UFD for each possible account user number. The pointers are the device cluster numbers (DCNs) of the first block of the corresponding UFD. The pointers are sorted by account user number. If a user number is not in use, the corresponding word will be zero.
776		This word will always contain a 0 since account user number 255 is not allowed.

1.3.3.3 Name Entry Pointer Block

The third block of the GFD contains a table of pointers to name entries corresponding to each possible account user number. The pointers in this block are used to access the GFD name entry (see section 1.3.3.4.1) during login and logout operations. The GFD name entry contains pointers to the GFD accounting entry and a list of attribute entries.



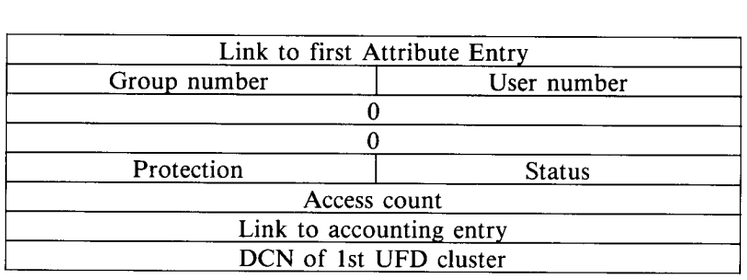
Offset	*Symbol*	*Description*
0-774		These 255 words contain directory links to the name entry for each possible account user number in this account group. The pointers are sorted by account user number. If a user number is not in use, the corresponding word will be zero.
776		This word will always contain a 0.

1.3.3.4 Other Blocks

The fourth and following blocks of the GFD contain name, accounting and attribute entries. Sections 1.3.3.4.1 through 1.3.3.4.3 describe these entries.

1.3.3.4.1 Name Entry

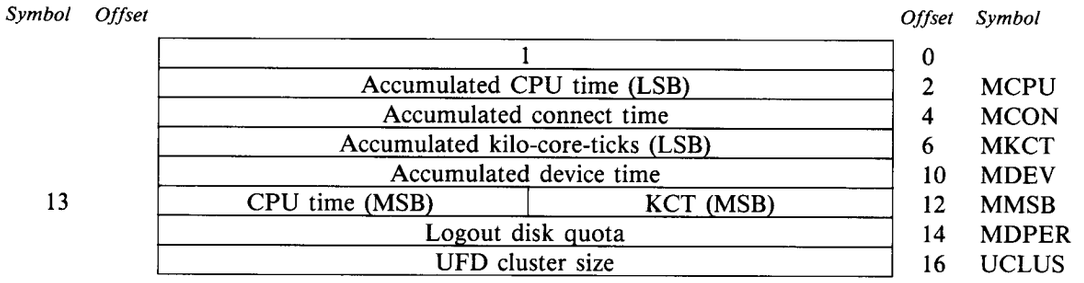
The Name Entry contains all the information necessary to identify a desired account. All other information about the account is accessed using pointers in the Name Entry.



Offset	*Symbol*	*Description*
0		This word contains a link to the first Attribute Entry. If there are no attribute entries, this link will be null.
2		This word contains the project programmer number (PPN) of the account associated with this Name Entry. The user number is in the low byte. The group number is in the high byte. The PPN cannot be [*,255], [255,*] or [0,*], except for account [0,1].
4		Reserved.
10		This byte contains a set of bits describing the status and restrictions of the UFD or file associated with this Name Entry (see section 1.2.2.1).
11		This byte is unused. It may be used in a future release of RSTS/E for the UFD protection code. It currently contains a value of 60.
12		This word is used as a pair of bytes to count current accesses to the UFD. The low byte (offset 12) is currently unused. The high byte (offset 13) is the current login count for this account.
14		This word is the directory link to the Accounting Entry for this account.
16		This word is the device cluster number (DCN) of the first cluster of the UFD for this account. If no files have been created for this account on this disk since the account was created or zeroed, this word will be 0. Disk space for the UFD is not actually allocated until the first file is created.

1.3.3.4.2 Accounting Entry

Every account has an Accounting Entry. The Accounting Entry stores accumulated resource usage counts for the associated account. Each time a user logs out, his current resource usage (maintained by the monitor) is added to the existing values in the Accounting Entry for that user.

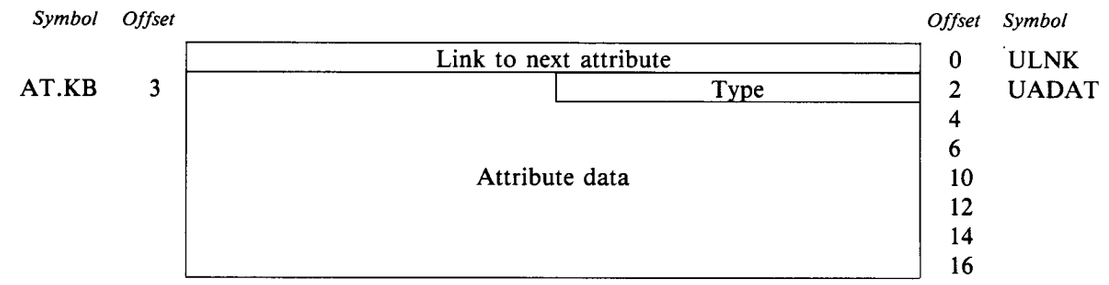


Offset	*Symbol*	*Description*
0		This word is always a 1.
2	MCPU	This word contains the least significant 16 bits of the accumulated CPU time (in tenths of seconds) used by this account. The most significant 6 bits of CPU time are stored at offset MMSB. The resulting 22 bit number can hold a total of 116.5 hours of CPU time.
4	MCON	This word records the accumulated connect time (in minutes) used by this account. This word can record up to approximately 45.5 days of connect time.
6	MKCT	A kilo-core-tick (KCT) is a combined measurement of CPU and memory usage. KCTs are calculated at the end of each run burst by multiplying the CPU time used (in tenths of a second) times the size of the job (in K-words). MKCT contains the least significant 16 bits of the accumulated kilo-core-ticks for this account. The most significant 10 bits are stored at offset MMSB. The resulting 26 bit number can hold a total of 67,108,863, KCTs, or the equivalent of 116.5 CPU hours at 16K-words.
10	MDEV	Device usage time is recorded in device-minutes. A device-minute is equivalent to having one device assigned (either explicitly or implicitly) for one minute. Having two devices assigned for one minute is two device-minutes. MDEV contains the accumulated device usage time for this account, in device-minutes. A maximum of 45.5 device days may be recorded.
12	MMSB	This word contains the most significant 10 bits of the accumulated kilo-core-ticks in bits 0 through 9 (see MKCT) and the most significant 6 bits of the accumulated CPU time in bits 10 through 15 (see MCPU).
14	MDPER	This word contains the logout disk quota. It is specified when the account is created and changed with a SYS call. While the RSTS/E monitor does not enforce the logout quota itself, the LOGOUT program checks this value and will not allow a job to logout if the quota is exceeded.
16	UCLUS	This word specifies the UFD cluster size. Given this information, and the cluster map at the end of each block in the UFD, each block of the UFD can be located.

1.3.3.4.3 Attribute Entries

Attribute entries are used for information about an account other than that provided by the name and accounting entries. It currently includes information about login passwords, and the date and time of the last login, the last password change and the account creation.

The attribute entries have the following general format:

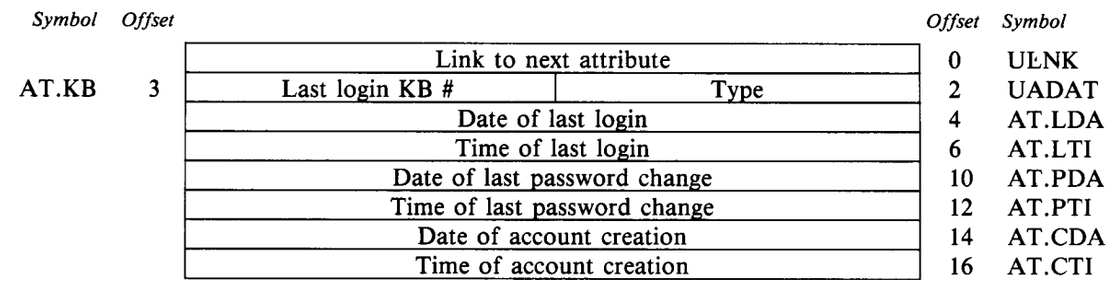


Offset	*Symbol*	*Description*
0	ULNK	This word contains a link to the next attribute entry for this account. If there is no following attribute entry, this word will have a null link. (See section 1.3.5 for information on directory links.)
2	UADAT	This byte specifies the attribute type, as follows:

Type	*Symbol*	*Description*
1	AA.QUO	Quota (currently unimplemented).
2	AA.PRV	Privileges (currently unimplemented).
3	AA.PAS	Password.
4	AA.DAT	Creation/Access date and times.

3-16	AT.KB	The following 13 bytes are used to store attribute data. The type of data stored depends on the attribute type. For a password attribute, the data is the password in ASCIZ. For a date/time attribute, the data has the format shown below.

The date/time attribute has the following format:



Offset	*Symbol*	*Description*
0	ULNK	This word contains a link to the next attribute entry for this account. If there is no following attribute entry, this word will have a null link. (See section 1.3.5 for information on directory links.)
2	UADAT	This byte contains a 4 to specify an attribute type of date/time.
3	AT.KB	This byte specifies the keyboard number of the last keyboard to login to this account. If the login request was made by a detached job, this byte will contain a -1.
4	AT.LDA	This word specifies the date that someone last logged into this account. It is originally set to 0.
6	AT.LTI	This word specifies the time that someone last logged into this account. It is originally set to 0.
10	AT.PDA	This word specifies the date that the password for this account was last changed.
12	AT.PTI	This word specifies the time that the password for this account was last changed.
14	AT.CDA	This word specifies the date that this account was created.
16	AT.CTI	This word specifies the time that this account was created.

1.3.4 User File Directory (UFD)

The User File Directory (UFD) catalogs a user's files. Each file in the associated account has an entry in the UFD which completely describes the file, including its protection requirements, current status and information needed to access the contents of the file. The format of the UFD is identical in RDS0 and RDS1 disks.

The UFD is almost identical in format to the RDS0 MFD. It contains a Label Entry, Name Entries, Accounting Entries, Attributes Entries (optional), Cluster Maps and Unused Entries. In addition, it contains Retrieval Entries which are used to determine the actual location on disk of each block of a data file.

1.3.4.1 UFD Label Entry

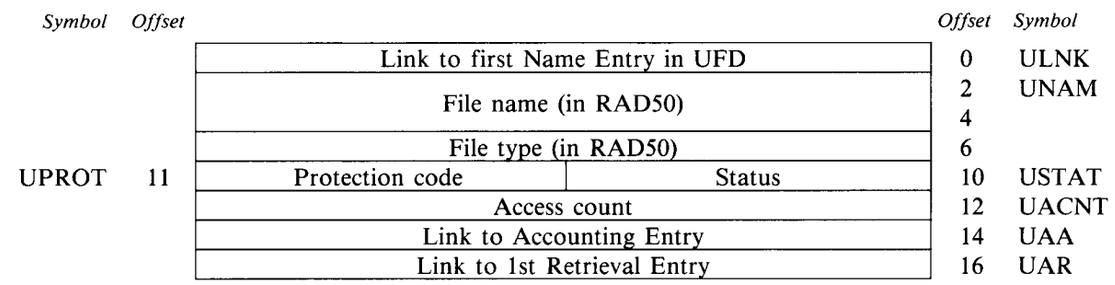
The UFD Label Entry is the root of the UFD structure. The list of Name Entries starts at this Label Entry. The Label Entry is created when the UFD is created (ie. when the first file is created in this account). There is one UFD Label Entry per UFD. It is always the first entry in the first cluster of the UFD.

Symbol	*Offset*			*Offset*	*Symbol*
		Link to first Name Entry in UFD		0	ULNK
		-1		2	
		0		4	
		0		6	
		0		10	
		0		12	
	15	PPN project #	PPN programmer #	14	
		"UFD" in RAD50		16	

Offset	*Symbol*	*Description*
0	ULNK	This word contains the link to the Name Entry of the first file in the UFD. If no files exist in the UFD, this link will be null.
2		This word is always -1 to mark this entry in use.
4-12		Unused.
14		This word contains the account number for this UFD. The programmer number is in the low byte. The project number is in the high byte.
16		This word contains the letters "UFD" in RAD50. UFDs that have been lost through corruption of the MFD can be found by searching for this value at offset 16 in each free cluster on the disk.

1.3.4.2 UFD Name Entry

There is a Name Entry in the UFD for every file in an account. The Name Entry contains all the information necessary to identify the file it belongs to. When opening a file, the monitor follows the linked list of Name Entries searching for the desired file name. If found, the file's current status and protection code are checked and the status is changed as necessary.



Offset	*Symbol*	*Description*
0	ULNK	UFD Entries are chained together in a linked list. This word contains a link to the next Name Entry in the UFD. If there is no following Name Entry, this word will have a null link. New files are normally added after the last Name Entry in the list. If "New Files First" is used, the new Name Entry is linked between the UFD Label Entry and the first UFD Name Entry.
2	UNAM	These three words contain the file name and type (called "file extension" in earlier versions of RSTS) in RAD50.
10	USTAT	This byte contains a set of bits describing the status and restrictions for the file associated with this Name Entry (see section 3.4.2.1).
11	UPROT	This byte specifies the file's protection code.
12	UACNT	This word is the current runtime system and resident library access count for this file. It is incremented each time a runtime system or resident library is added and decremented each time a runtime system or resident library is removed. All other access counts are kept in memory in File Control Blocks (FCBs) (see section 2.4.1).
14	UAA	This word is a link to the Accounting Entry for this file.
16	UAR	This word is a link to the first Retrieval Entry for this file. If there are no Retrieval Entries for this file (ie. the file is zero length), this link will be null.

1.3.4.2.1 USTAT - File Status

The bits in the file status byte, USTAT, in the UFD have the following meaning when set:

15	14	13	12	11	10	9	8
US.DEL	US.UFD	US.NOK	US.NOX	US.UPD	US.WRT	US.PLC	US.OUT

Bit	*Symbol*	*Description*
<0>	US.OUT	This bit is obsolete. If it is set, a protection violation will be generated if the file is opened.
<1>	US.PLC	The file was placed at a specific location on the disk.
<2>	US.WRT	Write privileges have been given out on a previous open.
<3>	US.UPD	The file is open for update.
<4>	US.NOX	The file is contiguous and may not be extended.
<5>	US.NOK	The file is non-deletable. This bit is normally set and cleared only by the REFRESH option in INIT.
<6>	US.UFD	This bit is always 0 to specify that this is a UFD Name Entry. This allows UFD Name Entries for account [1,1] to be intermixed with MFD Name Entries since the MFD is also the [1,1] UFD in RDS0 format disks.
<7>	US.DEL	The file should be deleted when the access count becomes 0.

1.3.4.3 UFD Accounting Entry

Every UFD Name Entry has an Accounting Entry associated with it. The Accounting Entry contains information about the file's creation and access date, its current length and static information, such as runtime system name and cluster size.

Symbol *Offset*		*Offset*	*Symbol*
Link to Attributes Entry		0	ULNK
Last access date		2	UDLA
Number of blocks in the file		4	USIZ
Creation date		6	UDC
Creation time		10	UTC
Runtime system name (in RAD50)		12 14	URTS
File cluster size		16	UCLUS

Offset	*Symbol*	*Description*
0	ULNK	This word contains a link to the first Attributes Entry. If there are no Attribute Entries, this word will contain a null link.
2	UDLA	This word contains the date of last access or the date of last modification (if the MFD Label Entry specifies "Date Last Written") in RSTS/E internal format. (See section 2.8 for a description of the internal date format.)
4	USIZ	This word is set to 1 when the file is created and adjusted by the file cluster size each time a file cluster is allocated. The actual number of blocks in the file is not updated until the file is closed, unless the file is opened using mode 8.
6	UDC	This word is the creation date in RSTS/E internal format. (See section 2.8 for a description of the internal time format.)

10	UTC	This word is the creation time in RSTS/E internal format. (See section 2.8 for a description of the internal time format.)
12	URTS	These two words contain the runtime system name, in RAD50. If a RUN request is issued for this file, the file will be opened on channel 15 and control will be passed to the runtime system specified in these two words. The "compiled" bit in the protection code and the first word of URTS are used to specify that the file may contain more than 65,535 blocks. If the compiled bit is not set and the first word or URTS contains a zero, the second word of URTS contains the most significant bits of the file size. These bits are combined with the value at offset USIZ to create the actual file size.
16	UCLUS	This word contains the file's cluster size.

1.3.4.4 UFD Attributes Entry

RMS-11 file structures use Attribute Entries to store record and file information. User programs can also use attributes on a file to record their own file specific information. Up to ten words of attribute values may be specified. Each Attribute Entry holds up to seven words. If more than seven values are required, an additional Attribute Entry is linked to the first one.

Symbol	*Offset*		*Offset*	*Symbol*
		Link to next Attribute Entry	0	ULNK
		Value 1	2	
		Value 2	4	
		Value 3	6	
		Value 4	10	
		Value 5	12	
		Value 6	14	
		Value 7	16	

Offset	*Symbol*	*Description*
0	ULNK	This word contains a link to the next Attribute Entry for this file. If there is no additional Attribute Entry, this link will be null.
2-14		These seven words contain the attribute values. These values are set and read by RMS and by user programs. File attributes are for use by RMS and user programs and are not used by the monitor. Only the first three words in the second Attribute Entry may be used to store values. See section 1.3.4.4.1 for a description of the RMS-11 file attribute values.

1.3.4.4.1 RMS-11 Attribute Values

RMS-11 uses file attributes to record extended information about an RMS file. This information includes file organization, record type, record size and end of file information. (See the *RMS-11 Users Guide* for more information.) The contents of each word of the Attribute Entries for an RMS-11 file are as follows:

Word	*Bits*	*Description*
1	<0:3>	Record format: 0 = Undefined 1 = Fixed length records 2 = Variable length records 3 = VFC (Variable with fixed control) 4 = Stream ASCII

	<4:7>	File organization: 0 = Sequential 1 = Relative 2 = Indexed
	<8:11>	Print control: 1 = FORTRAN 2 = Carriage return 4 = VFC records contain print control 8 = Does not span blocks
	<12:15>	Unused.
2		Record size (actual size for fixed length records or maximum size for variable length records).
3		Highest virtual block number (most significant bits).
4		Highest virtual block number (least significant bits).
5		End of file block number (most significant bits).
6		End of file block number (least significant bits).
7		Offset to first unused byte in the last block of the file.
8	<0:7>	This byte contains the bucket size for indexed files.
	<8:15>	This byte contains the number of bytes in the fixed control area.
9		Maximum length of record actually read by RMS.
10		Default extension quantity.

1.3.4.5 UFD Retrieval Entry

Every file in the UFD that contains at least one block has some number of Retrieval Entries associated with it. In most cases there are Retrieval Entries for a single file, each linked together in a list. These Retrieval Entries provide all the information necessary to translate the logical block number of a file into a physical block number on the disk.

For each cluster of a file there is a corresponding entry in one of the file's Retrieval Entries. When a particular block in a file is desired, its physical location is computed by finding the Retrieval Entry corresponding to the file cluster which contains the desired block. This entry provides the disk cluster number corresponding to the beginning of the file cluster that contains the desired block. Since clusters are contiguous, the location of the desired block can be determined as an offset from the beginning of the cluster.

When a file is opened, the seven cluster pointers contained in the file's first Retrieval Entry are copied into the corresponding window descriptor block of the Window Control Block (see section 2.4.2.). These seven pointers in memory are known as a "window".

When a request is made to access a file, the window is checked to see if any of its cluster pointers correspond to the desired block. If so, the block is accessed using tghe information contained in the window. No additional is required.

If the desired block is not mapped by the window, the linked list of Window Control Blocks is searched to see if any other window contains the desired information. If none of the windows in memory contain the information needed, the linked list of Retrieval Entries in the directory is followed until the desired Retrieval Entry is found. The pointers from this Retrieval Entry are copied into the window and used for accessing the file. This procedure is known as a "window turn".

As you can see from the description above, the cluster size of a file has a tremendous impact on the overhead required to access a file. If a clustersize can be chosen such that the entire file will fit within one window (ie. seven clusters), no window turns will be required, and no additional overhead will be incurred.

For this reason, file clustersize should be specified when the file is created whenever possible. This applies even to contiguous files since contiguous files contain all the Retrieval Entries of a normal file and because a contiguous file can be transformed into a non-contiguous file by extending it.

Symbol	*Offset*		*Offset*	*Symbol*
		Link to next Retrieval Entry	0	ULNK
		DCN of cluster n + 0	2	
		DCN of cluster n + 1	4	
		DCN of cluster n + 2	6	
		DCN of cluster n + 3	10	
		DCN of cluster n + 4	12	
		DCN of cluster n + 5	14	
		DCN of cluster n + 6	16	

Offset	*Symbol*	*Description*
0	ULNK	This word is the link to the following Retrieval Entries for this file. If this is the last Retrieval Entry in the list, the link will be null.
2-16		These six words contain the device cluster numbers (DCNs) that correspond to each file cluster. With this information, each block in a cluster can be accessed by an appropriate offset from the beginning of the cluster. Unused words in the last Retrieval Entry will be 0.

1.3.4.6 UFD Cluster Map

The UFD cluster map contains pointers (device cluster numbers) to each cluster in the UFD. There is a cluster map in every block of the UFD, starting at offset 760. Each cluster map in the UFD is identical. When a new cluster is allocated to extend the UFD, the cluster map in each block of the UFD is updated to show the change.

Symbol	*Offset*		*Offset*	*Symbol*
		UFD cluster size	0	
		DCN of UFD cluster 0	2	
		DCN of UFD cluster 1	4	
		DCN of UFD cluster 2	6	
		DCN of UFD cluster 3	10	
		DCN of UFD cluster 4	12	
		DCN of UFD cluster 5	14	
		DCN of UFD cluster 6	16	

Offset	*Symbol*	*Description*
0		This word contains the cluster size for the UFD.
2-16		The following seven words contain the device cluster number (DCN) for each corresponding cluster of the UFD. If less than seven clusters have been allocated to the UFD, the unused words will be 0.

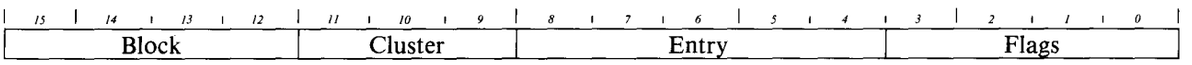
1.3.4.7 Unused Entries

Each block of the MFD and UFD can hold 32 entries. If the first two words of an entry are zero, the entry is considered to be unused and available for allocation when needed. The contents of an Unused Entry (other than the first two words) are unused by the monitor.

Symbol	*Offset*		*Offset*	*Symbol*
	0		0	
	0		2	
	Unused		4	
	Unused		6	
	Unused		10	
	Unused		12	
	Unused		14	
	Unused		16	

1.3.5 Directory Links

Directory links are internal pointers between directory entries. Given a link and a cluster map, any directory entry can be accessed in, at most, one disk access. The link is a packed word with the following format:



Use	*Bits*	*Description*
Block	<15:12>	These four bits select the block within the directory cluster (0 through the UFD cluster size).
Cluster	<11:9>	These three bits select the cluster within the cluster map (0 through 6).
Entry	<8:4>	These five bits select the entry within the directory block (0 through 31). Note that by clearing all the other fields within the link, the link can be used as a byte offset into the directory block.
Flags	<3:0>	These four bits are used as flags that relate to the current entry or to a block pointed to by the current entry. They have the following meaning when set:

<0> This bit denotes that the link word is in use. It is required in MFD and UFD Accounting Entries and Attributes Entries where the link field may be zero (null).

<1> When used in an MFD Accounting Entry, the associated UFD contains a bad block. When used in a UFD Accounting Entry, the associated file contains a bad block. When used in a Retrieval Entry, one or more of the associated clusters contains a bad block.

<2> When used in a Name Entry, the file is marked for data cacheing. When used in an Accounting Entry, the cacheing is specified as sequential.

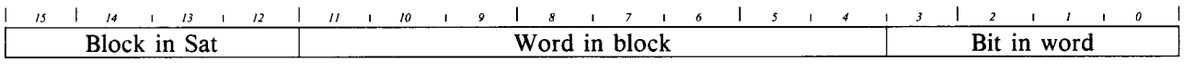
<3> This bit is set during a clean to mark the entry as being in use. It should not be set during normal operation.

1.4 STORAGE ALLOCATION TABLE (SAT)

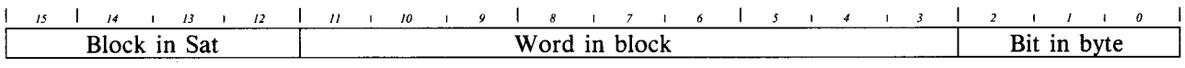
The Storage Allocation Table (SAT) is used to control allocation of space on each disk. The SAT is a bit map. Each bit corresponds to a pack cluster, beginning with cluster 0 and continuing for all pack clusters on the disk. The appropriate bit is set when the pack cluster is allocated and cleared when it is deallocated.

The SAT is originally created by the DSKINT option of INIT or the DSKINT utility program. It resides in [0,1] as the file SATT.SYS. When originally created it shows all blocks as being deallocated except for those used by the MFD, the UFD for [0,1], bad blocks found by DSKINT and the blocks used by the SATT.SYS file.

The pack cluster number can be used to access the SAT directly using the following indexing:



or:

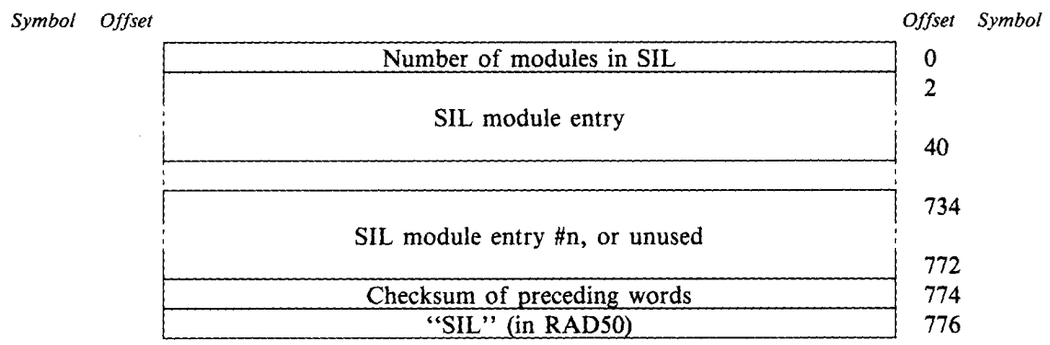


1.5 SAVE IMAGE LIBRARY (SIL)

Save image libraries (SILs) are used to combine the separate modules that make up the monitor into a single file. They are also used to contain runtime systems, resident libraries, the INIT monitor program and SAV images loadable by INIT. The index block contained within the SIL provides all the information needed to load the different portions of the monitor or other programs into memory and handle overlays when needed.

1.5.1 SIL Index Block

The first block of the SIL is an index block. The index block contains entries for up to 15 modules. See section 1.5.2 for a description of a module entry.



Offset	*Symbol*	*Description*
0		This word specifies the number of modules in the SIL (1 through 15).
2		These sixteen words are the module entry descriptor for the first module in the SIL (see section 1.5.2). If there is more than one module in this SIL, this module entry is followed by the other entries. If the SIL contains less than fifteen modules, the space between the last module entry and offset 774 is unused.
774		This word contains a checksum of all the preceding words. The checksum is computed by starting with a value of zero and XORing each word from offset 0 to offset 772 with it.
776		This word contains the letters "SIL" in RAD50.

1.5.2 SIL Module Entry

Each module within the SIL has a module entry associated with it in the SIL index. Each module entry contains information about the module, including load and transfer addresses, size, overlay descriptors and a pointer to the symbol table for the module.

Symbol	*Offset*		*Offset*	*Symbol*
		Module name (in RAD50)	0 2	SE.NAM
		Module .IDENT (in RAD50)	4 6	SE.IDN
		Block offset to module	10	SE.BLK
		Block offset to symbol table	12	SE.STB
		Number of symbols in symbol table	14	SE.STN
		Virtual load address	16	SE.LOD
		Virtual size of module	20	SE.SIZ
		Virtual transfer address	22	SE.XFR
		Size of module (in blocks)	24	SE.SZD
		Block offset to overlay descriptors	26	SE.OVB
		Number of overlay descriptors	30	SE.OVN
		Offset to start of module	32	SE.OFF
		0	34	
		RT-11 emulator parameter	36	SE.XXX

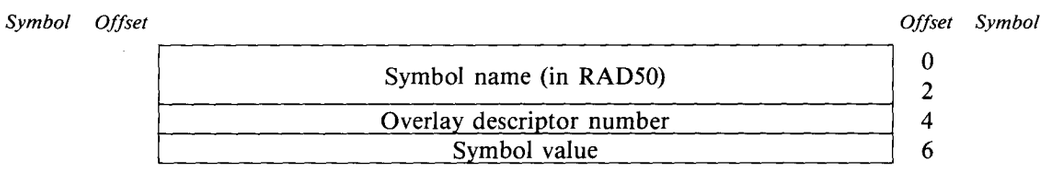
Offset	*Symbol*	*Description*
0	SE.NAM	These two words contain the module name (as specified in the MACRO source file) in RAD50.
4	SE.IDN	These two words contain the module .IDENT (as specified in the MACRO source file) in RAD50.
10	SE.BLK	This word specifies the block offset within the SIL to the first block of the module. In the case of a SAV format SIL this offset will be 0.
12	SE.STB	This word specifies the block offset within the SIL to the first block of the module's symbol table (see section 1.5.3). If the module does not have a symbol table, this word will be zero.
14	SE.STN	This word specifies the number of symbols in the symbol table for this module. If the module does not have a symbol table, this word will be zero.
16	SE.LOD	This word specifies the lowest address in the module's image, as follows: Monitor SIL — Low limit Runtime system — Low limit Non-PIC resident library — Lowest limit PIC resident library — 1 SAV format SIL and INIT.SYS — 0
20	SE.SIZ	This word contains the virtual size of the module. The virtual size added to the low address results in the address of the first illegal address for the module. The first illegal address for the module types are as follows: Monitor SIL — High limit Runtime system — 177776_8 Non-PIC resident library — Highest address PIC resident library — Highest address SAV format SIL — Highest address
22	SE.XFR	This word contains the virtual address for SAV format SILs. This word contains 1 for all other SIL types to specify that an address does not apply.
24	SE.SZD	This word specifies the size of the module, in disk blocks.
26	SE.OVB	This word specifies the block offset within the SIL to the module's overlay descriptors. If the module is not overlaid, this word will be 0. Any module which does not start on a 1000_8 boundary (ie. INIT.SYS), which has data below its start (ie. all RT-11 SAV format programs), or which is overlaid (ie. some resident libraries) requires overlay descriptors (see section 1.5.4).

30	SE.OVN	This word specifies the number of overlay descriptors for this module.
32	SE.OFF	This word specifies a word offset within the block specified by SE.BLK to the start of the module.
34		This word is unused and must be 0.
36	SE.XXX	This word is reserved for SAV format SILs and is used by the RT11 emulator.

1.5.3 SIL Table Entry

Every module within a SIL can contain a symbol table. These symbols are used by the PATCH option of INIT and by the ONLPAT program when patching SILs.

The symbol table for a module consists of one or more blocks of symbol table entries. Each symbol table entry contains the value and optional overlay descriptor number for each global symbol specified (in a .STB file) when the SIL was created.

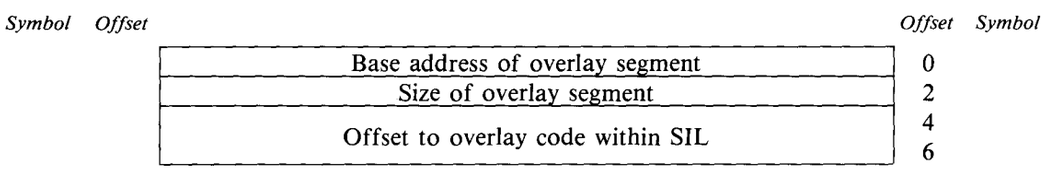


Offset	*Symbol*	*Description*
0		These two words contain the symbol name in RAD50. If the symbol is less than six characters long, it is padded with trailing spaces.
4		If this symbol is contained in an overlay, this word will contain the overlay descriptor number for the overlay segment which contains the symbol. If the symbol is not in an overlay, this word will contain a zero.
6		This word contains the numberic value of the associated global symbol.

1.5.4 SIL Overlay Descriptor Entry

Every module within a SIL that uses overlays must have an overlay descriptor table. Each entry in this table defines where an overlay is located in the SIL and where it should be loaded in memory. The first entry corresponds to the root module of the program. The remaining entries correspond to overlay segments.

Each Overlay Descriptor Entry has the following format:



Offset	*Symbol*	*Description*
0		This word specifies the lowest virtual address used by the overlay. The overlay code is loaded starting at this address.
2		This word specifies the size of the overlay segment, in bytes.
4		These two words specify the byte offset from the beginning of the SIL to the start of the overlay code. The word at offset 4 is the most significant word.

STRAP BLOCK

n every disk contains a bootstrap. The bootstrap on the system disk contains device dependant the RSTS/E monitor. The bootstrap on a non-system disk (ie. a public or private pack) prints ing the user that they have attempted to boot a data disk.

a system disk has a specific format. The first 22 words contain static information used by e or by INIT. Following this is device specific code for reading the required program (normally ry and for relocating the bootstrap to location 157000_8. The remaining space in the bootstrap list of clusters to load into memory at a specified load address. The last three words of the d to store the current date and time information, in case of a system restart.

disk is bootstrapped, the hardware reads the first block of the specified disk into memory, n zero. It then jumps to location 0 and begins executing the bootstrap code contained in the

le first copies itself into high memory, beginning at location 157000_8. It then uses device specific cluster specified in the bootstrap block into memory beginning at the specified load address. usters have been read into memory, the bootstrap jumps to the transfer address and the system

set		*Offset*	*Symbol*
NOP instruction		0	B.BOOT
Branch to setup code		2	
6		4	B.VE04
HALT		6	
10		10	B.VE10
HALT		12	
Device cluster size		14	B.DCS
Device CSR base		16	B.CSR
Device name (in ASCII)		20	B.NAME
JMP @(PC)+		22	B.JMP
Transfer address		24	B.XFER
Booted unit number		26	B.UNIT
Unit number shifted for controller		30	B.CSRU
Write function	Read function	32	B.RFUN
I/O function code		34	B.FUNC
Block number (LSB)		36	B.BLKL
Block number (MSB)		40	B.BLKH
Buffer address (LSB)		42	B.MEML
Buffer address (MSB)		44	B.MEMH
Xfer word count		46	B.TWC
Reset entry point		50	B.RSET
Read/write entry point		52	B.READ
Special function entry point		54	B.SPEC
Device specific code for loading specified blocks from disk		56	
Map of blocks to load			
0		772	B.DATE
0		774	
0		776	

Offset	*Symbol*	*Description*
0	B.BOOT	This word contains a NOP instruction (240_8), as required by the DEC standard for hardware bootstraps.
2		This word contains a branch to the code that performs initial setup and relocation.
4	B.VE04	This word contains a 6 and the following word contains a 0 (a HALT instruction). If a trap to 4 occurs, it will vector to location 6, where it will halt.
10	B.VE10	This word contains a 12_8 and the following word contains a 0 (a HALT instruction). If a trap to 10_8 occurs, it will vector to location 12_8, where it will halt.
14	B.DCS	This word specifies the device cluster size for the disk containing the bootstrap. Note that this is the device cluster size and not necessarily the pack cluster size. This word will be 0 for a tape bootstrap.
16	B.CSR	This word contains the address of the CSR register in the controller. All references to device registers will be made as offsets from this address.
20	B.NAME	This word contains the device name as two ASCII bytes.
22	B.JMP	This word originally contains the PDP-11 instruction JMP @(PC)+, which jumps to the location pointed to by the following word, B.XFER. When a SIL is installed (using the INSTALL option of INIT) this word changes to MOV (PC)+,PC. The effect of these two instructions is the same. The new instruction sequence is used as a flag to INIT to signify that a SIL has been installed.
24	B.XFER	This word contains the address to transfer control to after completing the bootstrap load. The transfer address is supplied by the HOOK program when the bootstap is initialized.
26	B.UNIT	Bits <0:2> of this word are used by the device specific bootstrap code to store the unit number from which the device was booted. INIT accesses this word to determine which disk unit the system was bootstrapped from. This value is normally 0 on the disk.
30	B.CSRU B.MMU	This word is used by the device specific bootstrap code for non-UDA disks and magtape to store the desired unit number shifted appropriately to allow its use is commands to the controller. For UDA disk bootstraps, this word contains the address divided by 100_8 of the buffer to use as an MSCP communications area.
32	B.RFUN	This byte is the value to load into the CSR of the disk controller to do a read. It is 0 for tape bootstraps.
33	B.WFUN	This byte is the value to load into the CSR of the disk controller to do a write. It is 0 for tape bootstraps.
34	B.FUNC	This word is the value to load into the CSR of the disk controller to do the first required function (initialized to a READ function). It is 0 for tape bootstraps.
36	B.BLKL B.SPFC	This word specifies the least significant 16 bits of the disk block number to access for disk I/O. This word also specifies the special function code for magtape special functions. The only special functions currently supported are rewind (3) and skip forward (4).
40	B.BLKH	This word specifies the most significant 16 bits of the disk block number to access for disk I/O. This word will be 0 for magtape bootstraps.
42	B.MEML	This word specifies the least significant 16 bits of the buffer address used for I/O. The size of the buffer is specified by B.TWC (offset 46).
44	B.MEMH	This word specifies the most significant 6 bits of the buffer address used for I/O. The most significant 10 bits of this word will always be 0.

46	B.TWC B.PARM	This word is used for several different purposes. It originally specifies the length of the device specific bootstrap code, in bytes. It is used during the operation of the bootstrap to specify the number of words to transfer in an I/O operation and to specify the number of records to skip in a magtape special function.
50	B.RSET	This location is the subroutine entry point used to perform any device specific initialization or reset and to select the specified unit for operation.
52	B.READ	This location is the subroutine entry point used to perform I/O operations. B.FUNC, B.BLKL, B.BLKH, B.MEML, B.MEMH, and B.TWC define the desired function, block number, buffer location and word count, respectively.
54	B.SPEC	This location is used by magtape bootstraps as the subroutine entry point used to perform a special function operation. This location contains device specific code in disk bootstraps (see offset 56).
56		The words between this location and the beginning of the load map contain device specific code for the initialization, I/O and magtape special functions. This area also contains code that copies the bootstrap block into high memory beginning at location 157000_8. It then copies the date and time words (from location 1000_8) into offset B.DATE of the relocated bootstrap block.
772	B.DATE	The three words beginning at this offset are used to store date and time information during a bootstrap operation. Immediately preceding this location is a table that specifies the blocks to be loaded into memory by the bootstrap loader. This table contains a series of three word entries. The first word (offset -2) specifies the number of words to load. The second and third words (offsets -4 and -6) are the MSB and LSB, respectively, of the FIP block number to load. This table grows backwards toward the beginning of the bootstrap. It is terminated by a 0 word count. One entry is needed in the table for each non-contiguous area of INIT.SYS to be loaded. Any space between the highest address of the device specific code and the lowest address of the load map is unused and will contain zeroes.

1.7 BAD BLOCK FILE

Most large disk packs contain some number of blocks that cannot be read for one reason or another. RSTS/E handles these bad blocks by allocating them to a bad block file, [0,1]BADB.SYS. Bad blocks are made known to the monitor in one of three ways: (1) Pattern tests using DSKINT (2) Block locations recorded on the disk pack at the factory (3) Bad blocks found during normal use.

When a block is added to the bad block file (either automatically or manually) the pack cluster that contains the bad block is allocated to the BADB.SYS file. A Retrieval Entry is made for the bad cluster and the file status is updated to show the change in size. By allocating the bad cluster to this file, it will never be available for allocation to another file during normal operation.

The BADB.SYS file is treated as a normal file in every way. It is never contiguous and has the no-delete bit (US.NOK) set.

Chapter 2

MONITOR TABLES

The monitor keeps track of everything being done on the system. It assigns memory to jobs as they need it. It swaps jobs in and out of memory to make room for other jobs. It schedules each job so that it receives a fair amount of the computer's resources. It interfaces to all the peripherals on the system. It also takes care of everything necessary to divide each disk into files so that they can be used for many purposes at once.

To provide all this functionality, the monitor needs a lot of information. It keeps this information in tables that are set up for specific purposes.

Several tables are used to control the memory and CPU usage of user jobs. These tables are described in sections 2.1, 2.2, and 2.3. Other tables control file and device usage. These tables are described in sections 2.4 and 2.5. Still other tables are used for system calls and miscellaneous functions. These tables are described in sections 2.6 and 2.7.

The symbols used in the discussion of monitor tables are defined as local symbols in the KERNEL.MAC and TBL.MAC files that are supplied with the sysgen kit. The symbols defined in these files should be used whenever possible to refer to monitor table information.

2.1 JOB CONTROL

RSTS/E can support up to 63 simultaneous jobs. Each of these jobs can be either a user at a terminal or a detached program. The job control structures allow the monitor to share resources (such as CPU time) properly among all users. In addition, they provide the means to access the information necessary for almost every other service provided by the monitor, such as device and file handling.

For example, the scheduler uses the information in the job descriptor block to determine which job to run. The memory manager uses the memory control sub-block and the residency quantum to set up memory management registers and perform swapping, if necessary.

The job control structures consist of a combination of tables and blocks. The size of the tables is determined by the number of jobs specified at sysgen time. The size of a block is typically 16 words.

The tables typically contain one word or byte for each possible job on the system. As a job's status changes, information in the tables is changed, but the size of the tables remain the same.

The job control blocks, with the exception of the Secondary Job Data Block (JDB2), exist only while a job exists. They are created (from small buffers) when a job is first created and deleted when the job is removed (eg. by logging out).

The location of specific tables can be determined using the GET MONITOR TABLES SYS calls. Once you know where a table starts, the values within the table can be accessed by adding the required offset to the starting address of the table. The following example gets the address of the Job Data Block (JDB) for job 5 (see section 2.1.1 and 2.1.2 for information about JOBTBL and JDB):

```
10 JOBTBL%=SWAP%(CVT$%(MID(SYS(CHR$(6%)+CHR$(-3%)),11%,2%)))
        !Get starting address of job table (JOBTBL)
20 JDBPTR%=PEEK(JOBTBL%+(5%*2%))
        !Get pointer to JDB for job 5 from JOBTBL
```

2.1.1 JOBTBL - Job Table

The job table, JOBTBL, is the root of the job control structures. It points to the Job Data Block which, in turn, points to the other job related blocks. See Appendix A for information on related job control structures.

JOBTBL contains an entry for each possible job on the system. The job control information for each job can be accessed by using the job number times two as an offset from the beginning of JOBTBL. The value found at this location will be the address of the Job Data Block (see section 2.1.2) for that job. If a 0 is found, there is currently no job by that number.

The first word of JOBTBL (at offset 0) corresponds to the entry for the null job. It contains the address of the JDB for the system job currently using FIP, if any. System jobs are used for error logging and network service processing. They have the following function and job numbers:

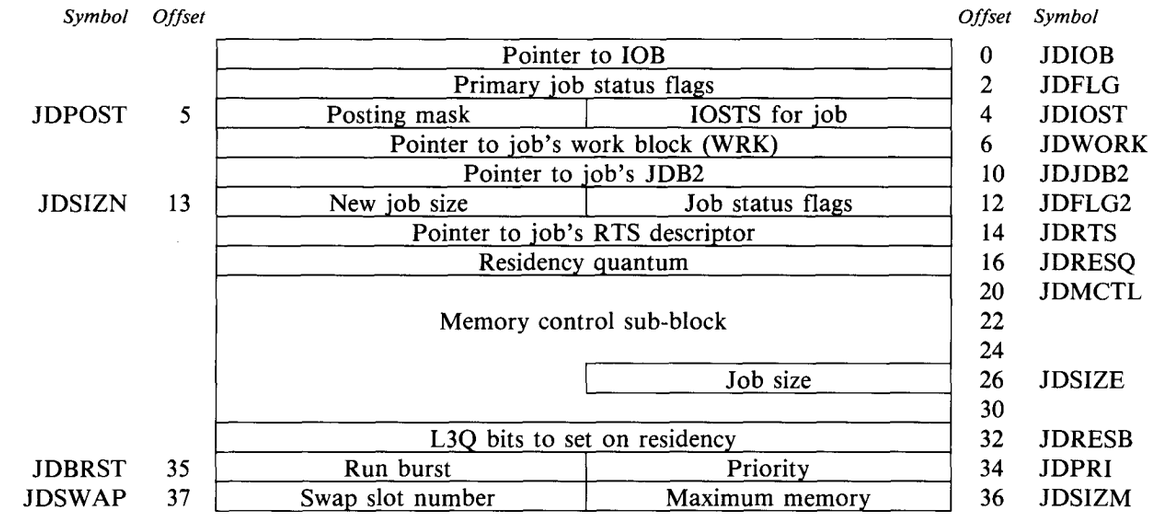
Job	*Name*	*Function*
0.5	ERRLOG	Error logger
1.5	NSP	Network services protocol handler
2.5	TRN	Transporter (interprocessor message routing)

The last word in JOBTBL contains a -1 to signify the end of the table. Thus, the total length of JOBTBL, in words, is JOBMAX (the maximum number of jobs, specified at sysgen time) plus two.

2.1.2 JDB - Primary Job Data Block

The Primary Job Data Block (JDB) contains the most commonly used information about a job. It is pointed to by the entry in JOBTBL corresponding to its job number (see section 2.1.1). The JDB points to three other job control blocks for the job: JDB2, IOB and WRK (see sections 2.1.3, 2.1.4 and 2.1.5 respectively).

Information in the JDB is used by many routines within the monitor. The scheduler uses JDPRI and JDBRST to determine which job to run next. The memory manager uses JDMCTL, JDSIZE, JDSIZM and JDSIZN to set up the memory mapping registers and to schedule a swap-in. The swapper uses JDRESQ, JDSWAP, JDMCTL and JDSIZE to swap jobs in and out of memory. The EMT handler uses JDIOST, JDFLG and JDWORK to process system and I/O calls.

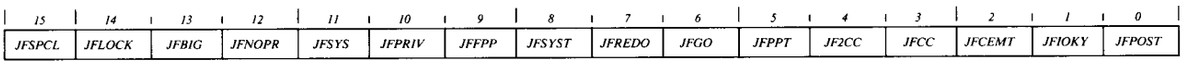


Offset	*Symbol*	*Description*
0	JDIOB	This word contains the address of the I/O Data Block (IOB) for this job (see section 2.1.4).
2	JDFLG	This word contains the primary job status flag bits and a copy of the job's keyword bits (see section 2.1.2.1)
4	JDIOST	This byte contains the error code to be returned to the programmer after the completion of the current monitor call. If the JFIOST bit is set in the job status flags (JDFLG), JDIOST is moved to the IOSTS word of the user's FIRQB before job execution continues. A value of 0 indicates that no error occurred. (See the *System Directives Manual* for more information on the FIRQB and IOSTS.)
5	JDPOST	If the JFPOST bit is set in the job status flags (JDFLG), this byte is used to specify which information in the job's Work Block (WRK) is to be posted to the job's FIRQB or XRB. If the value in JDPOST is positive, it is used as a word offset into MSKTBL, a table of bit masks for posting to the FIRQB. If it is negative, the low order 7 bits are used as a bit mask for posting to the XRB. Each bit in the mask corresponds to a word in the Work Block to copy to the job's XRB if the bit is set.
6	JDWORK	This word contains the address of the Work Block (WRK) for this job (see section 2.1.5),
10	JDJDB2	This word contains the address of the Secondary Job Data Block (JDB2) for this job (see section 2.1.3).
12	JDFLG2	This byte contains the secondary job status flags (see section 2.1.2.2).
13	JDSIZN	This byte specifies the size (in K-words) to make this job the next time it is swapped in. The memory manager uses this location when a job must be swapped out to find additional memory as it attempts to grow in size. The job's current size setting (JDSIZE) will be updated when the job is swapped back in.

14	JDRTS	This word contains the address of the Runtime System Descriptor Block (RTS) for the runtime system currently in use by this job (see section 2.3.4). If the disappearing RSX runtime system is in use by this job, JDRTS will contain the address of the null runtime system descriptor block (NULRTS).
16	JDRESQ	This word contains the current residency quantum for the job. The residency quantum is used to reduce memory thrashing. When a job is brought into memory it is given a residency quantum. Each time the job executes, does disk I/O or uses the file processor (FIP) its residency quantum is reduced in proportion to the amount of time it ran or the estimated time required to complete the service request. If a job is stalled on I/O to a non-disk device, its residency quantum is immmediately reduced to zero to allow it to be swapped. As long as the job's residency quantum is non-zero, the job is not eligible to be swapped out.
20	JDMCTL	These five words are the Memory Control Sub-Block (MCB) for the job (see section 2.3.1).
26	JDSIZE	This byte (within the Memory Control Sub-Block) specifies the current size of the job, in K-words.
32	JDRESB	This word specifies the bits to be set in the second Level Three Queue word when the job is made resident. It is used by the monitor to notify itself when a job is made resident so that a function that required the job to be resident can be continued (see section 2.2).
34	JDPRI	This byte specifies the job's priority. It can range from -128 to +127. The scheduler uses this byte, along with other information, to determine which job to run next.
35	JDBRST	This byte specifies the job's run burst. The run burst is the amount of time (in clock ticks) the job may execute without stalling before the scheduler is called to schedule another job.
36	JDSIZM	This byte specifies the job's private memory size maximum, in K-words.
37	JDSWAP	If the job is currently swapped out, this byte specifies the swap file slot number of the slot containing the job. The swap file number is in bits <6:7> and the swap slot number is in bits <0:5>. Swap file numbers 0 through 3 correspond to the swap files SWAP0.SYS through SWAP3.SYS, respectively.

2.1.2.1 JDFLG - Primary Job Status Flags

The job status flags contained in JDFLG are defined as follows:



Bit	*Symbol*	*Description*
<0>	JFPOST	The monitor checks this bit before making a job runnable. If it is set, the information in JDPOST is used as mask for updating the job's FIRQB or XRB (see section 2.1.2). The information in J2PPTR and J2PCNT in JDB2 (see section 2.1.3) may also be used to post large amounts of data to a buffer in the user program.
<1>	JFIOKY	If this bit is set when a job is made runnable, the job's keyword is updated in the job's image and the contents of JDIOST are posted to the job's FIRQB.
<2>	JFCEMT	If this bit is set, the monitor resident RSX support is used to post the job information indicated by JFIOKY, rather than using the standard monitor routines.
<3>	JFCC	This bit is set when at least one ^C is typed at the job's terminal. When the job becomes runnable, the P.CC pseudo-vector in the runtime system will be entered unless JF2CC is also set.

<4> JF2CC This bit is set wh en a ^C is typed and at least one ^C has already been typed since the job was last run. When the job becomes runnable, the P.2CC pseudo-vector in the runtime system will be entered.

<5> JFPPT If this bit is set when the job is made runnable, the floating point trap pseudo-vector, P.FPP, will be entered.

<6> JFGO If this bit is set when the job is made runnable, the I/O redo request specified by the JFREDO bit will be ignored. This bit is set if a user types ^C during a ^C interruptable I/O operation.

<7> JFREDO If this bit is set when a job becomes runnable (and JFGO is not also set), the device driver that requested I/O redo will be reentered at it's SER$xx entry point (see section 3.2.5).

<8> JFSYST This bit is set if the job can use temporary privileges. This bit is one of the keyword bits.

<9> JFFPP If this bit is set, the contents of the floating point hardware registers (if any) will be saved and restored along with the job image. This bit is one of the keyword bits.

<10> JFPRIV This bit is set if the job is logged into a privileged account. This bit is one of the keyword bits.

<11> JFSYS This bit is set if the job is currently running with temporary privileges. This bit is one of the keyword bits.

<12> JFNOPR This bit is set if the job is running without having been logged in. This bit is one of the keyword bits.

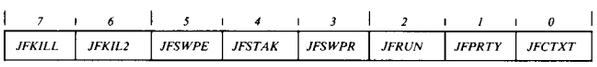
<13> JFBIG If this bit is set, the job can exceed its private memory size (as defined in JDSIZM). This bit is one of the keyword bits.

<14> JFLOCK This bit is set if the job is locked in memory. This bit is one of the keyword bits.

<15> JFSPCL This bit is set is special processing is required before running the job. The flag bits in JDFLG2 specify the special processing to be performed (see section 2.1.2.2).

2.1.2.2 JDFLG2 - Secondary Job Status Flags

The job status flags contained in JDFLG2 have the following meaning when set:



7	6	5	4	3	2	1	0
JFKILL	JFKIL2	JFSWPE	JFSTAK	JFSWPR	JFRUN	JFPRTY	JFCTXT

Bit	*Symbol*	*Description*
<0>	JFCTXT	The job's context should be saved.
<1>	JFPRTY	The special condition shown by JFSPCL is a memory parity error.
<2>	JFRUN	The special condition shown by JFSPCL is a new program run request.
<3>	JFSWPR	The special condition shown by JFSPCL is a runtime system or resident library load failure.
<4>	JFSTAK	The special condition shown by JFSPCL is a stack overflow.
<5>	JFSWPE	The special condition shown by JFSPCL is a swap error.
<6>	JFKIL2	The logout phase of killing a job has completed. The control structures associated with a job should now be released.
<7>	JFKILL	The job should be killed.

2.1.3 JDB2 - Secondary Job Data Block

The Secondary Job Data Block (JDB2) contains information about the job that is used less often or is less time critical than the information in the JDB. Its primary use is for accounting and directory information.

Symbol	*Offset*		*Offset*	*Symbol*
		Unposted clock ticks	0	J2TICK
		CPU time (LSB)	2	J2CPU
		Connect time	4	J2CON
		Kilo-core-ticks (LSB)	6	J2KCT
		Device time	10	J2DEV
J2CPUM	13	CPU TIME (MSB) | KCT(MSB)	12	J2KCTM
		Program name	14	J2NAME
			16	
		Default runtime system pointer	20	J2DRTS
		Receiver ID block pointer	22	J2MPTR
		Large data posting pointer	24	J2MPTR
		Large data posting byte count	26	J2PCNT
		Project-Programmer number	30	J2PPN
		DCN of first UFD block	32	J2UFDR
		Pointer to Window Descriptor Block	34	J2WPTR
		Extended job flags	36	J2FLAG
		Pointer to SPAWN process control area	40	J2SPWN
		Pointer to EMT logger message packet	42	J2EMLP
		^T CPU time		J2CPUI

Offset	*Symbol*	*Description*
0	J2TICK	This word is incremented at each interrupt when the job is executing. When the job is descheduled, this value is converted to the equivalent number of 1/10th seconds and added to J2CPU. Any amount less than 1/10th second is left in this word. The units of this word depend on how fast the clock is interrupting. For a KW11L clock, running at 60 hertz, the units are 1/60th seconds.
2	J2CPU	This word contains the low order 16 bits of the total CPU time used by this job, through the last time J2TICK was posted. The units of this value are 1/10th seconds.
4	J2CON	This word contains the total connect time, in minutes, for this job. Connect time is only computed while a job is logged in.
6	J2KCT	This word contains the low order 16 bits of the job's kilo-core-ticks. One kilo-core-tick is the use of 1 K-word of memory while executing for 1/10th second. Using 2 K-words for 1/10th second is two kilo-core-ticks.
10	J2DEV	This word contains the total device time for this job, in device-minutes. A device minute is the use of one device for one minute. Using two devices for one minute is two device-minutes.
12	J2KCTM	This byte contains the high order 8 bits of the job's kilo-core-ticks (see J2KCT, above).
13	J2CPUM	This byte contains the high order 8 bits of the job's CPU time (see J2CPU, above).
14	J2NAME	These two words contain the program name, in RAD50. The program name is specified using the .NAME system call. All the standard runtime systems issue this call to post the program name when a program is run. The contents of these words are for information only, and are unused by the monitor.

20	J2DRTS	This word is the address of the Runtime System Descriptor Block (RTS) for the job's default runtime system (see section 2.3.4). When a running program exits, control returns to this default runtime system. If the default runtime system is no longer installed when the program exits, the system default runtime system will be used instead.
22	J2MPTR	If this job is a message receiver, this word contains the address of its Receiver ID Block (RIB) (see section 2.6.1). If not, this word will be zero.
24	J2PPTR	This word is used as a pointer to a large monitor buffer to be used to transfer information to or from a user program. It is normally used for large message send/receive buffer transfers. If the least significant five bits of the pointer are zero, the pointer is an address in the small buffer area. If the least significant five bits of the pointer are non-zero, it is a "contorted" pointer into the extended buffer pool. The actual address has been rotated left seven bits to ensure that the least significant bits are non-zero.
26	J2PCNT	This word specifies the number of bytes to transfer to or from the buffer specified by J2PPTR.
30	J2PPN	This word contains the job's PPN. The project number is in the high byte (offset 31). The programmer number is in the low byte (offset 30). If the job is not logged in, this word will be zero.
32	J2UFDR	This word contains the device cluster number (DCN) of the first cluster of the user's UFD on SY0:. The value of this word is undefined if the job is not logged in.
34	J2WPTR	If the job is attached to any resident libraries, this word contains a pointer to the job's first Window Descriptor Block (WDB) at offset W.WIN1 (see section 2.3.6). If the job is not attached to any resident libraries, this word will be zero.
36	J2FLAG	This word contains extended job status flag bits. The only bit currently defined is bit 0 (symbolically J2FSPW). This bit specifies that this job is being spawned.
40	J2SPWN	This word is used during job spawning to hold a pointer to a large buffer that contains information about the job being spawned. Use of this large buffer allows the job requesting the spawn to be removed from memory.
42	J2EMLP	If the EMT logger option was included during system generation, this word is used to contain a pointer to the pending EMT message buffer or to contain a value of 177640_8 used to specify that EMT logging is not desired. If the EMT logging option was not included during system generation, this word will not be allocated.
	J2CPUI	If the one-line mini-systat option was included during system generation, this word contains the least significant 16 bits of the CPU time used by this job when the last ^T mini-systat was taken. If this option was not included, this word will not be included. This word is at either offset 42 if EMT logging was not included or offset 44 if EMT logging was included.

2.1.4 IOB - I/O Block

The I/O Block (IOB) is used to access information about each or a job's open channels. It contains one entry for each of the possible 16 channels. (Note: BASIC-PLUS only allows 12 channels because channel 0 is the user's terminal and channels 13-15 are used internally by the BASIC-PLUS interpreter for its temp file, source file, and compiled file, respectively.)

The IOB is 16 words long. Each word is zero if the corresponding channel is closed and non-zero if it is open. If the channel is open and the device is not a disk, this word will contain a pointer to the Device Data Block (DDB) for the associated device. If the device is a disk, this word will contain a pointer to the Window Control Block (WCB) for the associated disk file.

One exception is the entry in the IOB for channel 0. This entry corresponds to the DDB for the job's terminal. If the job is detached and no longer owns a console terminal, this entry will still point to a terminal DDB. However, the ownership byte (DDJBNO) within the DDB (see section 2.5.1) will not contain the job's job number (times two).

Symbol	*Offset*		*Offset*	*Symbol*
		DDB pointer for channel 0	0	
		DDB, WCB or SCB pointer for channel 1	2	
		DDB, WCB or SCB pointer for channel 2	4	
		DDB, WCB or SCB pointer for channel 3	6	
		DDB, WCB or SCB pointer for channel 4	10	
		DDB, WCB or SCB pointer for channel 5	12	
		DDB, WCB or SCB pointer for channel 6	14	
		DDB, WCB or SCB pointer for channel 7	16	
		DDB, WCB or SCB pointer for channel 8	20	
		DDB, WCB or SCB pointer for channel 9	22	
		DDB, WCB or SCB pointer for channel 10	24	
		DDB, WCB or SCB pointer for channel 11	26	
		DDB, WCB or SCB pointer for channel 12	30	
		DDB, WCB or SCB pointer for channel 13	32	
		DDB, WCB or SCB pointer for channel 14	34	
		DDB, WCB or SCB pointer for channel 15	36	

2.1.5 WRK- Work Block

The Work Block (WRK) is a scratch area used to hold information that normally would only be contained in the user's program area. Copying this information into the Work Block allows device drivers, the file processor and other modules within the monitor to access information in a user's FIRQB or XRB without requiring an additional memory management register to access them.

The contents of the FIRQB are copied into the Work Block on SYS calls and file or device opens. The contents of the XRB are copied into the Work Block on I/O requests or special function (.SPEC) calls. Information from the Work Block is copied back to the user's FIRQB or XRB when the monitor function completes if the JFPOST bit is set in the user's JDB (see section 2.1.2.1).

2.1.6 JBWAIT, JBSTAT - Job Status Tables

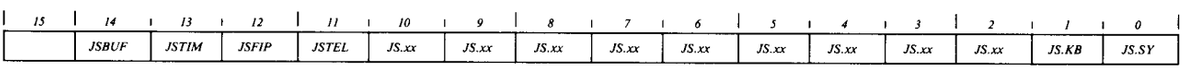
Two tables, JBWAIT and JBSTAT, are used to determine if a job is runnable. Each of these tables contain one word for every possible job on the system, including the null job (job 0). They are accessed using the job number times two as an offset from the beginning of the table.

When a job is stalled for any reason (such as waiting on I/O), a bit is set in the job's JBWAIT entry to show what the job is waiting for. The corresponding JBSTAT entry is set to zero. When the condition the job is waiting for has been met, the same bit that was set in JBWAIT will be set in JBSTAT.

When the scheduler wants to determine if a job is runnable, it performs a logical AND of the job's JBSTAT table entry with its JBWAIT table entry. If the result is non-zero, the job is runnable. If the result is zero, the job is stalled waiting for something and cannot be run.

When a job is stalled for hibernation, both the JBWAIT and JBSTAT words are set to zero. JBWAIT will be set to a non-zero value when the job is reattached to a terminal.

The bit values within the JBWAIT and JBSTAT entries show why a job is stalled. They have the following meaning.



Bit	*Symbol*	*Description*
<0>	JS.SY	This bit is used for I/O on all devices that are not ^C interruptable. These devices are normally: NL, RJ, MT, MM, MS, DT, DX, DD, XK and all disks.
<1>	JS.KB	This bit is used for terminal input.
<2>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<3>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<4>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<5>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<6>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<7>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<8>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<9>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<10>	JS.xx	This bit may be assigned at sysgen time to a device that is ^C interruptable.
<11>	JSTEL	This bit is used for terminal output.
<12>	JSFIP	This bit is used for FIP (SYS call) waits.
<13>	JSTIM	This bit is used for timeouts from various time restricted events, such as .SLEEP, message receive timeout, etc.
<14>	JSBUF	This bit is set when a monitor routine or device driver checks for small buffer availablity and there are less than 10 small buffers available on the entire system. The job will be stalled until there are more small buffers available.
<15>		This bit is not used, but is reserved for future use.

2.1.7 JOBCLK - Job Sleep Time Table

Each entry in JOBCLK contains the sleep time counter for the corresponding job. If the entry is non-zero, the corresponding job is in a sleep wait state. The value of the entry specifies the number of seconds remaining in the job's sleep time. When the entry becomes zero, the job will continue execution. The entries in this table are accessed by job number times two.

2.2 LEVEL THREE QUEUE

A typical problem in operating system design is the prevention of a mutual exclusion condition where one routine in the monitor calls a second routine which eventually calls the first one. The result is a circular call that never gets resolved. Another problem is the requirement that certain routines operate at a different processor priority level than the routine that envoked them.

RSTS/E solves both of these problems using the Level Three Queue (L3Q) structure. The Level Three Queue is a pair of words that are used as a set of 32 bits. Each of these bits is assigned to a specific routine in the monitor.

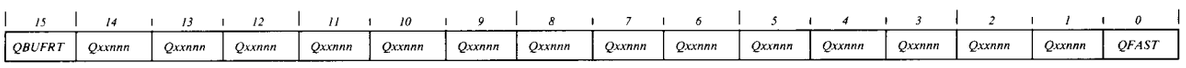
When a routine within the monitor determines that another routine in the monitor should be executed, it sets the appropriate bit in L3Q. Before returning to a user program, L3Q is checked. If any bits are found set, the associated routine will be executed. This process repeats until no bits remain set in L3Q. The monitor then continues execution of the user program.

For example, if the real-time clock driver determines that a second has passed, it sets the bit in L3Q that envokes the "once a second, every second" timer service. The clock driver will also set the bit to call the scheduler if the current user's runburst has been exhausted.

The Level Three Queue is divided into two words. The first word (L3QUE) is mainly used for L3Q bits assigned to device drivers (see section 3.2.12). The second word (L3QUE2) is used to envoke routines within the monitor.

The order of the bits in L3Q determines the relative priority of the associated routines. Bit 0 of the first word of L3Q corresponds to the highest priority routine. Bit 15 of the second word in L3Q corresponds to the lowest priority routine.

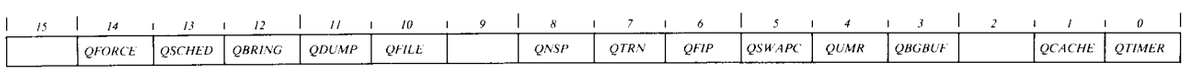
The bits in L3QUE have the following meaning when set:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
QBUFRT	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	Qxxnnn	QFAST

Bit	*Symbol*	*Description*
<0>	QFAST	Reserve for use by some fast device driver. The DU: disk driver redefines this bit as QPHCON for use with add-on disks and the UDA5O controller.
<1:14>	Qxxnnn	These bits are available for use by device drivers. The individual bits are assigned during the TBL assembly using the L3Q bit requirements specified in the DEVICE macro used for each device driver. The bit name is formed by replacing "xx" by the device name and "nnn" by the L3Q bit name assigned by the driver. For example, QMTCON specifies the CON bit (used for a continuation entry point) used by the MT driver.
<15>	QBUFRT	Return small buffers to the monitor pool.

The bits in L3QUE2 have the following meaning when set:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	QFORCE	QSCHED	QBRING	QDUMP	QFILE		QNSP	QTRN	QFIP	QSWAPC	QUMR	QBGBUF		QCACHE	QTIMER

Bit	*Symbol*	*Description*
<0>	QTIMER	Once a second, every second timer service.
<1>	QCACHE	Disk cacheing has completed.
<2>		Reserved.
<3>	QBGBUF	Unstall processes waiting on big buffers.
<4>	QUMR	Unstall processes waiting on unibus mapping registers.
<5>	QSWAPC	Swap completion.

<6>	QFIP	FIP completion or continuation.
<7>	QTRN	DECNET transfer completion or continuation.
<8>	QNSP	DECNET network service completion or continuaton.
<9>		Reserved.
<10>	QFILE	User disk I/O completion.
<11>	QDUMP	Deschedule the current job.
<12>	QBRING	Run the memory manager to ensure residency of a job.
<13>	QSCHED	Run the scheduler.
<14>	QFORCE	Deschedule the current job.
<15>		Reserved.

2.3 MEMORY CONTROL

Memory is used for many different purposes. The monitor and cache buffering use a large chunk. Runtime systems and resident libraries take their toll. And, of course, let's not forget application programs. They are what we bought this machine for in the first place.

With all these demands on memory, the monitor has to make some pretty smart decisions to control this resource efficiently. The primary way it provides this control is through memory control sub-blocks.

2.3.1 MCB - Memory Control Sub-Blocks

The available memory on a system is typically broken down into many pieces, each being used for a different purpose. Memory Control Sub-blocks (MCBs) are used to keep track of each of these pieces of memory.

An MCB is not a structure by itself. It is a part of other structures that describe functions which use memory, such as the Job Data Block (JDB). The structures which contain Memory Control Sub-blocks are: Job Data Blocks (JDB), Runtime System Descriptor Blocks (RTS), Library Descriptor Blocks (LIB), the RSTS/E monitor, XBUF, locked out memory and non-existent memory.

The MCB is contained at different offsets within different types of structures. The offset can be used to identify the type of structure that contains the MCB if all that is available is a pointer to the MCB. The following table lists the offsets and the type of structure they relate to:

Offset	*Symbol*	*Description*
0	ML.MON	Monitor, ODT or tail MCB.
2	ML.XBF	XBUF.
4	ML.LCK	Locked out memory (locked by INIT or parity error).
6	ML.NXM	Non-existent memory.
10	ML.RTS	Runtime System or Resident Library Descriptor Block.
12-16		Unused.
20	ML.USR	User program.

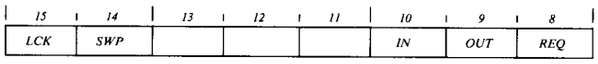
The Memory Control Sub-blocks are always in one of three states: (1) linked into a list of current memory users, (2) linked into a list of users that desire memory residency, (3) not in memory and not desiring memory residency.

The Memory Control Sub-block has the following format:

Offset	*Symbol*	*Description*
0	M.PPRV	Pointer to the previous MCB in this memory control list at offset M.PNXT.
2	M.PNXT	Pointer to the next MCB in this memory control list.
4	M.TSIZ	Number of K-words mapped by this MCB. This size includes the amount of memory actually used, plus any available memory that follows it.
6	M.SIZE	This byte specifies the number of K-words actually used. This size subtracted from M.TSIZ yields the number of K-words mapped by this MCB which are available for allocation to other uses.
7	M.CTRL	This byte contains the memory status information about the portion of memory mapped by this MCB.
10	M.PHYA	This word contains the physical starting address of the piece of memory mapped by this MCB, divided by 100_8.

2.3.1.1 M.CTRL - Memory Status Information

The memory status information bits contained in M.CTRL have the following meaning when set:



Bit	*Symbol*	*Description*
<8>	REQ	Residency is requested but the MCB is not linked into RESLST because it is currently being swapped out.
<9>	OUT	Entry should be removed from memory or is currently removed from memory.
<10>	IN	Entry should be brought into memory.
<11:13>		Unused.
<14>	SWP	A swap is desired. OUT and IN determine the direction of the swap.
<15>	LCK	The memory segment described by M.SIZE is not available for allocation for other uses or for swapping out.

Some typical combinations of bits in M.CTRL are:

LCK,SWP,OUT	The entry is resident but should be swapped out.
LCK,OUT	The entry is in the process of swapping out.
LCK,SWP,IN	The entry has been allocated memory and should be swapped in now.
LCK,IN	The entry is in the process of swapping in.
LCK	The entry is not available for swapping out.
OUT	The entry is not currently in memory and does not desire to be made resident.

2.3.2 MEMLST - Resident Memory List

All of the memory in a system is mapped by the resident memory list, MEMLST. As memory is divided among several different usages, the Memory Control Sub-blocks for each usage are linked into MEMLST in ascending address order. Thus, by following the links between the MCBs in MEMLST, we have seen all the memory on the system.

The memory control list is based at the location MEMLST. This location is the address of the first entry in the memory control list, rather than a pointer to the first entry as in most other linked lists. The first entry describes the memory used by the monitor and any free memory following it.

The memory control list always contains at least three entries. These are the root MCB, the system default runtime system, and the tail MCB. The root is actually the monitor MCB. The tail terminates the list and shows the highest memory location addressable on the system.

2.3.2.1 Root Memory Control Sub-Block

The first entry in the resident memory list is the root MCB. This entry starts at location MEMLST and describes the memory used by the monitor and any free memory following it.

The format of the root MCB is as follows:

Offset	*Symbol*	*Description*
0	M.PPRV	The link to the previous entry is zero since this is the first entry in MEMLST.
2	M.PNXT	This word contains a pointer to the next entry in MEMLST.
4	M.TSIZ	This word specifies the total of the monitor's size plus any free memory following it.
6	M.SIZE	This byte specifies the size of the monitor, in K-words.
7	M.CTRL	The LCK bit is set to show that the monitor's memory is not available for other uses.
10	M.PHYA	The starting physical address is 0 since the monitor always starts at location 0.

2.3.2.2 Tail Memory Control Sub-Block

The tail MCB is the last entry in MEMLST. It terminates the list and defines the highest memory address available on the system. The format of the tail MCB is as follows:

Offset	*Symbol*	*Description*
0	M.PPRV	The backward link points to the previous entry in MEMLIST at its M.PNXT entry.
2	M.PNXT	The forward link contains a zero to show that this is the end of the list.
4	M.TSIZ	The total size for this entry is set to a value of 1, but is not used by the monitor.
6	M.SIZE	The size of this entry is set to a value of 1, but not used by the monitor since the LCK bit is set in M.CTRL.
10	M.PHYA	The starting physical address of this memory segment corresponds to the address of the first non-existent memory on the system (divided by 100_8). Therefore, this value is the total system memory size (in K-bytes, divided by 100_8).

2.3.3 RESLST - Desired Residency List

When an entry is not currently resident in memory and wants to become resident, it is linked onto the end of the desired residency list, RESLST. The memory manager uses this list in a first-in/first-out basis to keep track of requests for memory beyond what is available in MEMLST.

When an entry is added to RESLST, the memory manager is scheduled to fulfill the desired residency request. If there is not enough memory available to honor the residency request, entries that are currently resident in memory are reviewed. Those which are eligible for swapping are then scheduled to be swapped out. Once sufficient memory is made available, the requestor is made resident and the MCB is removed from RESLST and added to MEMLST.

The first entry in the desired residency list is pointed to by the location RESLST. If no entries desire residency, RESLST will contain a zero.

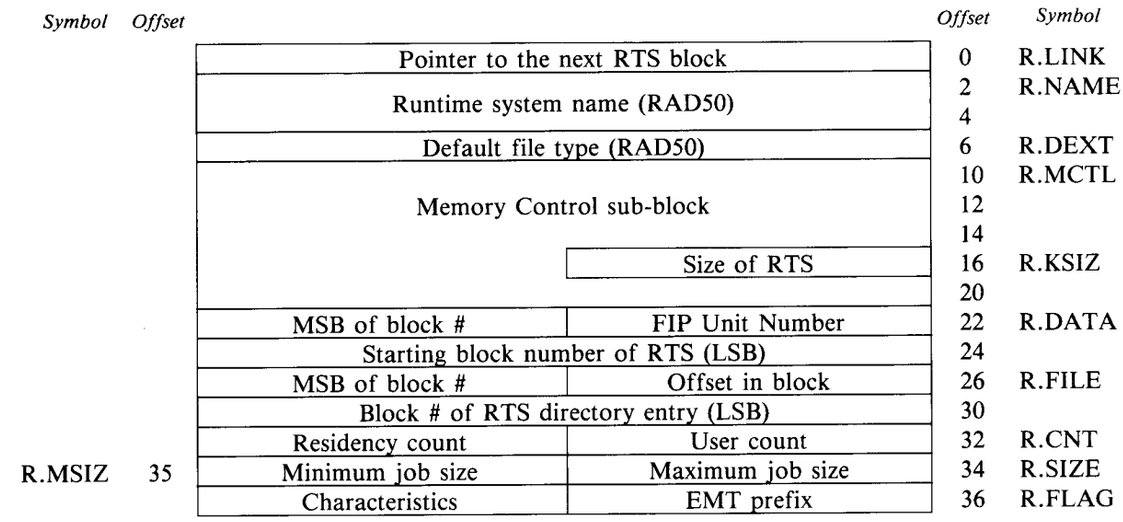
Memory control sub-blocks in RESLST have the following format:

Offset	*Symbol*	*Description*
0	M.PPRV	Pointer to the next entry in RESLST or 0 if this is the last entry.
2-5		Unused.
6	M.SIZE	The size of this entry, in K-words.
7	M.CTRL	The control bits LCK, SWP and IN are set to show that swap-in is desired.
10	M.PHYA	This word contains either a 0 to show that no specific memory address is required or it contains the desired memory address, divided by 100_8.

2.3.4 RTS - Runtime System Descriptor Block

Every runtime system that is currently installed in the system has a Runtime System Descriptor Block associated with it. This structure contains all the information about the runtime system, including its name, memory control information, disk address and characteristics.

The RTS block has the following format:

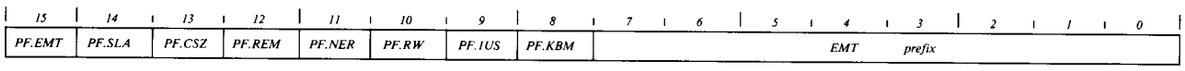


Offset	*Symbol*	*Description*
0	R.LINK	This word contains the address of the next RTS block in the list. If this entry is the last in the list, it will contain a 0.
2	R.NAME	These two words contain the runtime system name, in RAD50.
6	R.DEXT	This word contains the default file type, in RAD50, for executable files used by this RTS. If a RUN command is issued without specifying a file type for the file to be executed, this value will be used for the file type on files executed under this runtime system.
10	R.MCTL	These five words are the Memory Control Sub-block for the runtime system (see section 2.3.1).
16	R.KSIZ	This byte (within the Memory Control Sub-block) specifies the size of the runtime system, in K-words.
22	R.DATA	This byte is the FIP Unit Number of the disk containing the runtime system. It is used when loading the runtime system image and when closing the runtime system file when the runtime system is removed.
23		These three bytes contain the FIP Block Number (FBN) of the first block of the runtime system image. When a runtime system is loaded into memory it is accessed from the disk using this block number. Byte 23 is the most significant byte of the block number.
26	R.FILE	This byte is the offset, in words, to the Name Entry for the RTS file within the directory block specified at offset 27.
27		These three bytes contain the FIP Block Number (FBN) of the block that contains the UFD Name Entry for this runtime system. It is used to close the RTS file when the RTS is removed. Byte 27 is the most significant byte of the block number.
32	R.CNT	This byte contains a count of the number of jobs currently using this runtime system.

33		This byte contains a count of the number of jobs using this runtime system which are currently resident in memory. If this count is 0, the runtime system is eligible for "swapping out". If a runtime system is loaded with the /STAY switch, the high bit of this byte is set, ensuring that the residency count will never be 0 and the runtime system will always remain in memory.
34	R.SIZE	This byte specifies the maximum size (in K-words) for a job image using this runtime system.
35	R.MSIZ	This byte specifies the minimum size (in K-words) for a job image using this runtime system.
36	R.FLAG	The high byte of this word (offset 37) contains a set of bits that describe the characterstics of the runtime system (see section 2.3.4.1). If the PF.EMT bit (see section 2.3.4.1) is set in the high byte of the word, the low byte will contain the special EMT prefix value. See the *RSTS/E Programming Manual* for more information.

2.3.4.1 R.FLAG - Runtime System Characteristics

The runtime system characteristics flags contained in R.FLAG in the RTS block have the following meaning when set:



Bit	*Symbol*	*Description*
<0:7>		Special EMT prefix (see PF.EMT below)
<8>	PF.KBM	The runtime system can act as a keyboard monitor.
<9>	PF.1US	The runtime system is single user, non-sharable.
<10>	PF.RW	Map runtime system read/write.
<11>	PF.NER	Do not log errors occurring under this runtime system.
<12>	PF.REM	Remove the runtime system image from memory when R.CNT becomes 0.
<13>	PF.CSZ	Compute initial job size.
<14>	PF.SLA	Load runtime system at the address specified by M.PHYA of the MCB.
<15>	PF.EMT	Low byte of R.FLAG is the special EMT prefix code.

2.3.4.2 NULRTS - Disappearing RSX Runtime System

One of the options at sysgen time is to embed support for the RSX emulator into the monitor. When this is done, the RSX runtime system disappears after initiating program execution.

Every job on the system is required to have a runtime system associated with it at all times. The monitor meets this requirement when using the disappearing RSX runtime system by using the Null RTS Descriptor Block, NULRTS.

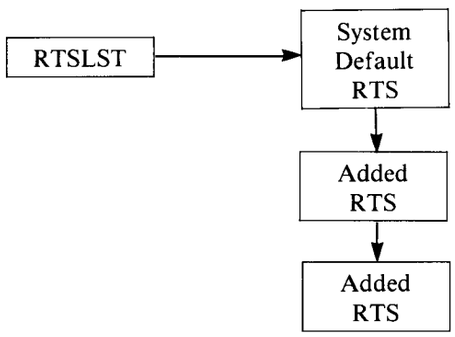
The Null RTS Descriptor Block is not linked into RTSLST (see section 2.3.4.3). It is only used to provide an RTS block for the Job Descriptor Block (JDB) to point to.

The format of the Null RTS Block is the same as a normal RTS block. All the fields contain 0 except the following:

Symbol	*Value*
R.NAME	"...RSX" in RAD50
M.CTRL	LCK bit set
R.SIZE	System swap maximum
R.MSIZ	1

2.3.4.3 RTSLST - Runtime System List

The RTS blocks are linked together in a list pointed to by the location RTSLST. The first entry is always the system default runtime system. It links to the other RTS blocks in the order displayed by SYSTAT. The following figure illustrates the linkages used for runtime system control:



2.3.4.4 DEFKBM - Pointer to Default RTS

The contents of DEFKBM is a pointer to the RTS block for the system default keyboard monitor. This runtime system is used as each user's runtime system unless they specify a different private runtime system.

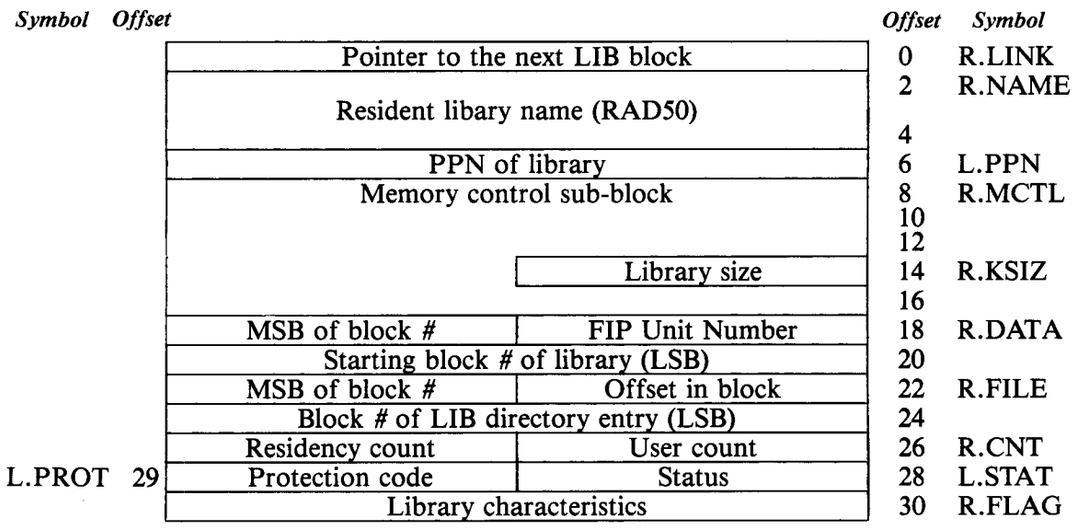
2.3.5 LIB - Resident Library Descriptor Block

Each resident library installed in the system is described by a Resident Library Descriptor block (LIB). A LIB block is very much like an RTS block in that it contains information about the resident library's name, memory control information, disk address and characteristics.

The LIB blocks for the currently installed libraries are kept in a linked list. The first element in the list is pointed to by the location LIBLST (which immediately follows RTSLST in memory). The following elements are linked by the first word of each LIB block.

Refer to the desciption of the .PLAS call in the *System Directives Manual* for a complete description of resident library support and the use of memory windows.

The LIB block has the following format:

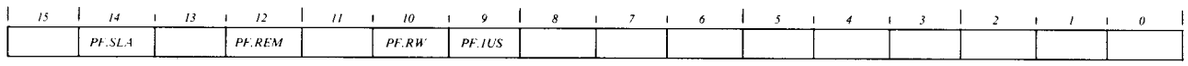


Offset	*Symbol*	*Description*
0	R.LINK	This word contains the address of the next LIB block in the list. If this entry is the last in the list, it will contain a 0.
2	R.NAME	These two words contain the resident library name, in RAD50.
6	L.PPN	This word contains the account number (PPN) of the resident library file. It is used when determining access privileges when a resident library is attached. The project number is in the high byte. The programmer number is in the low byte.
10	R.MCTL	These five words are the Memory Control Sub-block for the resident library (see section 2.3.1).
16	R.KSIZ	This byte (within the Memory Control Sub-block) specifies the size of the resident library, in K-words.
22	R.DATA	This byte is the FIP Unit Number for the disk containing the resident library. It is used when loading the resident library image and when closing the resident library file when the resident library is removed.
23		These three bytes contain the FIP Block Number (FBN) of the first block of the resident library image. When a resident library is loaded into memory it is accessed on disk by this block number. Byte 23 is the most significant byte of the block number.
26	R.FILE	This byte is the offset, in words, to the Name Entry for the LIB file within the directory block specified at offset 27.
27		These three bytes contain the FIP Block Number (FBN) of the block that contains the UFD name entry for this resident library. It is used to close the resident library file when the library is removed. Byte 27 is the most significant byte of the block number.
32	R.CNT	This byte contains a count of the number of jobs currently attached to this resident library.
33		This byte contains a count of the number of jobs using this resident library which are currently resident in memory. If the residency count is 0, the resident library is eligible for "swapping out". If a resident library is loaded with the /STAY switch, the high bit of this byte is set, ensuring that the residency count will never be 0 and the resident library will always remain in memory.

34	L.STAT	This byte is used to differentiate between an RTS block and a LIB block. If bit 7 (symbolically, LS.LIB) is set, this is a LIB block, otherwise it is an RTS block. LS.LIB is the only bit currently defined for L.STAT.
35	L.PROT	This byte is the library protection code. The protection code is used to control access to the memory space of a resident library. It is identical in usage to the file protection codes except that bits 6 and 7 have no meaning.
36	R.FLAG	This word contains a set of bits that describe the characteristics of the resident library (see section 2.3.5.1).

2.3.5.1 R.FLAG - Resident Library Characteristics

The library characteristic bits contained in R.FLAG in the LIB block have the following meaning when set:



Bit	*Symbol*	*Description*
<0:8>		Unused.
<9>	PF.1US	The resident library is single user, non-sharable.
<10>	PF.RW	The resident library may be mapped read/write if allowed by the protection code.
<11>		Unused.
<12>	PF.REM	Remove from memory when R.CNT becomes 0.
<13>		Unused.
<14>	PF.SLA	Load at specific address. This bit is always set for a resident library since libraries must always be loaded at a specific address.
<15>		Unused.

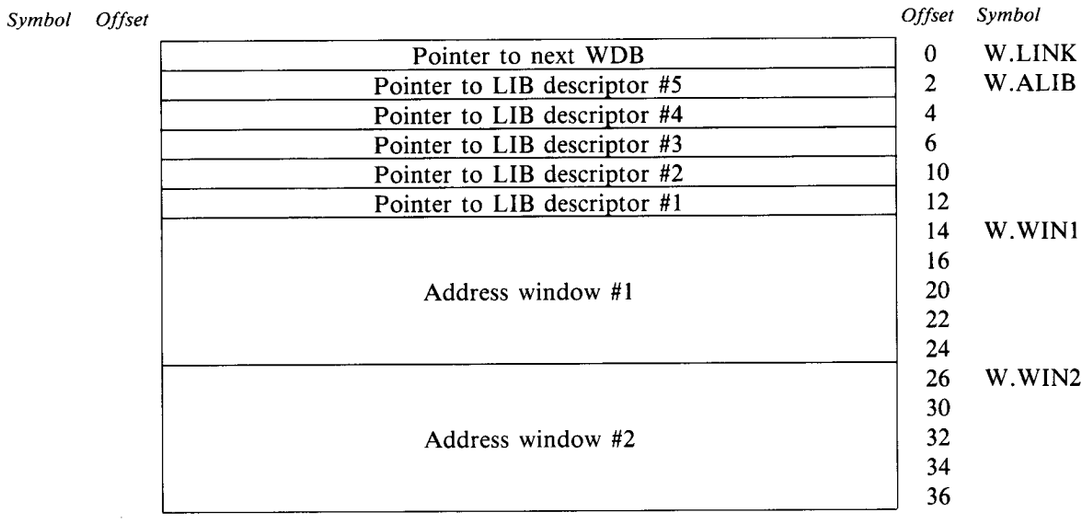
2.3.6 WDB - Window Descriptor Block

A job's memory space consists of the user low segment, the runtime system and up to five resident libraries, mapped by up to seven windows. If a job is not attached to any resident libraries, the job's memory requirements are totally described by the Memory Control Sub-blocks in its JDB and RTS blocks.

However, when a job attaches to one or more resident libraries, an additional control structure is needed to keep track of the extra memory windows. This structure is the Window Descriptor Block (WDB).

The WDB consists of up to three small buffers of information that describe up to seven memory windows and five resident libraries. Up to two windows and five resident libraries can be described with a single small buffer. An additional small buffer is required for each three additional windows.

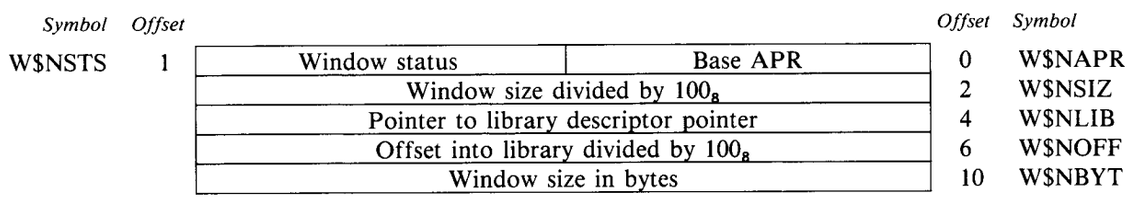
The first Window Descriptor Block has the following format:



Offset	*Symbol*	*Description*
0	W.LINK	If this job has more than two windows attached, this word contains a pointer to the second Window Descriptor Block, at offset W.ALIB. Otherwise, this word contains a 0.
2	W.ALIB	These five words contain pointers to the library descriptor blocks for up to five libraries. If less than five libraries are attached, the unused entries will be 0. The least significant 5 bits of each word may be used as flags by the monitor and should be cleared before using the pointer as an address. Note that the first LIB pointer is at offset 12. The remaining LIB pointers are at lower numbered offsets.
14	W.WIN1	These five words are the first address window (see section 2.3.6.1).
26	W.WIN2	These five words are the second address window (see section 2.3.6.1).

2.3.6.1 W.WIN? - Address Windows

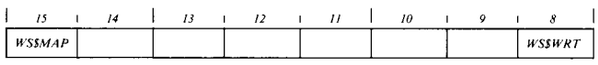
If a window is not in use, its first word will be 0. Address windows have the following format:



Offset	*Symbol*	*Description*
0	W$NAPR	This byte contains the number of the first APR used to map this window.
1	W$NSTS	This byte contains a bit pattern describing the status of the window (see section 2.3.6.1.1).
2	W$NSIZ	This word contains the desired window size, in bytes, divided by 100_8.
4	W$NLIB	This word is the address of the pointer (in W.ALIB) to the library descriptor block for the library associated with this window.
6	W$NOFF	This word is the byte offset into the library, divided by 64, to the beginning of this window.
10	W$NBYT	This word is the window size, in bytes.

2.3.6.1.1 W$NSTS - Window Status

The current status of each window is described by the bits contained in W$NSTS in each address window. These bits have the following meaning:



Bit	*Symbol*	*Description*
<8>	WS$WRT	Write access is desired.
<9:14>		Unused.
<15>	WS$MAP	The window is currently mapped.

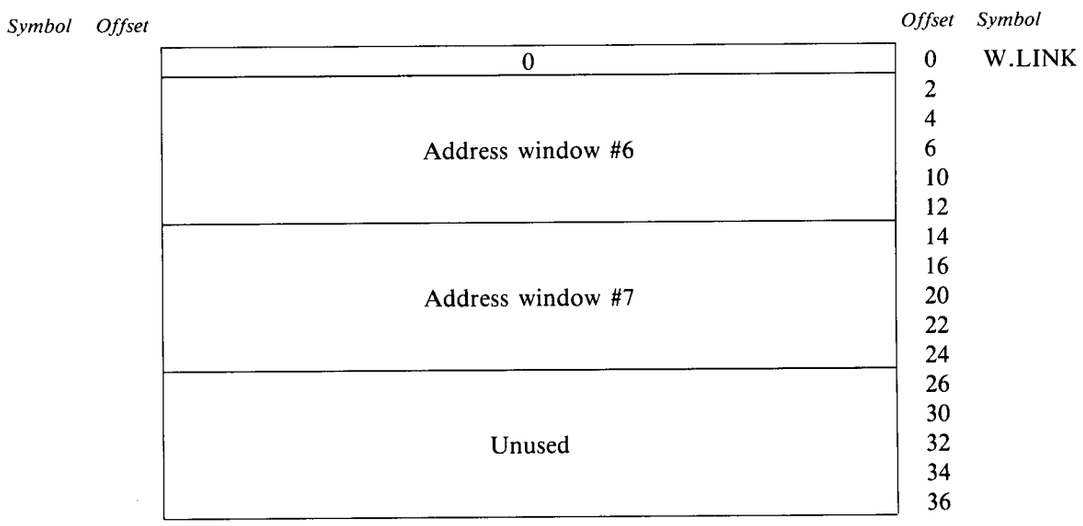
2.3.6.2 Extended Window Descriptor Blocks

If more than two windows are in use by a job, an additional WDB is allocated and linked to by the primary WDB. If more than five windows are in use, a third WDB is allocated and linked to from the second one.

The second WDB has the following format:

Symbol	*Offset*		*Offset*	*Symbol*
		Pointer to third WDB	0	W.LINK
		Address window #3	2 4 6 10 12	
		Address window #4	14 16 20 22 24	
		Address window #5	26 30 32 34 36	

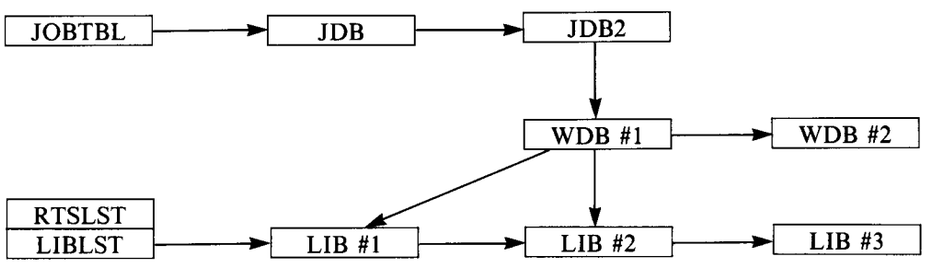
The third WDB has the following format:



The format of each address window is identical to that of the first WDB (see section 2.3.6.1).

2.3.6.3 Resident Library Linkages

All the information about a job's resident libraries can be found by following pointers starting at the job's Job Descriptor Block (JDB). The following figure illustrates the linkages used for resident library handling. In the illustration, the job is currently attached to two of the three libraries installed on the system. It has three windows mapped into these libraries.



2.4 FILE CONTROL

A disk is typically broken down into several files. Each file can be treated as if it were a separate disk. The monitor takes care of finding a place to store data for the file and then retrieving that data when needed.

Less disk overhead is required to open a file for the first user with the large file system control structures. No disk overhead is required for subsequent users of the same file. In addition, disk overhead associated with window turns (loading the retrieval window from the directory) is reduced, especially when the file is simultaneously open by more than one user.

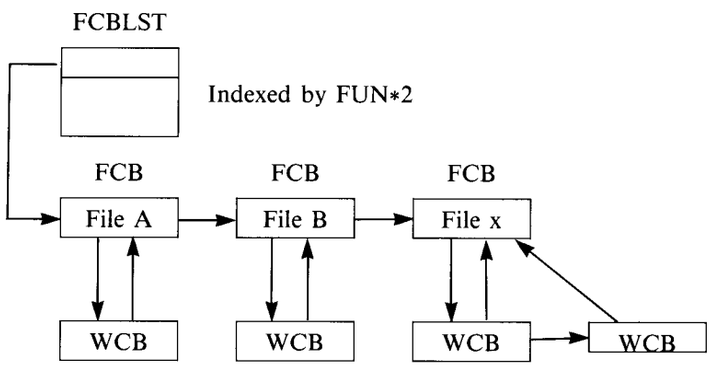
The structures used in the large file system are the File Control Block (FCB) and the Window Control Block (WCB). When a file is opened, the list of FCBs for the specified disk or disks is searched. If the desired file is found in the FCB list, a WCB is allocated for the new open request and linked to the existing FCB. If not, the directory is searched on disk to find the directory entry for the file. If the desired file is found in the directory, and FCB is allocated and initialized with information from the directory entry. A WCB is then allocated and linked to the FCB.

When the same file is opened simultaneously on another channel, or by another user, the information about the file is already in an FCB. No search of the directory, with its attendant disk overhead, is needed. All the information about the file is already contained in the existing FCB. A new WCB is allocated for the open request and is linked to the previous WCB or list of WCBs for this file. No disk accesses are required and FIP is released immediately.

As you can see, the overhead involved in opening a file is drastically reduced when the file is already open on another channel or by another job. This can be used to our advantage by writing a small program that opens the most commonly used files in read-regardless mode (mode 4096) and then detaches and goes to sleep indefinitely. Data files, commonly used programs and even UFDs can be opened in this way. Once opened, there will be essentially no overhead the next time any of these files are opened.

To allow the existing FCBs to be searched quickly, they are kept in a linked list. Each disk unit has its own linked list. A pointer to the first element of each list is kept in the FCBLST table. Each pointer in this table is accessed using the FIP Unit Number (FUN) times two of the associated disk unit as an offset from the beginning of the table.

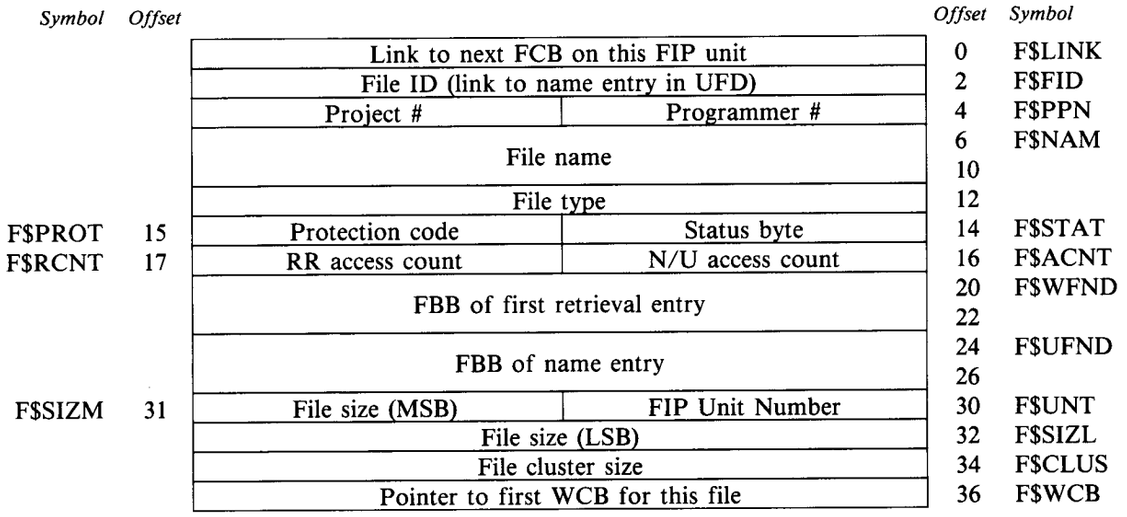
The Window Control Block (WCB) or blocks associated with each file are also kept in linked lists. Each file has its own list of WCBs. The pointer to the base of each list is kept in each file's FCB. The following diagram shows the lists of FCBs and WCBs:



2.4.1 File Control Block

The File Control Block (FCB) is used to store information about an open file. Each open file has an FCB associated with it. Only one FCB is required per file, regardless of the number of times the file is opened.

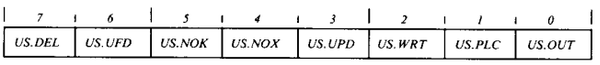
A File Control Block (FCB) has the following format:



Offset	*Symbol*	*Description*
0	F$LINK	This word contains a pointer to the next FCB on this FIP unit. The end of the list is terminated by a forward link of 0.
2	F$FID	This word contains the file ID for this file. It is used when searching the FCB list during an open by file ID (not available from BASIC-PLUS or BASIC+2).
4	F$PPN	This word contains the PPN of the file. The programmer number is in the low byte. The project number is in the high byte.
6	F$NAM	These three words contain the file name and type, in RAD50. The file name is in the first two words, the file type is in the third word.
14	F$STAT	This byte contains file status bits (see section 2.4.1.1).
15	F$PROT	This byte specifies the file's protection code.
16	F$ACNT	This byte contains a count of the number of times this file is currently open for normal or update access.
17	F$RCNT	This byte contains a count of the number of times this file is currently open for read-regardless access.
20	F$WFND	This double word is the FIP Block Sub-Block (FBB) of the first retrieval entry for this file in the directory. (FBBs are described in section 2.4.1.2.)
24	F$UFND	This double word is the FIP Block Sub-Block (FBB) of the name entry for this file in the directory. (FBBs are described in section 2.4.1.2.)
30	F$UNT	This byte specifies the FIP Unit Number for the disk unit containing this file.
31	F$SIZM	This byte contains the most significant byte of the file size.
32	F$SIZL	This word contains the least significant bytes of the file size.
34	F$CLUS	This word specifies the file's cluster size.
36	F$WCB	This word contains a pointer to the first Window Control Block (WCB) for this file.

2.4.1.1 F$STAT - Status Flags

The status flags contained in F$STAT have the following meaning when set:



Bit	*Symbol*	*Description*
<0>	US.OUT	This bit is obsolete and will never be set.
<1>	US.PLC	This file is placed starting at a specific disk block.
<2>	US.WRT	Write access has already been given out.
<3>	US.UPD	This file is open in update mode.
<4>	US.NOX	This file is contiguous. No extends are allowed.
<5>	US.NOK	This file may not be deleted or renamed.
<6>	US.UFD	This file is really a UFD opened in non-file structured mode (eg. OPEN "DB0:[1,2]" AS FILE 1).
<7>	US.DEL	This file has been deleted.

2.4.1.2 FBB - FIP Block Sub-Block

A FIP Block Sub-Block (FBB) provides all the information necessary to immediately access a specific block on a disk. An FBB has the following format:

Offset	*Symbol*	*Description*
0		This byte contains the FIP Unit Number of the desired disk.
1		This byte contains the most significant byte of the FIP Block Number for the desired block.
2		This word contains the least significant bytes of the FIP Block Number for the desired block.

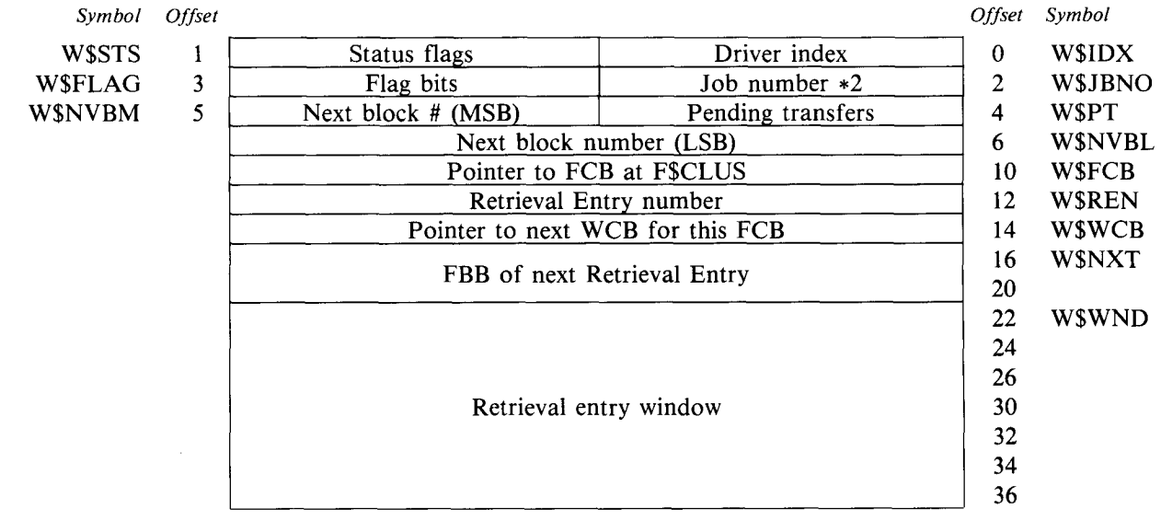
2.4.2 WCB - Window Control Block

The Window Control Block (WCB) contains all the information needed by the monitor for an individual user of a file. Each opening of a file generates a unique WCB. These WCBs are pointed to by the user's IOB (see section 2.1.4).

When an I/O request is made, the WCB associated with the specified user and channel number is used as the current retrieval window. If the desired retrieval information is not already present in the WCB, a" window turn" is performed. The large file system will start following Retrieval Entry links at the current position if the desired retrieval information is beyond the current position in the linked list of Retrieval Entries.

As the list of retrieval entries is followed, the linked list of WCBs for this file is checked. If the desired retrieval entry is already contained in a WCB, that WCB is used instead of following the link on disk. When the desired retrieval entry is found, its information is copied into the current WCB's retrieval entry window.

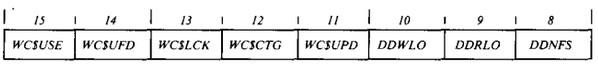
A Window Control Block has the following format:



Offset	*Symbol*	*Description*
0	W$IDX	This byte is the driver index. It is always 0 to show that the device associated with the WCB is a disk device.
1	W$STS	This byte contains the file status bits for this user (see section 2.4.2.1).
2	W$JBNO	This byte contains the job number, times 2, of the job that owns this WCB.
3	W$FLAG	This byte contains file status bits and locked block information (see section 2.4.3.2).
4	W$PT	This byte contains a count of the number of transfers pending on this file by this user. When an I/O is requested, this byte is set to 1. If the request requires more than one physical I/O transfer, this value is increased as necessary. An I/O request can require more than one transfer if it involves more than one block and the blocks either cross a cluster boundary or cross a cylinder boundary on a disk that doesn't do automatic cross-cylinder movement (such as an RL01/02).
5	W$NVBM	This byte contains the most significant byte of the next block number to use for sequential access.
6	W$NVBL	This word contains the least significant bytes of the next block number to use for sequential access.
10	W$FCB	This word contains a pointer to the File Control Block (FCB) associated with this WCB. It points to offset F$CLUS in the FCB.
12	W$REN	This word contains the current Retrieval Entry number. It identifies which Retrieval Entry is currently stored in W$WND of the WCB.
14	W$WCB	The low order bits of this word are file status bits. The high order bits are a pointer to the next WCB for this FCB. See section 2.4.2.3 for more information.
16	W$NXT	This double-word is the FIP Block Sub-Block (FBB) of the next Retrieval Entry in the directory. It allows immediate access to the next Retrieval Entry if sequential access is taking place.
22	W$WND	The following seven words are the current retrieval window. They contain the device cluster number of the first block of each cluster for this retrieval entry.

2.4.2.1 W$STS - Status Flags

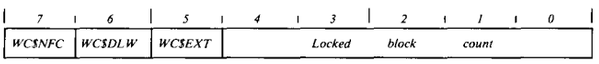
The status flags in W$WSTS show the status and restrictions placed on the file for the current user. Bear in mind that each time a file is opened a new WCB is allocated and each WCB has its own set of status flags. The bits in W$STS have the following meaning when set:



Bit	*Symbol*	*Description*
<8>	DDNFS	The disk is opened non-file structured. The remaining status bits apply to the entire disk as if it were opened file structured.
<9>	DDRLO	The file is read protected against the user.
<10>	DDWLO	The file is write protected against the user.
<11>	WC$UPD	The file is open for update (mode 1).
<12>	WC$CTG	The file is contiguous.
<13>	WC$LCK	The current block of the file is implicitly locked.
<14>	WC$UFD	The file is really a UFD opened in non-file structured mode (eg. OPEN "DB0:[1,2]" AS FILE 1).
<15>	WC$USE	This WCB received the original write privileges.

2.4.2.2 W$FLAG - Flag Bits

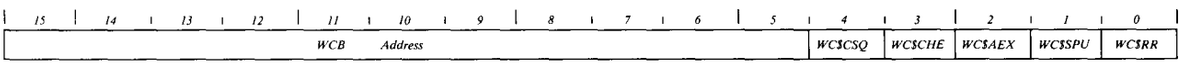
W$FLG serves two purposes. First, it defines the number of blocks that are included in an implicit lock. Second, it contains status bits similar in purpose to W$STS. The format of W$FLG is as follows:



Bit	*Symbol*	*Description*
<0:4>	WC$LLK	These five bits specify the number of blocks that are included in the current implicit block. This value may range from 0 to 31.
<5>	WC$EXT	This WCB is immediately followed in memory by another small buffer of information about explicitly locked blocks (see section 2.4.2.4).
<6>	WC$DLW	The file was either written into or has been extended. The file size and date last written need to be updated when the file is closed.
<7>	WC$NFC	The file is opened non-file structured in cluster mode (mode 128 was not used).

2.4.2.3 W$WCB - File Flags and Link to WCB

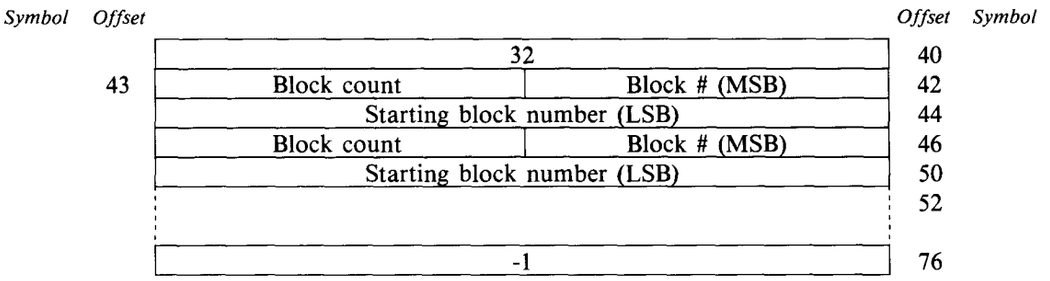
Since the small buffers from which WCBs are allocated are always allocated on even boundaries of 32 bytes, the low order five bits of their address are always 0. These low order bits are used as extended flag bits. They are not actually part of the address of the next WCB and should be treated as zeroes when used in a pointer. The format of W$WCB is as follows:



Bit	*Symbol*	*Description*
<0>	WC$RR	The file is open in read regardless mode (mode 4096).
<1>	WC$SPU	The file is open in special update mode (mode 5).
<2>	WC$AEX	Always do a real extend on the file (mode 8).
<3>	WC$CHE	The file is open for user data caching (mode 256 or 2048).
<4>	WC$CSQ	The user data caching specified by SC$CHE is sequential (mode 2048).
<5:15>		These bits are the address of the next WCB for this FCB, divided by 32. This pointer is used by clearing bits <0:4> of W$WCB and using the result as the address of the next WCB. If the resulting address is 0, this is the last WCB in the list.

2.4.2.4 Extended WCB

If WC$EXT is set in W$FLAG, the WCB is immediately followed in memory by an Extended WCB. The Extended WCB is used to specify blocks in a file which are explicitly locked (see the SPEC% function for disks in the RSTS/E Programming Manual). The first word in the Extended WCB is unused and will normally contain the value 32. The remainder of the extension contains seven double-words that each define a locked block range. The last word in the extension contains the value -1, which terminates the table of double-words.



Offset	*Symbol*	*Description*
40		This word is unused but will contain a value of 32.
42		This byte is the most significant byte of the starting block number of the locked block range (see offset 44).
43		This byte specifies the number of blocks that are locked by the explicit lock (1-31). If this byte is zero, this double-word entry is not in use. If negative, this is the end of the extended WCB.
44		This word is the least significant word of the starting block number of the locked range. It is combined with the byte at offset 42 to form a 24 bit block number.

2.5 DEVICE CONTROL

Device control breaks down into two parts: information needed by the monitor to perform logical operations (opens, closes, etc.) on the device and information needed by the device driver to interface with the device at the physical level. For more information on device drivers see Chapter 3.

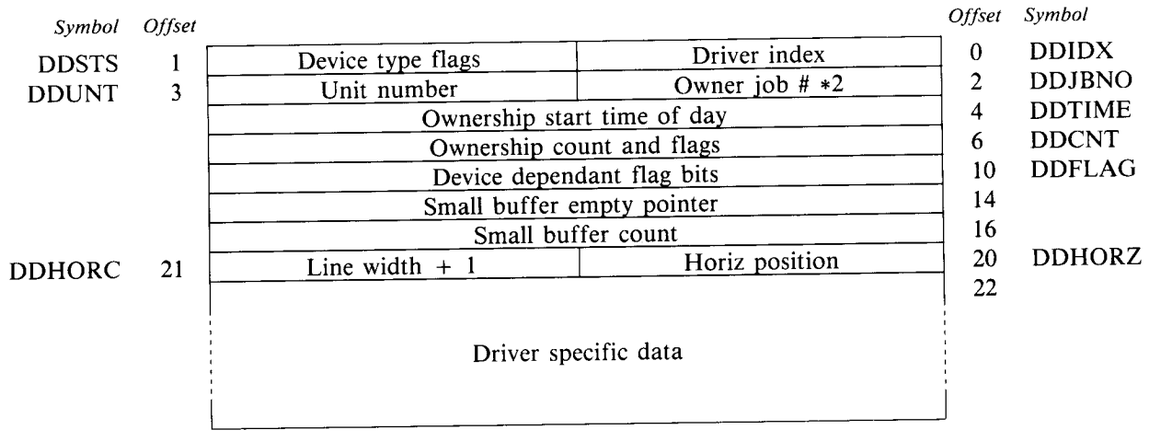
The logical operations available to a user are OPEN, CLOSE, GET, PUT, and SPEC. Except for opening a device, the only structures required are the user's XRB (see the *System Directives Manual*) and the Device Data Block (DDB) (see section 2.5.1). Opening a device requires the data structures DEVNAM, DEVCNT, DEVPTR, UNTCNT and DEVTBL, which are used to verify the name and unit standardized between all devices. Other portions are defined differently by different device drivers.

2.5.1 DDB - Device Data Block

The Device Data Block (DDB) is used to control physical use of a device by a driver and to communicate between monitor routines and the driver. A separate DDB is allocated for every unit of every device on the system.

A DDB contains information about device opens and assignment, small buffer usage and printer position, if appropriate. It also contains other information specific to the particular device, such as controller status and temporary buffering.

Every DDB must contain a certain amount of specific information. This minimal DDB has the following format (the information starting at offset 10_8 is only required if the driver uses small buffers):

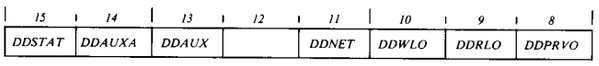


Offset	*Symbol*	*Description*
0	DDIDX	This byte contains the driver index. It is assigned to the device driver by the TBL assembly. This value is unique to each device driver (see section 3.3.14.)
1	DDSTS	This byte contains a set of bits that describe the characteristics of the device (see section 2.5.1.1).
2	DDJBNO	This byte is the job number, times 2, of the job that has this device unit assigned (either explicitly by an ASSIGN or implicitly by an OPEN). This byte is set to 1 by the DISABLE command in INIT and UTILITY. This effectively preassigns the device so that it cannot be assigned to any user. If DECNET owns this device, this byte will be set to 3 if the network services handler (NSP) owns the device or 5 if the network routing handler (TRN) owns the device.

3	DDUNT	This byte specifies the unit number of this device.
4	DDTIME	This word contains the system time at the time the device was assigned. It is used to charge device time when the device is deassigned.
6	DDCNT	This byte is the ownership count for this device unit. It is incremented each time the device is opened and decremented when it is closed.
7		This byte is a set of device flags (see section 2.5.1.2).
10	DDFLAG	This word contains device specific flag bits and is not used by the monitor.
12	DDBUFC	The following three words are the Small Buffer Control Block (see section 2.5.1.3). The first word (DDBUFC + EP) is the pointer to where bytes are to be removed from the small buffer chain. The second word (DDBUFC + FP) is the pointer to where bytes are to be inserted into the small buffer chain. The third word (DDBUFC + BC) is the count of small buffers remaining in the allocation for this device. When this word reaches 0 or becomes negative, the program may be stalled until more small buffers become available.
20	DDHORZ	This byte specifies the current horizontal position of the "print head". A value of 0 is the right hand margin. A value equal to DDHORC is the left hand margin.
21	DDHORC	This byte specifies the device's line width, plus one.

2.5.1.1 DDSTS - Device Characteristics Flags

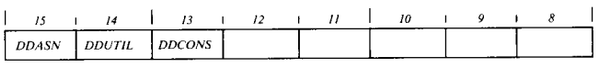
The device characteristics bits in DDSTS have the following meaning when set:



Bit	*Symbol*	*Description*
<8>	DDPRVO	Ownership of this device unit requires privileges.
<9>	DDRLO	Read privileges are never given for this device. This device is inherently write only.
<10>	DDWLO	Write privileges are never given for this device. This device is inherently read only.
<11>	DDNET	This device exists elsewhere in a DECNET network.
<12>		Unused.
<13>	DDAUX	This device can use KMC11 bridge blocks. Bridge blocks provide a means of communication between the computer and a KMC11 processor.
<14>	DDAUXA	This device is currently bridged to a KMC11.
<15>	DDSTAT	This bit is not used by the monitor but is cleared each time the device is closed. The driver can assign this bit to a condition that should be automatically reset each time the device is closed.

2.5.1.2 DDCNT - Device Flags

The device flag bits in DDCNT have the following meaning when set:



Bit	*Symbol*	*Description*
<8:12>		Unused.
<13>	DDCONS	This device is its owner's console terminal.
<14>	DDUTIL	This device is temporarily assigned to FIP.
<15>	DDASN	This device is explicitly assigned to the job shown in DDJBNO.

2.5.1.3 Small Buffer Control Block

Device drivers that do not transfer data directly to and from a user's buffer using DMA transfers store data in small or large buffers within the monitor while it is being transferred. Use of small buffers requires a Small Buffer Control Block to control insertion and removal of characters in a chain of small buffers.

The small buffers are chained together in a linked list using the first word of each small buffer as a link. Characters are inserted at the end of the chain and removed from the beginning of the chain. If an attempt is made to insert a character and the last small buffer in the chain is totally filled, an additional small buffer is allocated and linked onto the end of the list. If the last character in a small buffer is removed, the small buffer will be removed from the list and returned to the small buffer pool.

A device driver can use any number of small buffer chains. Each of these chains requires a Small Buffer Control Block within each DDB. The Small Buffer Control Block consists of a fill pointer, an empty pointer and a buffer count. The Small Buffer Control Block has the following format:

Symbol *Offset*		*Offset*	*Symbol*
	Empty pointer	0	EP
	Fill pointer	2	FP
	Remaining buffer quota	4	BC

Offset	*Symbol*	*Description*
0	EP	This word is the "empty pointer." It is a pointer to the next byte to be removed from the small buffer chain. If bits <0:4> are zero when a request is made to retrieve the next byte from the chain, this pointer will be adjusted to point to the first byte in the small buffer in the list before retrieving the byte. The current small buffer will be returned to the small buffer pool.
2	FP	This word is the "fill pointer." It is a pointer to the location where the next byte should be stored in the small buffer chain. If bits <0:4> are zero when a request is made to store a character in the chain, a new small buffer will be linked onto the end of the chain and the pointer will be adjusted to store the character in the first byte of the new small buffer.
4	BC	This word is the small buffer count. It specifies the remaining small buffer quota for this chain. It is initially set to the value of the associated small buffer quota (see section 3.3.8). It is decremented each time a small buffer is added to the chain and incremented each time one is removed. If the remaining buffer quota becomes negative, a request to add a small buffer to the list may fail due to a lack of small buffers on the system (see 3.5.1).

2.5.2 DSQ - Disk I/O Queue Entry

When a disk I/O is requested, a small buffer is allocated for use as a Disk Queue Entry (DSQ). This entry is added to a linked list of DSQs waiting for I/O on the desired disk and unit. The base of each linked list is pointed to by an entry in the DQS$XX table.

The requested I/O will be performed immediately if there are no entries already in the queue for this disk unit. Otherwise, the request will remain in the queue and will be performed at a later time.

Before a physical I/O is started on a disk unit, the queue of DSQs is reordered to minimize head movement. The entry that requires the least head movement from its current location will be performed next, unless circumvented by an expired fairness count.

Each time the queue is reordered, the fairness count in each DSQ is decremented. If an entry's fairness count becomes zero, the entry will be moved to the head of the queue and will be processed immediately, regardless of the amount of head movement required. Disk I/O requests for swapping and directory lookups are issued with a fairness count that is already zero to circumvent normal optimization.

Information in a DSQ is initialized and updated at several different times. Information that can be derived from the specifications in the XRB is loaded into the DSQ immediately. This information includes job number, retry count, block number, buffer address, function code and transfer counts. Information about physical disk addresses and optimization is entered by the disk driver before the queue is optimized. The remaining information is entered by the disk driver and common disk code during I/O processing.

A DSQ has the following format:

Symbol	*Offset*			*Offset*	*Symbol*
		Queue link word		0	
DSQERR	3	Retry count/Error	Job # *2	2	DSQJOB
		Pointer to L3Q bits to set		4	DSQL3Q
DSQFBM	7	FIP Block # (MSB)	FIP Unit Number	6	DSQUNT
		FIP Block Number (LSB)		10	DSQFBL
DSQMAM	13	Buffer addr (MSB)	RH-11 function	12	DSQRFN
		Buffer address (LSB)		14	DSQMAL
		Transfer word count		16	DSQCNT
DSQFAR	21	Fairness count	Function code	20	DSQFUN
		Miscellaneous pointer		22	DSQMSC
		Total transfer word count		24	DSQTOT
		Physical disk address		26	DSQPDA
		Optimization word		30	DSQOPT
DSQSAV	33	Saved function	Unit number *2	32	DSQOUN
		Offset pointer		34	DSQPTO
DSQPUN	37	Unit number	Offset retry cntr	36	DSQCTO

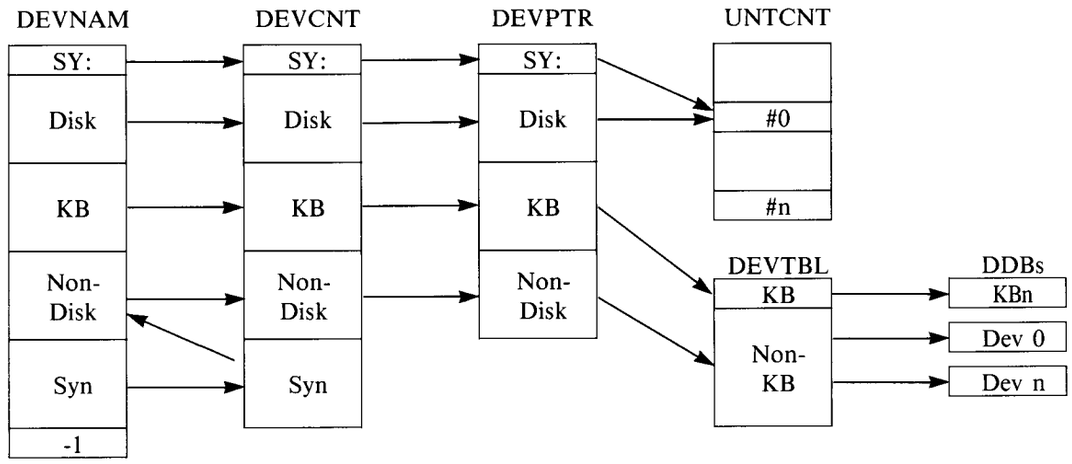
Offset	*Symbol*	*Description*
0		This word contains a pointer to the next DSQ in this list. If this is the last entry in the list, this word will contain a zero.
2	DSQJOB	This byte specifies the job number, times 2, of the job waiting on the I/O.
3	DSQERR	This byte contains the retry count and error flag. If this byte is negative, it is the remaining entry count. If it is zero, an unrecoverable user data error occurred. If it is positive, it is the error code to return to the caller. The retry count is initially set to -9.
4	DSQL3Q	This word points to a word that specifies the L3Q bits to set on I/O completion. The word pointed to also specifies which completion queue to put this DSQ in when the I/O completes.

6	DSQUNT	This byte specifies the FIP Unit Number.
7	DSQFBM	This byte specifies the most significant byte of the desired FIP Block Number.
10	DSQFBL	This word specifies the least significant word of the desired FIP Block Number.
12	DSQRFN	This byte specifies the RH-11 function code for the desired function. If the device does not use the RH-11, the driver will clear this byte.
13	DSQMAM	This byte specifies the most significant byte of the 22-bit address of the user's I/O buffer.
14	DSQMAL	This word specifies the least significant bytes of the 22-bit address of the user's I/O buffer.
16	DSQCNT	This word specifies the word count of the requested transfer. It may be less than the word count requested by the user if the I/O request must be broken into several separate transfers. The driver may negate this value if the controller requires a negative word count.
20	DSQFUN	This byte specifies the function to be performed by the disk driver. The valid function codes are: WFUN.C 0 Write data with write check WFUN 2 Write data RFUN 4 Read data FRUN.C 6 Write check
21	DSQFAR	This byte contains the fairness count. This value is initially set either to ..FCNT (which is normally 6) or 0 (to circumvent optimization during swapping and FIP accesses). It is decremented each time the queue is optimized. If the fairness count becomes zero, this DSQ is used as the most optimal, regardless of the head movement required.
22	DSQMSC	This word is used to contain miscellaneous information. It commonly contains a pointer to the associated FCB, WCB or SCB.
24	DSQTOT	This word specifies the total number of blocks to be transfered. It can be different than DSQCNT/256 if the request has been broken into several sub-transfer requests. This is done if the request crosses cluster boundaries, or if it crosses a cylinder boundary on a disk that does not do automatic cross-cylinder movement.
26	DSQPDA	This word contains the physical disk address of the requested data. It normally contains the track and sector address in a format specified by the driver.
30	DSQOPT	This word contains the head movement optimization word. It normally contains the cylinder address of the requested disk block.
32	DSQOUN	This byte contains the physical disk unit number times two.
33	DSQSAV	This byte can be used by the device driver to save the requested function code.
34	DSQPTO	This word is a pointer to the entry in the offset table for this disk type to use when automatic head offsetting is required. Each offset table contains a list of head offset specifications, in micro-inches, and is used for correction of minor alignment errors.
36	DSQCTO	This byte specifies the number of retries remaining at the current head offset specification.
37	DSQPUN	This byte contains the physical disk unit number.

2.5.3 Logical Device Tables

Verifying a device name and unit number and associating the logical name with a DDB (when a device is opened) or a WCB or SCB (when a file is opened) is performed using a series of related tables. The entries in each table are ordered so that once an entry is found in one table, the associated information in the other tables will be found at the same offset.

The following figure shows the relationship between the device tables:



2.5.3.1 DEVNAM - Device Name Table

The Device Name Table (DEVNAM) contains an entry for every disk type supported by RSTS/E and an entry for every non-disk device generated into this system. It also contains a list of synonyms for physical device names, such as MT: for MM:

All of the entries in DEVNAM consist of a pair of ASCII characters, stored in a word. Each disk name and the name of each configured device is included. DEVNAM has the following format:

Offset	*Symbol*	*Description*
-2		This word always contains the ASCII value "SY". It is used for specifying the system disk structure, SY:. It is at an offset of -2 to show that it is not a normal device.
0-16	DEVNAM	These nine words contain the names of all the disks supported by RSTS/E, in the order: DC, DF, DS, DK, DL, DM, DP, DR, DB, DU. All disk names are included, even if the device is not configured in this system.
24	DEVNKB	This word contains the device name "KB". It corresponds to all keyboard type devices, including pseudo-keyboards.
26		This word contains the device name "NL". H corresponds to the null device.
30		The device names for all remaining devices configured into this system begin at this word.
+n	DEVSYN	Beginning at this word is a list of synonyms for the devices configured into this system. These synonyms are: TT=KB, CR=CD, MT=MM, MT=MS, DX=DY, plus synonyms for user written device drivers.
+n+m		This word contains a -1 to show the end of the table.

2.5.3.2 DEVCNT - Device Unit Count Table

The Device Unit Count Table (DEVCNT) contains one entry for each entry in DEVNAM (except the terminating -1). Each entry corresponds to either the maximum unit number for the corresponding device in DEVNAM or a pointer to a synonym.

DEVCNT has the following format:

Offset	*Symbol*	*Description*
-2		This word contains a 0 to specify that unit 0 is the only explicit unit number allowed for SY:.
0	DEVCNT	The ten words starting here correspond to the maximum unit number for each corresponding disk type in DEVNAM. A value of -1 indicates that the corresponding disk type does not exist on this system.
24		This word contains the maximum unit number for KB type devices, including pseudo-keyboards.
26		This word contains a 0 to specify that only unit 0 may be specified for NL:.
30		Each following word corresponds to the maximum unit number for the corresponding device in DEVNAM.
+n		Each word beginning at an offset equivalent to DEVSYN is a pointer to the appropriate physical name in DEVNAM for each synonym.

2.5.3.3 DEVPTR - Device Information Pointer Table

The Device Information Pointer Table (DEVPTR) contains pointers into one of two other tables. In the case of disk devices, DEVPTR points into the Disk Unit Status Table, UNTCNT. For non-disk devices, DEVPTR points into the Device Retrieval Table, DEVTBL.

In both cases, the entry pointed to is the information for unit zero of the associated device. If more than one unit is present for a device, the entries for the additional units immediately follow the unit 0 entry. See section 2.5.3.4 and 2.5.3.5 for more information on UNTCNT and DEVTBL.

DEVPTR has the following format:

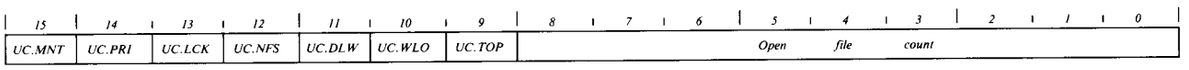
Offset	*Symbol*	*Description*
-2		This word contains a pointer to the entry in UNTCNT corresponding to SY0:. This unit is the device from which RSTS/E was bootstrapped.
0	DEVPTR	The ten words beginning here contain pointers to the entries in UNTCNT for unit 0 of the corresponding disk type. If a disk type does not exist on this system, its pointer is to the entry in UNTCNT for the next valid disk type. If no following disk types are valid, this word points to a dummy entry at the end of UNTCNT.
24		The words beginning here contain pointers to the entries in DEVTBL corresponding to unit 0 of each non-disk in DEVNAM.

2.5.3.4 UNTCNT - Disk Unit Status Table

The Disk Unit Status Table (UNTCNT) contains information about each disk unit configured in this system. There is one word for each disk type and unit. These words are ordered by disk type in the same order as the disk names in DEVNAM and, within disk type, by unit number. Each word contains the status and current open count for the corresponding disk unit.

The last entry in the table is followed by a word containing 177001_8 which denotes a non-mounted disk which cannot be mounted. This is used for proper termination of DEVPTR entries for non-existent disk types (see section 2.5.3.3).

Each word of UNTCNT has the following format:



Bit	*Symbol*	*Description*
<0:8>	UC.CNT	Count of currently open files.
<9>	UC.TOP	Order the directory with new files first.
<10>	UC.WLO	The disk is mounted read only.
<11>	UC.DLW	Update access date with date last written.
<12>	UC.NFS	Non-file structured processing is currently occurring.
<13>	UC.LCK	Don't allow non-privileged opens on this disk.
<14>	UC.PRI	This disk unit has a private pack mounted.
<15>	UC.MNT	This disk unit does not have a pack mounted. Bits 9-14 are meaningless.

2.5.3.5 DEVTBL - DDB Pointer Table

The DDB Pointer Table (DEVTBL) contains a pointer to the DDB for each unit of each non-disk device on the system. The entries are ordered in the same order as DEVNAM. If more than one unit is present for a device, the entries for the additional units immediately follow the unit 0 entry.

DEVTBL has the following format:

Offset	*Symbol*	*Description*
0	DEVTBL	The words beginning here are pointers to the DDBs for the each terminal device and the pseudo-keyboard unit, beginning with KB0:.
+n	DEVTBE	The words beginning here are pointers to the DDBs for each unit of the non-terminal devices, beginning with NL0:.

2.5.3.6 LOGNAM - Logical Name Table

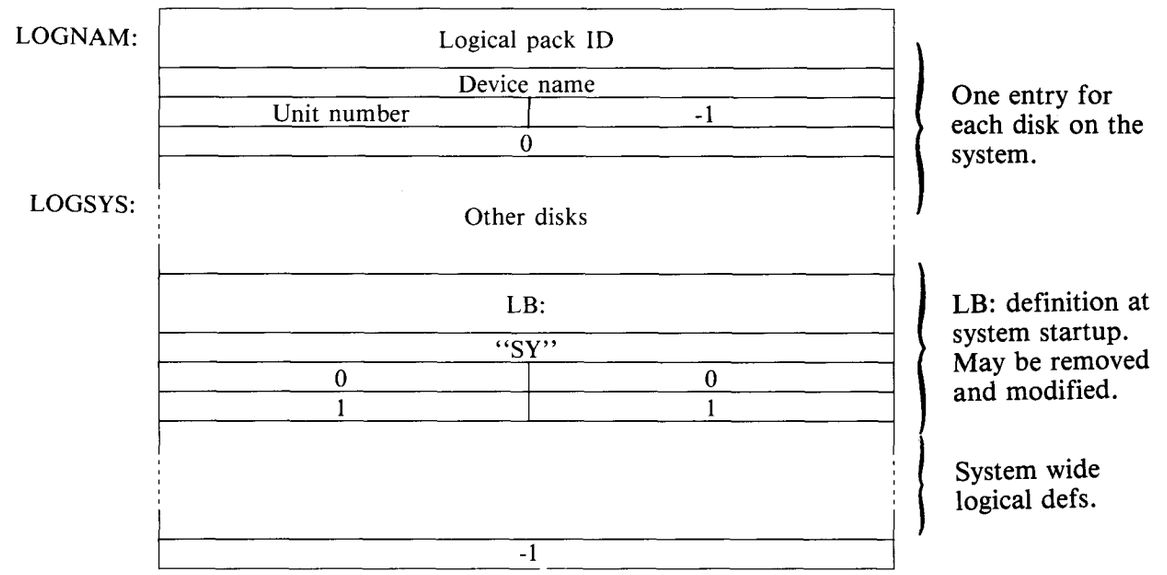
The Logical Name Table (LOGNAM) is used to translate system wide logical device assignments into physical device names and account numbers. System wide logicals include the pack ID for all mounted disks, the logical LB: and any additional logical name assignments defined using the ADD LOGICAL command in UTILITY or the associated SYS call.

The size of LOGNAM depends on the number of disks present on the system and the number of system wide logicals declared during system generation. Each system wide logical requires a five word entry in the LOGNAM table. Each entry has the following format:

Symbol *Offset*		*Offset* *Symbol*
Logical name (in RAD50)		0 2
Device name (in ASCII)		4
Unit "real" flag	Unit number	6
PPN		10

Offset	*Symbol*	*Description*
0		For user defined system logicals, these two words contain the logical device name, in RAD50. For disks, these two words contain the logical pack ID, in RAD50, for the associated disk unit. If the disk is not mounted, these words will be zero.
4		This word contains the physical device name that corresponds to the logical device name.
6		This word contains the device unit number and the unit number "real" flag. (See the description of the FIRQB in the RSTS/E System Directives Manual for more information on device naming.)
10		If this entry is for a disk, this word will contain the asscitated account number, if any, for the logical name. If not, this this word will be zero.

The first entries in LOGNAM correspond to the logical name of each disk unit on the system. The following entries correspond to system wide logical names for other devices. The first entry is initialized with the definition for LB: at system startup. The remaining entries are available for user defines system wide logicals. The last entry is followed by a -1, which terminates the table.



2.5.4 Device Driver Dispatch Tables

Device drivers are divided into several specific routines. These routines are used to perform the device specific operations necessary when a request is made of the driver (see Chapter 3).

The monitor has a group of tables that contain pointers to the entry points of each driver. Each table contains a one word entry for every non-disk device on the system. The entries are indexed by the device index, which is a unique number assigned to each device driver by the TBL assembly.

The device driver dispatch tables and their associated entry points are as follows:

Table Name	*Entry Point*	*Description*
ASNTBL	ASN$xx	Assign
DEATBL	DEA$xx	Deassign
OPNTBL	OPN$xx	Open
CLSTBL	CLS$xx	Close
SERTBL	SER$xx	I/O service (.READ/.WRITE)
SPCTBL	SPC$xx	Special function (.SPEC)
SLPTBL	SLP$xx	Check before sleeping. If the device does not need to be notified of a sleep, its entry will be zero.
TMOTBL	TMO$xx	Timeout
ERRTBL	ERL$xx	Error logging
UMRKB	UMR$xx	Unibus mapping register is available. If the device does not use unibus mapping registers, its entry will be zero.

2.5.5 Device Driver Support Tables

Most of the work involved in assigning or opening a device is done by common code in the monitor. This code checks that the user has the proper privileges to assign the device and that the device is not already assigned to someone else. It then updates the user's IOB with a pointer to the device's DDB or WCB and updates the DDB to show the device as being assigned. Once all this housekeeping has been performed, it calls the driver to perform its own device specific initialization, if any.

In order to provide this generalized handling of devices, each driver defines information that is stored in a group of monitor tables. In most cases, the information for a particular driver can be found within a table using the driver index as an offset into the table.

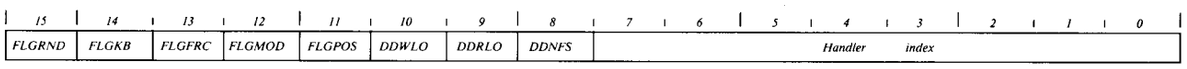
Two exceptions to this are CSRTBL and TIMTBL. These tables contain information for every unit of every device. The first entry for each device is defined by a global symbol of the form CSR.xx and TIM.xx, respectively, where "xx" is the device name. Information about the remaining units is accessed using the unit number times two as an offset from the first entry.

2.5.5.1 FLGTBL - Device Characteristics Flags

Each entry in FLGTBL contains the handler index and a set of bits that describe the characteristics of the associated device. These bits are returned in FQFLAG of the FIRQB (the STATUS variable in BASIC-PLUS) on an open call. Each entry in FLGTBL is accessed using the device index (IDX.xx) as an offset.

The characteristics flag bits describe the general characteristics of each device. For example, a lineprinter would have the DDRLO bit set to signify that input requests are not allowed.

The format of each word of FLGTBL is a follows:



Bit	*Symbol*	*Description*
<0:7>		This byte is the handler index for the associated device. Each device supported by RSTS/E is assigned a unique handler index (see section 3.3.2). The RSTS/E Programming Manual lists the standard handler indices.
<8>	DDNFS	This device does not require a file name. If a file name is supplied, it will be ignored.
<9>	DDRLO	This device is generically read locked. It cannot do input.
<10>	DDWLO	This device is generically write locked. It cannot do output.
<11>	FLGPOS	This device keeps track of its own horizontal position and expands tabs automatically. The current horizontal position can be found in the DDHORZ byte of the DDB for this device and unit.
<12>	FLGMOD	This device used modifiers passed in location XRMOD of the XRB. The modifiers are specified in BASIC-PLUS using the RECORD clause.
<13>	FLGFRC	This device is byte oriented and does not require a specific number of characters to be transferred per I/O request.
<14>	FLGKB	The device is a terminal. This bit in only used by the terminal driver.
<15>	FLGRND	The device accepts record numbers on GET and PUT (ie. the device is generically a random access device).

2.5.5.2 SIZTBL - Line Width

Each word of SIZTBL defines the line width of the associated device. The entries in SIZTBL are accessed using the device index (IDX.xx) as an offset. Each entry has one of three possible values:

5*14+1	signifies that line width does not apply to this device.
width+1	signifies that line width applies and is permanently fixed at the specified width.
0	signifies that line width applies and is variable. The real line width will be found in the DDHORC byte of the DDB.

2.5.5.3 BUFTBL - Default I/O Buffer Size

Each word of BUFTBL defines the default buffer size for the associated device. This value is returned in FQBUFL of the FIRQB on a device/file open request. Each entry in BUFTBL is accessed using the device index (IDX.xx) as an offset.

The user can generally use any buffer size desired when processing I/O. The default buffer size is only used as a suggested size in case the user does not specify a buffer size on a file open. BASIC-PLUS uses the suggested buffer size unless overridden by the RECORDSIZE option.

2.5.5.4 JSBTBL - JS.xx Bit for Device

Each word of JSBTBL specifies the bit in JBWAIT to be set when an I/O service request is initiated for the associated device. If this bit is still set when the driver returns to the monitor, the calling job will be stalled until the driver exits its I/O service routine by jumping to IOEXIT (see section 3.2.5). Each entry in JSBTBL is accessed using the device index (IDX.xx) as an offset.

2.5.5.5 DVRAP5 - APR5 Pointers

Each word of DVRAP5 contains a pointer to the value to be loaded into memory management register APR5 to map the associated device driver code. This value will be loaded into APR5 before the monitor enters any of the driver's entry points. Each entry in DVRAP5 is accessed using the device index (IDX.xx) as an offset.

2.5.5.6 CSRTBL - CSR Addresses

CSRTBL contains the address of the most commonly used register for each unit of each device on the system. For most devices, CSRTBL contains the address of the CSR register for the associated device.

Since a separate entry is provided for each unit of each device, this table cannot be accessed by device index. Instead, a global symbol of the form CSR.xx (where "xx" is the device name) is equated to the entry within CSRTBL for unit 0 of the associated device. The CSR address for any remaining units of a device can be found using the unit number times two as an offset from the entry for unit 0.

CSRTBL contains entries for user-written device drivers as well as standard devices. The entries for user-written device drivers will be zero, unless patched to a CSR address. The driver cannot set this value during execution because CSRTBL may be in the read-only portion of the monitor.

2.5.5.7 TIMTBL - Timeout Counters

TIMTBL is a table of words that contain timeout counters for each unit of each device on the system. When a device driver stalls a user program while waiting for an interrupt, it sets the associated entry in TIMTBL to some non-zero value. The monitor will enter the driver's timeout entry point if the time specified in TIMTBL expires before the device interrupts. This protects against having a job permanently stalled if an interrupt is lost or in case of hardware failure.

Each entry in TIMTBL can have one of three possible values:

0	No timeout is in progress. No action is required of the monitor.
>0	A timeout is in progress. Decrement the timer value each second and enter the driver's timeout entry point if the value decrements to zero.
<0	Enter the driver's timeout entry point every second.

Since a separate entry is provided for each unit of each device, this table cannot be accessed by device index. Instead, a global symbol of the form TIM.xx (where "xx" is the device name) is equated to the entry within TIMTBL for unit 0 of the associated device. The timeout counter for any remaining units of a device can be found using the unit number times two as an offset from the entry for unit 0.

2.5.6 Disk Control Tables

There are several monitor tables that contain information used for performing disk I/O. Some of these tables provide information needed by the drivers and the common I/O routines. Others provide information used by FIP to find directory information and to convert logical block numbers to FIP Block Numbers.

The disk control tables contain entries for each disk unit on the system. They are indexed by FIP Unit Number. A unique FIP Unit Number (FUN) is assigned to each disk unit at sysgen time. The FUNs start at zero and increase by two for each unit of the system.

2.5.6.1 DEVCLU/CLUFAC - Device Cluster Size, Cluster Ratio

The low byte of each entry in the combined DEVCLU/CLUFAC table specifies the device cluster size for the associated disk unit. The high byte of each entry in DEVCLU/CLUFAC specifies the cluster ratio for the associated disk unit.

This cluster ratio is the ratio of the pack cluster size to the device cluster size. It is calculated by dividing the pack cluster size by the device cluster size. It will always be at least one and a power of two.

2.5.6.2 UNTCLU/UNTERR - Pack Cluster Size, Error Count

The low byte of each entry in the combined UNTCLU/UNTERR table specifies the pack cluster size of the pack mounted on the associated unit. If no pack is mounted, this word will be zero.

The high byte of each entry in UNTCLU/UNTERR contains a count of the number of errors that have occurred on the associated disk pack since it was mounted.

The entries in the low bytes are accessed using the FIP Unit Number times two as an offset from the symbol UNTCLU. The entries in the high bytes are accessed using the FIP Unit Number times two as an offset from the symbol UNTERR.

2.5.6.3 UNTLVL/UNTREV - Disk Unit RDS Level/Revision

Each word in the combined UNTLVL/UNTREV table specifies the RSTS Disk Structure (RDS) and revision level for the corresponding disk unit.

The low byte of each word is the level number. This byte is 0 for RDS0 level disks and 1 for RDS1 level disks. UNTLVL entries are accessed using the FIP Unit Number (FUN) times two as an offset from the symbol UNTLVL.

The high byte of each word is the RDS revision level. This byte 0 for RDS0 level disks and 1 for RDS1 level disks. UNTREV entries are accessed using the FIP Unit Number (FUN) times two as an offset from the symbol UNTREV.

2.5.6.4 UNTSIZ - Disk Size

Each word in UNTSIZ specifies the size of the associated disk unit, in device clusters. This value is initialized by INIT at system startup time.

2.5.6.5 UNTLIB - [1,2] Starting Cluster

Each word in UNTLIB contains the device cluster number (DCN) of the first device cluster used by the [1,2] UFD on the associated disk pack. This value is initialized when the pack is mounted.

2.5.6.6 MFDPTR - MFD Pointers

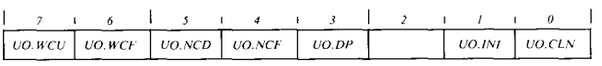
Each word of MFDPTR specifies the device cluster number (DCN) of the first block of the MFD for the associated disk pack. This value is initialized when the pack is mounted. It is used to access the MFD without requiring it to be at a fixed location at the beginning of the disk pack. The entries in MFDPTR are accessed using the FIP Unit Number (FUN) times two as an offset from the symbol MFDPTR.

2.5.6.7 UNTOWN/UNTOPT - Unit Owner, Unit Options

The low byte of each entry in the combined UNTOWN/UNTOPT table specifies the job number, times two, of the job,if any, that owns the associated disk pack. The high byte of each entry in UNTOWN/UNTOPT specifies the current option settings for the associated disk unit. It is similar in purpose to the UNTCNT table.

The entries in the low bytes are accessed using the FIP Unit Number times two as an offset from the symbol UNTOWN. The entries in the high bytes are accessed using the FIP Unit Number times two as an offset from the symbol UNTOPT.

The bits in UNTOPT have the following meaning when set:



Bit	*Symbol*	*Description*
<0>	UO.CLN	The disk needs cleaning.
<1>	UO.INI	The disk is being initialized.
<2>		Unused.
<3>	UO.DP	The disk is dual ported.
<4>	UO.NCF	Don't cache directory information for this unit.
<5>	UO.NCD	Don't cache data files for this unit.
<6>	UO.WCF	Write check all FIP writes.
<7>	UO.WCU	Write check all writes.

2.5.6.8 DSKMAP - FUN to Disk Index

Each word of DSKMAP contains the Disk Type Index of the corresponding FIP Unit Number. Each disk type on the system has a unique Disk Type Index. This index identifies which driver to use for device specific functions. This index is not to be confused with the driver index specified by IDX.XX which is zero for all disk types.

2.5.7 SAT Tables

Every file structured disk pack contains a Storage Allocation Table (SAT). The SAT maps all the device clusters available on the disk and shows which clusters are in use and which are available for allocation (see section 1.3).

The monitor uses several tables to store information about the SAT for each disk unit on the system. These tables provide information such as disk size, number of free blocks, last allocated cluster and the location of the SAT on each disk. Access to the information in each table is made using the FIP Unit Number times two as an offset from the beginning of the table.

2.5.7.1 SATCTL, SATCTM - Count of Free Blocks

The SATCTL and SATCTM tables specify the number of free blocks on the corresponding disk pack. The entry in SATCTM contains the most significant word of the block count. The entry in SATCTL contains the least significant word of the block count.

2.5.7.2 SATPTR - DCN of Last Allocated Cluster

The SATPTR table specifies the device cluster number (DCN) of the last cluster allocated on the corresponding disk pack. When a request to allocate a new cluster is received, the search for a free cluster begins at this device cluster. If a free cluster is not found by the end of the SAT, the SAT will be searched from the beginning.

2.5.7.3 SATEND - Ending PCN

The SATEND table specifies the pack cluster number (PCN) of the last cluster on the corresponding disk pack.

2.5.7.4 SATSTL, SATSTM - Starting FBN of SATT.SYS

The SATSTL and SATSTM tables specify the FIP Block Number (FBN) where SATT.SYS begins on the corresponding disk. The entry in SATSTM contains the most significant word of the FBN. The entry in SATSTL contains the least significant word of the FBN.

2.5.8 DECNET Device Control Tables

Version 7.1 of RSTS/E introduced the notion of circuits for DMC/DMR and DMP/DMV devices used for DECNET. The device names for these devices were changed to conform with the DECNET standard for device names. In order to support this new naming convention, two new tables were added: DDCTBL and UCTTBL. These tables are only used for DECNET communication devices.

2.5.8.1 DDCTBL - Number of DECNET Controllers, UCTTBL Bias

The DDCTBL table is used to check the validity of a specified controller and circuit number. The low byte of each word in DDCTBL specifies the number of controllers of the corresponding type that are on the system. The high byte specifies an offset (in bytes) from the beginning of UCTTBL to the information for controller 0 of the corresponding device.

DDCTBL contains one entry for each DECNET communication device on the system. Since the only communication devices currently supported for DECNET are the DMC/DMR (XM:) and the DMP/DMV (XD:), the maximum size of this table is currently two words.

2.5.8.2 UCTTBL - Number of Units per Controller

The UCTTBL table specifies the number of units supported by each corresponding DMC/DMR or DMP/DMV controller. The DDCTBL table points to the entry in UCTTBL that corresponds to controller zero of each DECNET device. The entries for additional controllers for that device, if any, will immediately follow the entry for controller 0. Since RSTS/E only supports one unit per controller, all entries in UCTTBL will be set to one.

2.6 SEND RECEIVE

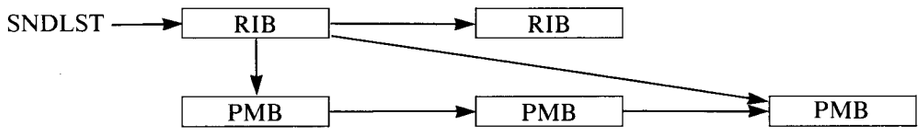
The send/receive capability of RSTS/E allows information to be sent between programs. The data structures required to provide this capability are the Receiver ID Block (RIB) and the Pending Message Block (PMB).

Each program that has declared itself to be a receiver is allocated a Receiver ID Block. This ID block is pointed to by the job's secondary JDB (JDB2).

The RIBs on the system are linked together in a list of message receivers. When a message is sent, or when a DECNET link is received, this list is searched to find the specified receiver. Once found, the message is inserted into the list of pending messages for the receiver.

Each message that is waiting to be received has a Pending Message Block (PMB) associated with it. All the PMBs for a particular receiver are kept in a linked list pointed to by S.MLST in the receiver's RIB. The list is kept in first-in, first-out order.

The following figure shows the relationship between RIBs and PMBs:



For more information on the contents of the Receiver ID Block and the Pending Message Blocks see the *RSTS/E Programming Manual and Network Programming in BASIC-PLUS and BASIC-PLUS-2.*

2.6.1 RIB - Receiver ID Block

When a job declares itself a receiver (using the .MESAG monitor directive or SYS(CHR$(6) + CHR$(22)) in BASIC-PLUS), a Receiver ID Block (RIB) is allocated. The RIB contains all the information necessary to allow receipt of intra-CPU and inter-CPU messages. It is pointed to by J2MPTR in its owner's JDB2.

In addition, all the RIBs in the system are linked together in a list. The first element in this list is pointed to by the location SNDLST. Since the receiver name ERRLOG is always present in the system, SNDLST will never be 0.

The Receiver ID Block has the following format:

Symbol	*Offset*			*Offset*	*Symbol*
		Link to next RIB		0	S.LINK
		Receiver ID (6 bytes of ASCII)		2 4 6	S.RCID
S.OBJT	11	Object type	Job number *2	10	S.JBNO
S.SRBN	13	Sub-RIB number	Access flags	12	S.ACCS
		Buffer maximum		14	S.BMAX
S.MCNT	17	Pending count	Message maximum	16	S.MMAX
		Pointer to first PMB, if any		20	S.MLST
		Pointer to last PMB, if any		22	
S.LCNT	25	Link count	Max link count	24	S.LMAX
		Pointer to first logical link block		26	S.LLST
		Pointer to last logical link block		30	
S.PQTA	33	Packet/message quota	Out link maximum	32	S.OMAX
		Reserved for network use		34 36	

Offset	*Symbol*	*Description*
0	S.LINK	This word contains the address of the next RIB in the linked list. If this is the last RIB in the list, this word will be 0.
2	S.RCID	These six bytes contain the receiver name as six bytes of ASCII. If the receiver name is less than six bytes in length, it will be filled with trailing zero bytes.
10	S.JBNO	This byte contains the job number, times two, of the job to which this RIB belongs.
11	S.OBJT	This byte contains the object type for use in DECNET messages (see section 2.6.1.1)
12	S.ACCS	This byte contains the access controls for this receiver (see section 2.6.1.2).
13	S.SRBN	This byte contains the sub-RIB number.
14	S.BMAX	This word specifies the buffer usage restrictions made at the time the receiver was declared.
16	S.MMAX	This byte specifies the restriction on the number of pending messages made at the time the receiver was declared.
17	S.MCNT	This byte contains a count of the number of messages that are currently pending for this receiver.
20	S.MLST	This word is a pointer to the pending message block (PMB) for the first message pending for this receiver. If no messages are pending, this word will contain a 0.
22		This word is a pointer to the pending message block (PMB) for the last message pending for this receiver. If no messages are pending, this word points to S.MLST.
24	S.LMAX	This byte specifies the restriction on the number of pending DECNET messages made at the time the receiver was declared.
25	S.LCNT	This byte contains a count of the number of logical links for DECNET messages currently pending for this receiver.
26	S.LLST	This word is a pointer to the logical link descriptor block for the first logical link pending for this receiver. If no logical links are pending, this word will contain a 0.
30		This word is a pointer to the logical link descriptor block for the last logical link pending for this receiver. If no logical links are pending, this word points to S.LLST.
32	S.OMAX	This byte specifies the outgoing network link maximum
33	S.PQTA	This byte specifies the quota of packets per message.
34-36		Reserved for network use.

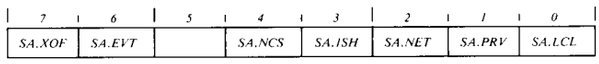
2.6.1.1 S.OBJT - Object Types

The object types specified by S.OBJT have the following meanings:

Value	*Meaning*
0	Local receiver.
1	Error logger.
2	EMT logger.
3	Queue manager.
4-63	Reserved.
64	Disk initializer program.

2.6.1.2 S.ACCS - Access Control Bits

The access control bits contained in S.ACCS have the following meaning when set (see the *RSTS/E Programming Manual* for more information on access control bits):



7	6	5	4	3	2	1	0
SA.XOF	SA.EVT		SA.NCS	SA.ISH	SA.NET	SA.PRV	SA.LCL

Bit	*Symbol*	*Description*
<0>	SA.LCL	Local senders may send to this receiver.
<1>	SA.PRV	Local senders must be privileged to send to this receiver.
<2>	SA.NET	Network senders may send to this receiver.
<3>	SA.1SH	This receiver can only handle one logical link at a time.
<4>	SA.NCS	Do not perform conditional sleep check on this RIB.
<5>		Unused.
<6>	SA.EVT	This RIB is the DECNET event logger.
<7>	SA.XOF	Receipt of messages from local senders is temporarily disallowed.

2.6.2 PMB - Pending Message Block

Each message waiting to be received has a Pending Message Block (PMB) associated with it. The PMB describes the data received for large messages and contains the data sent for small messages.

The pending message blocks for each receiver are kept in a linked list, ordered first-in, first-out. The first and last pending message blocks in the list are pointed to by the two words starting at S.MLST in the associated RIB (see section 2.6.1).

The PMB has following format:

Symbol	*Offset*		*Offset*	*Symbol*
		Link to next PMB if any	0	P$LINK
		Contorted pointer to message buffer	2	P$BUFA
P$SNDR	5	Sender job #*2 | Message type	4	P$TYPE
		Sender PPN	6	P$SPPN
		Unused | Sender KB #	10	P$SKBN
		Bytes remaining in message	12	P$BREM
			14	P$PARM
			16	
			20	
			22	
		Small message data	24	
			26	
			30	
			32	
			34	
			36	

Offset	*Symbol*	*Description*
0	P$LINK	This word contains the address of the next PMB in the linked list. If this is the last PMB in the list, this word will be 0.
2	P$BUFA	This word contains the "contorted" address of the buffer containing the large message data. If the low order five bits of the address are zero, it is a pointer to a small buffer. Otherwise, the address has been rotated left seven bits and points into the extended buffer area, XBUF. If this word is zero, no large message data was sent with this message.
4	P$TYPE	This byte specifies the type of message described by this PMB (see section 2.6.2.1).
5	P$SNDR	This byte contains the job number, times 2, of the message sender.
6	P$SPPN	This word contains the PPN of the message sender.
10	P$SKBN	This byte contains the sender's keyboard number. If the sender was detached, this byte will contain 255.
11		This byte is unused.
12	P$BREM	This word specifies the number of bytes remaining in the message buffer. It can be less than the original length of the message if the message was too long to fit in the user's buffer on a previous receive request and truncation was not requested. See section 2.6.2.2 for information on the buffer format.
14	P$PARM	These ten words contain the message data for small message send/receive or the network parameters for DECNET.

2.6.2.1 P$TYPE - Message Type

If DECNET is not used, the message type (P$TYPE) will always be -1 to signify that a local sender message was received. However, if DECNET is used, several other message types are possible, as follows:

Type	*Description*
-1	Local sender message received.
-2	Connect Initiate received.
-3	Connect Confirm received.
-4	Connect Reject received.
-5	Network sender message received.
-6	Interrupt received.
-7	Link Service received.
-8	Disconnect received.
-9	Link Abort received.
-10	Event logger message received.

2.6.2.2 Buffer Format

Each large message is stored in a buffer of up to 536 bytes (unless an optional patch is entered in the monitor) using either a block of small buffers or a buffer from XBUF. The actual data of the message is preceded by a two word header, as follows:

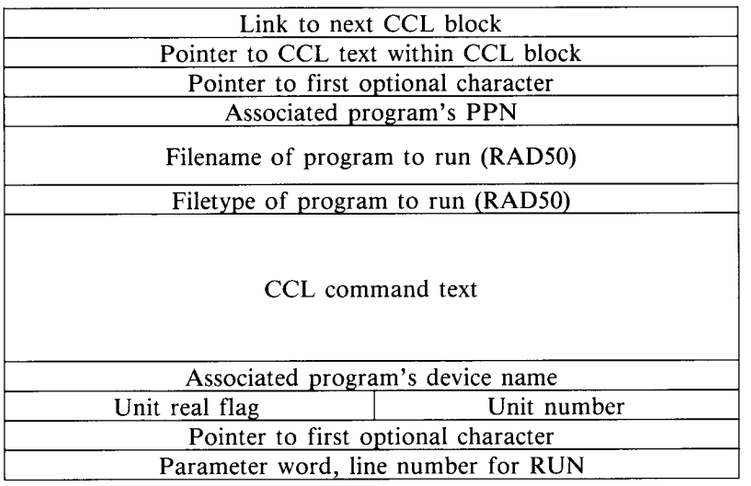
Offset	*Symbol*	*Description*
0		This word specifies the buffer size, in bytes.
2		This word points to the first byte to be returned for the message data. It initially points to offset 4 but is updated if a receive request overflows the user's buffer and truncation is not requested
4		This word starts the message data area. It can be up to 532 bytes in length (unless an optional patch is entered in the monitor).

2.7 CCL - CONCISE COMMAND LANGUAGE BLOCK

Each concise command language (CCL) definition is stored in a CCL block. These blocks are allocated and deallocated dynamically from the FIP pool or the small buffer pool.

The CCL blocks are kept in a linked list in the order in which they were defined. The first element in the list is pointed to by the location CCLLST.

The CCL block has following format:



Offset	*Symbol*	*Description*
0		This word contains the address of the next CCL block in the linked list. If this is the last CCL block in the list, this word will be 0.
2		This word is a pointer to the first character in the CCL command text string.
4		This word is a pointer to the first optional character in the CCL command text string.
6		This word contains the PPN of the program to run in response to this CCL command.
10		These two words contain the file name (in RAD50) of the program to run in response to this CCL command.
12		This word contains the file no type (in RAD50) of the program to run in response to this CCL command. A -1 in this word signifies a wildcard file type (eg. LOGIN.*).
14		These ten bytes contain the CCL command text. The text is terminated by a byte containing 255.
24		This word contains the device name (in ASCII) of the disk device that contains the program to run in response to this CCL command. A 0 specifies the public disk structure, SY:.
26		This word contains the device unit number and unit number "real" flag. See the description of the FIRQB in the RSTS/E System Directives Manual for more information on device naming.
28		This word contains a copy of the pointer stored in offset 4.
30		This word contains the parameter word to pass to the runtime system when the program is run. See the .RUN monitor directive for more information on the parameter word.

2.8 FIXED MEMORY LOCATIONS

Several fixed locations in low memory within the monitor are used to store static information and information about the currently executing job. This information provides a shortcut for accessing the most used job control information.

The dates and times stored in these locations are in RSTS's own internal format. Dates are stored as (year-1970)*1000 + (day of year). Times are stored as the number of minutes before midnight.

The fixed memory locations have the following values:

Octal	*Decimal*	*Symbol*	*Description*
44	36	IDATE	System startup date.
46	38	ITIME	System startup time.
50	40		Start at this location to do a system reload.
52	42		Start at this location to do a crash dump, auto-restart.
54	44	HALT	HALT instruction for system crash.
56	46		Start at this location to do a system reload.
1000	512	DATE	Today's date.
1002	514	TIME	Current time.
1004	516	TIMSEC	Seconds to next minute.
1005	517	TIMCLK	Ticks to next second.
1006	518	JOB	Job number of current job (times 2). If the null job is running, this byte will be zero.
1007	519	NEXT	If this byte is non-zero, it is the job number (times 2) of the job that was scheduled to be the current job, but is not yet swapped into memory. This job will begin execution immediately upon gaining memory residency. If a job is currently running, it was "sub-scheduled" to use the available CPU time while waiting for the next job to swap in.
1010	520	JOBDA	Pointer to current job's Job Data Block (JDB).
1012	522	JOBF	Pointer to current job's flags (JDFLG).
1014	524	IOSTS	Pointer to current job's IOSTS (JDIOST).
1016	526	JOBWRK	Pointer to current job's Work Block (WRK).
1020	528	JOBJD2	Pointer to current job's Secondary Job Data Block (JDB2).
1022	530	JOBRTS	Pointer to current job's Runtime System Descriptor Block (RTS).
1024	532	CPUTIM	Pointer to current job's CPU time bucket (J2TICK).
1026	534	JOBWDB	Pointer to current job's Window Descriptor Block (WDB) at offset W.WIN1.
1040	544	MEMLST	Root of memory control list.
1100	576	DFTRTS	RTS Block for default runtime system.
1140	608		Tail of memory control list.

1200	640	ERLRIB	RIB for ERRCPY receiver.
1240	672	NULRTS	RTS Block for null runtime system.
1300	704	FIQUE	FIP queue root.
1302	706	FIJOB	Job number (times 2) of job currently using FIP or last job to use FIP.
1303	707	FIPRIV	If this byte non-zero, the current user of FIP is non-privileged.
1304	708	FIUSER	PPN of current job in FIP.
1306	710	FIJBDA	Pointer to the Job Data Block (JDB) for the current job in FIP.
1310	712	FIJBD2	Pointer to the Secondary Job Data Block (JDB2) for the current job in FIP.
1312	714	FIPSJN	Job number (times 2) of the system job currently-using FIP.
2070	1080	SYSTAK	Top of the monitor's stack when the null job is not running. The monitor's stack is 256 words deep.
2600	1408	FISTAK	Top of FIP's stack. FIP's stack is 124 words deep.

Chapter 3
DEVICE DRIVERS

When support is needed for a device not normally supported by RSTS/E, a custom device driver must be written. Although user written device drivers are not supported by Digital, all the files necessary to build them are provided in the system generation kit.

User written device drivers are essentially only allowed for non-disk devices. Although user written disk device drivers are possible, they generally require the source code for INIT so that it may be rebuilt with extended tables and support code for the new disk. For this reason, disk drivers will not be specifically discussed.

Even though foreign disk drivers are not supported, various disk related tables and structures used by the monitor and INIT can be modified to better support foreign disks that look substantially like supported Digital disks. See Chapter 2 for more information on disk related monitor tables.

The procedure for writing a driver for RSTS/E is very similar to that used for other Digital operating systems. The driver must provide several "subroutines" for the monitor to use to perform specific functions. The monitor, in turn, provides a set of routines for the driver's use.

Before a device driver can be written, several design decisions must be considered. Is the device character oriented or block oriented? How fast is the data being transferred? Can the transfer be interrupted by a ↑C? Does the device do DMA transfers? Does the job need to be resident during the I/O operation? Can the controller support more than one concurrent I/O request? Each of these considerations will have an effect on how the driver should be written.

Block oriented devices can transfer data directly to and from user buffers using DMA transfers or one character at a time under control of the driver. The data can be stored in small or large buffers, or within an area set aside in the Device Data Block (DDB), until needed by a user program.

Character oriented devices store data in temporary buffers within the monitor before transferring it to the output device or to the user program. Unless the program is stalled waiting on input, or for buffer availability on output, it can overlap its execution with the transfer of data to or from the I/O device. Small or large buffers can be used to store the data.

When a user program requests input from an asynchronous device, the input request will be honored immediately if the input is already totally contained in small buffers. If the input is not already totally buffered, the user program will be stalled until the data is available. Once the data is available, the input request is requeued and can now be processed using the buffered data.

When a user program requests output to an asynchronous device, the data is transferred to small buffers before being sent to the physical device. If there are sufficient buffers available to hold all the output data, the user program can continue to execute while the data is being transferred to the device. If insufficient buffers are available, the user program will be stalled until the device driver has freed up enough buffers to hold the remaining data.

3.1 GENERAL STRUCTURE

Device drivers must follow very specific rules regarding format and content. There are several global symbols and specific entry points in every driver. Each driver must be divided into several specific program sections (PSECTs) with each PSECT having a specific name. There are also optional entry points and program sections which, if used, must follow specific rules.

Every user written device driver requires at least two files: xxPRE.MAC and xxDVR.MAC (where "xx" is the name of the device). In addition, disk device drivers and other drivers that are being included in INIT, require a third file, xxDEF.MAC. This file provides information about register and bit definitions for the device.

The xxPRE.MAC file contains the information about the driver that is required to build the TBL.MAC module. This information includes use of the DEVICE macro (which defines the device name and other information) and definition of several constants used by the TBL assembly to set up monitor tables for this device.

The required constants for the xxPRE.MAC file are CNT.xx (number of units of this device), DDS.xx (size of the DDB), and CCC.xx (↑C interuptable flag). Several other constants are optional. They are BFQ.xx (small buffer quota), HOR.xx (horizontal width), SLP.xx (check with driver before allowing a conditional SLEEP), and UMR.xx (driver uses unibus mapping registers).

The xxDVR.MAC file contains the program code and data for the driver. The driver is divided into two to four program sections (PSECTs). Each PSECT has a specific name and is used for a specific purpose. The first PSECT is named xxDVR and contains the code to perform all the functions required for interfacing with a device. The second PSECT is named xxDINT and contains a specific set of instructions which map and execute the interrupt handler in the xxDVR section when an interrupt occurs. An optional PSECT may be included to contain read-write data for the driver other than that contained in the DDB. This PSECT is names xxDCTL. Another optional PSECT may be included to contain read-only data that is permanently mapped by the monitor and available even when the read-only code for the driver is not mapped. This PSECT is named xxDTBL.

Several global constants are defined in the xxDVR file. These constants are used to build tables when the driver is linked with the monitor. These global constants are: STS.xx (DDSTS value for the DDB), FLG.xx (device description flags), SIZ.xx (line width), and BUF.xx (default buffer size).

Several global constants are defined for the driver by the TBL assembly. These constants are: JS.xx (JBWAIT/JBSTAT bit), IDX.xx (device driver index), TIM.xx (location of timeout table entry for unit 0), CSR.xx (location of CSR table entry for unit 0), xxDDDB (location of DDB for unit 0), ALT.xx (synonym for device name), and LOG$xx (EMT to envoke error logging).

Several specific entry points are required within the xxDVR PSECT. These entry points are defined by the following global symbols: .ASN$xx (assign), DEA$xx (deassign), OPN$xx (open), CLS$xx (close), SER$xx (read/write), SPC$xx (.SPEC), INT$xx (interrupt), and TMO$xx (timeout).

Several additional entry points can be defined when needed. They are: ERL$xx (error logging), SLP$xx (check before allowing conditional SLEEP), UMR$xx (unibus mapping register is available) and a set of entry points for Level Three Queue (L3Q) entries.

All communication between the diver and a user program is made via the XRB or the FIRQB. The contents of the FIRQB are passed to the driver when the device is opened. The contents of the XRB are passed to the driver when a read, write or special service request is made. Information may also be returned to the user program by the driver using these structures. See the *System Directives Manual* for more information on the XRB and FIRQB.

The FIRQB or XRB is copied to the user's WRK block before being made available to the driver. This allows information from the FIRQB or XRB to be accessed even if the job image containing the FIRQB and XRB is swapped out.

A pointer to the user's buffer is passed to the driver when entered at its SER$xx or SPC$xx entry point. The beginning of the user's buffer is mapped by memory management register APR6 before the driver is entered. The pointer refers to data mapped by this memory management register.

Each unit of each device has a Device Data Block (DDB) associated with it. The DDB contains all the read-write data required to control each device. Portions of the DDB have a specific format, while other portions are defined by the driver for its own use. See section 2.5.1 for more information about DDBs.

3.1.1 PSECT Usage

Three program sections (PSECTs) are used when writing a driver. The first two, xxDVR and xxDINT, are required. The third one, xxDCTL, is optional and is only included when general read-write data is needed for a driver.

3.1.1.1 xxDVR — Driver Read-Only Code Section

The xxDVR PSECT contains all the code for the device driver. This section is located in the read-only portion of the monitor's virtual address space. It does not actually take up address space in the monitor unless it is in use. When needed, the driver is mapped by kernel memory management register APR5 at a base address of 120000_8.

The xxDVR PSECT contains the entry points and associated code for all services provided by the driver. It is read-only. Any read-write data must be contained in the driver's DDBs or in the optional program section, xxDCTL.

To define this PSECT use the DEFORG macro as follows (replace "xx" with the device name):

```
DEFORG xxDVR
```

3.1.1.2 xxDINT — Interrupt Service Dispatch Section

The xxDINT PSECT contains code to map the interrupt service code contained in xxDVR, save all registers and dispatch to the appropriate interrupt service routine. It is contained in the read-only portion of the permanently mapped section of the monitor.

The interrupt vectors for all devices handled by this driver point into this section. When unit 0 of a device interrupts, it vectors to the beginning of this section at location xxDINT. The code at this location calls a routine named INTSAV which loads kernel APR5 with the value 1200_8. This value maps the driver into virtual addresses starting at 120000_8. INTSAV then saves registers R0-R5 on the stack and transfers control to the location specified in the second parameter word of this call (normally INT$xx).

If only one interrupt vector is required for this device driver, the following code should be used to define this section (replace "xx" with the device name):

```
        DEFORG  xxDINT
xxDINT: CALLX   INTSAV,R5       ;Map and enter interrupt routine
          +     1200            ;PAR5 value
          +     INT$xx          ;Interrupt service routine address
        ORG     xxDVR
```

If more than one interrupt vector is required for the device driver, one of two approaches may be taken. The preferred method is to point each vector to the same interrupt service routine and store the associated device unit number in the PSW word of each vector. The monitor multiplies the value in the least significant four bits of the PSW word of the vector by two and passes this value to the interrupt service routine in R0 when an interrupt occurs. The interrupt service routine can determine which unit interrupted by checking this value.

The second method is to have a separate interrupt entry point in xxDINT for each interrupt vector. The following code can be used to define this section in this way (replace "xx" with the device name):

```
        DEFORG  xxDINT
        ;Interrupt vector for unit 0
xxDINT: CALLX   INTSAV,R5       ;Map and enter interrupt service
          +     1200            ;PAR5 value
          +     INT$xx          ;Interrupt service routine address
        ;Repeat the following for each additional interrupt vector
        ;  xxnINT is the symbol for the location to vector
        ;  to for unit "n" of the device. INTxxn is the
        ;  address of the associated interrupt service routine.
xxnINT: CALLX   INTSVX,R5       ;Map and enter interrupt service
          +     xxDAP5          ;Pointer to PAR5 value
          +     INTxxn          ;Interrupt service routine address
        ORG     xxDVR
```

3.1.1.3 xxDCTL - Read-Write Data Section

This PSECT is optional. It is used when the driver needs read-write data that is general to the driver, as opposed to data kept for each unit of a device. A common example of this is the root word of a queue used to store overlapping requests for use of a device by more than one job.

To use this section, include the following in the xxDVR module:

```
DEFORG  xxDCTL          ;Define PSECT for device "xx"
                        ;Read-write data definitions go here
ORG     xxDVR           ;Return to driver code PSECT
```

If the device supports only one unit, all the read-write information about a device can be stored in its DDB; xxDCTL is not required.

3.1.1.4 xxDTBL - Read-Only Data Section

This PSECT is optional. It is used to contain read-only data that should be included in the permanently mapped portion of the monitor. Information in this PSECT is available for use by other device drivers and other portions of the monitor.

To use this section include the following in the xxDVR module:

```
DEFORG  xxDTBL          ;Define PSECT for device "xx"
                        ;Read-only data definitions go here
ORG     xxDVR           ;Return to driver code PSECT
```

3.2 ENTRY POINTS

A device driver actually consists of several modules of code. Each of these modules performs a separate function when requested by the monitor. For example, there is a module which performs device specific code for an open request and another module which performs device specific code for a close request.

The starting address of each of these modules is defined by specially named global symbols. These global symbols are called entry points. Every driver contains several entry points. These entry points are used for opens, closes, I/O requests and other functions.

The entry points have specific names, such as OPN$ and CLS$. These names are combined with the appropriate device name to form a unique symbol for the entry point. For example, the entry point for open requests for the device DX would be OPN$DX.

Before the monitor enters the driver it verifies that the requested operation is allowed, and sets up information in the FIRQB or XRB as required. The driver only has to handle the device specific operations on the requested function. All common functions have already been taken care of by the monitor.

3.2.1 ASN$xx - Assign

```
Input:  R0  Job number of assigner times 2.
        R1  Pointer to DDB for appropriate unit of device.

Output: All registers must be preserved.

Exit:   RETURN              ;No error
or:     ERROR code          ;If error in assignment
```

This is the entry point for device assignment. It is used to perform any necessary device specific functions at the time a device is assigned. This entry point is entered when the device is explicitly assigned by an ASSIGN or implicitly assigned by an OPEN when it is not already assigned.

Since this section is entered before the first time the device is opened it can be used to set up the interrupt vectors in user written device drivers. The address of location xxDINT in PSECT xxDINT should be put into the first word of the vector. The desired PSW should be stored in the second word of the vector. See section 3.6 for a discussion of interrupt vector handling.

3.2.2 DEA$xx - Deassign

```
Input:  R1  Pointer to DDB for appropriate unit of device.

Output: All registers must preserved.

Exit:   RETURN
```

This is the entry point for device deassignment. It is used to perform any necessary device specific functions at the time a device is deassigned. This entry point is entered when the device is explicitly deassigned by a DEASSIGN or implicitly deassigned by closing the device if it is not also open on another channel.

3.2.3 OPN$xx - Open

Input:	R0	Unit number times two.
	R1	Pointer to DDB for appropriate unit
	R4	Pointer to FIRQB (in WRK block). Default values have been loaded for FQFLAG and FQBUFL using the monitor tables FLGTBL and BUFTBL.
	R5	Pointer to job's IOB entry for this channel.
Output:	R0	Random
	R1	Must be preserved
	R4	Random
	R5	Must be preserved

```
Exit:   RETURN                      ;For successful open
 or:    CALLX   RETDEV              ;For unsuccessful open
        ERROR   code
```

This routine is entered each time the device is opened. It is used to perform any initialization necessary when the device is first opened. A common requirement is to use information available in the FIRQB to initialize or modify parameters in the DDB. Another common requirement is to turn on interrupt for the associated device.

3.2.4 CLS$ - Close

Input:	R0	Unit number times 2
	R1	Pointer to DDB
	R5	Pointer to DDB
	Z-bit	Set: This is a real close
		Clear: This is a "reset" close
Output:	R0	Random
	R1	Random
	R5	Must be preserved

```
Exit:   RETURN
```

This routine is entered each time the device is closed. It is used to perform any housekeeping necessary when the device is closed. A common requirement is to turn off interrupts for the device. Note that data may still be buffered for I/O at the time the close request is received. Input data should normally be discarded. Output data should normally continue to be transferred to the output device before turning off interrupts.

There are two different types of close requests: normal and reset. The type of close is signified by the Z-bit of the PSW. A "reset" close is the same as a normal close except that any buffers currently in use are immediately released and no housekeeping functions are performed that would access the device. A reset close is signified in BASIC by a negative channel number in the CLOSE call.

3.2.5 SER$xx - I/O Service

Input:	R0	Unit number times 2
	R1	Pointer to DDB for this unit
	R2	Function code: 2 (.READ) or 4 (.WRITE)
	R3	Pointer to XRB (contained in WRK block)
	R4	Calling job number times 2
	R5	Pointer to user's buffer (mapped through APR6)
	Z-bit	Set: This is the first entry for this request
		Clear: This entry is for an IOREDO
	C-bit	Set: This entry is for an IOREDO
		Clear: This is the first entry for this request
Output:	All registers random	
	XRBC	Adjusted for the number of bytes transferred to or from the user's buffer
	XRLOC	Adjusted for the bytes transferred to or from the user's buffer.
	XRBLK	0 for non-block structured devices. Next virtual block number for block structured devices.

```
Exit:   JMPX    IOEXIT          ;I/O completed without error
or:     SETERR  code,@IOSTS     ;I/O completed with error
        JMPX    IOEXIT
or:     JMPX    IOREDO          ;Stall the job and then reenter
                                ;SER$xx when the job is unstalled
```

This routine is entered when an I/O request is received. The monitor has already verified that the device is open and that the caller has the necessary access rights for the requested function.

Before the monitor enters the driver's SER$xx entry point it stalls the job by setting the JS.xx bit for this device in the job's JBWAIT word. The job will remain stalled until the driver allows it to continue by jumping to the IOEXIT routine in the monitor.

If the device uses block mode transfers, the job should be locked in memory, stalled for I/O redo and a DMA transfer started for the requested function. When the transfer is complete, the interrupt service routine will cause the driver to be reentered at its SER$xx entry point, but with the Z-bit set and the C-bit clear. The I/O service routine can then exit with the error condition, if any, shown in IOSTS.

If the device uses buffered asynchronous transfers, one of two things occurs:

(l) If the request is for a .READ, the DDB should be checked to see if enough data has already been buffered to complete the requested input. If so, the data is transferred to the caller and the input request is completed by exitting to IOEXIT. If not, the caller is stalled waiting on input by exitting to IOREDO. When the interrupt service routine has buffered enough data to complete the request it uses the IOFINI subroutine to unstall the waiting job. The job's SER$xx routine will be reentered to redo the input request. This time the required data will already be available in buffers and the I/O can complete immediately.

(2) If the request is for a .WRITE, the data is transferred from the caller into monitor buffers and device output is started. If there are insufficient small buffers to contain all the data being supplied, the caller is stalled by exitting to IOREDO. When the interrupt service routine determines that there should now be enough buffers available to continue, it uses IOFINI to reschedule the entry into SER$xx so that additional data may be buffered. This process continues until the entire request is buffered, at which time the output request completes by exitting to IOEXIT.

Whenever a device transfer is initiated, a timeout check should be started. This timeout check ensures that the I/O will abort if the device hangs for some reason and never responds to the I/O request.

To initiate a timeout check, the location TIM.xx should be set equal to the maximum number of seconds, plus one, that a device should ever take to respond to a request. For example, if the lineprinter should never take more than one second to respond to a request, the following sequence would cause a timeout if the printer controller was to hang:

```
MOV     #2,TIM.LP(R0)          ;Set LP: timeout counter to 1 (+1) seconds
```

The timeout check can also be used to periodically enter the driver so that time dependant conditions can be checked. This is especially useful when a device may require periodic correction of an error condition and its status should be checked regularly. See section 3.2.8 for more information on timeouts.

3.2.6 SPC$xx - Special Service

```
Input:  R0      Unit number times 2
        R1      Pointer to DDB for this unit
        R2      Special function code
        R3      Pointer to XRB (in WRK block)
        R4      Calling job number times 2
        R5      Pointer to XRB (mapped through APR6)

Output: All registers random

Exit:   JMPX    RTI3            ;If no error
or:     ERROR   code            ;If error
```

This routine is entered when the user issues a .SPEC call. .SPEC calls are used to perform device specific special function handling. An optional parameter can be passed in the .SPEC call. This parameter is stored in the second word of the XRB (XRBC).

If the driver needs to return any information to the user, it can do so by accessing the mapped XRB directly or by storing the information in the copy of the XRB in the WRK block. If the WRK block is used, the JFPOST bit must be set in the job's JDB to request that the data be posted to the user's XRB.

If the driver does not support .SPEC calls, it should return the error PRVIOL to the caller using the following statement:

```
ERROR   PRVIOL
```

See the description of .SPEC in the *System Directives Manual* or SPEC% in the *RSTS/E Programming Manual.*

3.2.7 INT$xx - Interrupt Service

```
Input:  R0      Unit number times two (unit number is found in the least significant four bits of
                the PSW word of the vector)
        PR      Priority is device interrupt priority

Output: All register random

Exit:   RETURN                          ;Normal return from interrupt
or:     CALLX   IOFINI,R5,JS.xx         ;Reschedule stalled job
        RETURN                          ;Return from interrupt
```

This routine is entered every time the device interrupts. The interrupt vectors to the entry in PSECT xxDINT. This code maps the driver code using APR5, saves registers R0-R5 and calls this interrupt service routine to process the interrupt (see section 3.1.1.2).

R0 is loaded with the unit number times two, taken from the PSW word of the interrupt vector. If the device supports more than one interrupt vector, they can all interrupt to the same address (xxDINT). The unit number for the interrupting device is stored in the least significant four bits of the PSW word of each vector for the device. This unit number (times two) will be loaded into R0 before the interrupt service routine is called and can be used to determine which device interrupted.

The interrupt service routine controls the rescheduling of I/O service requests (ie. re-entries into SER$xx) that have been stalled waiting for buffer availability or other conditions. When the interrupt service routine determines that a condition has been met that the service request is waiting for, it reschedules an entry into the SER$xx entry point using the IOFINI monitor subroutine.

If the interrupt service routine needs to do any extended processing that does not have to be done at interrupt priority, it should schedule a reentry into the driver and continue processing after being reentered. A reentry is requested by setting an appropriate bit in the Level Three Queue (L3Q). These bits are defined by the DEVICE macro used in the xxPRE.MAC file see section 3.4.9).

When the interrupt service routine exits, the monitor will check L3Q to see if any requests are pending. The driver will then be reentered at its appropriate L3Q entry point and can continue processing at priority level three. See section 3.2.12 for more information on L3Q entries.

3.2.8 TMO$xx - Timeout

Input:	R0	Unit number times 2
	R1	Pointer to the DDB for this unit
	R3	Pointer to the device CSR (loaded from CSRTBL)
	PR	Priority is PR3
Output:	All registers are random	
Exit:	RETURN	

This routine is entered when the time period specified by TIM.xx has expired. It is typically used to handle devices that never complete the I/O function requested of them. It can also be used for device specific periodic checks.

The TBL assembly equates a word in the monitor table TIMTBL to the name TIM.xx for each driver. If more than one unit of a device is present, additional words are allocated immediately following TIM.xx for use by the additional units (see section 2.5.5.7). This word can be set to one of three ranges of values, depending on the desired function:

1. Zero signifies that no timeouts are desired.
2. A non-zero, positive value indicates the number of seconds desired before a timeout occurs. This value will be decremented each second by the monitor. If it reaches zero, the associated driver will be entered at its timeout entry point (TMO$xx). Note that a full second does not necessarily pass between the time TIM.xx is set and the first time it is decremented. For this reason the value stored in TIM.xx should normally be one greater than the desired value.
3. A negative value indicates that the timeout entry point should be entered each second. Note that less than a full second may pass between the time TIM.xx is set and the first entry at TMO$xx.

The typical procedure for handling devices that have actually hung (presumably due to a hardware problem) is to return any monitor buffers currently in use, call error logging, post a hung device error and reschedule the job for execution. The following procedure will perform these functions for a device that supports one unit and uses only one chain of small buffers:

```
CLR     TIM.xx                      ;Reset the timeout indication
                                    ; Use the following two lines only if error logging is
                                    ; supported
MOV     xxCSR,R3                    ;Point to CSR
LOG$xx                              ;Enter error logging routine
CALLX   CLRBUF,R5,DDBUFC+EP         ;Clear the small buffer chain
SETERR  HNGDEV,@IOSTS               ;Post a "Hung device" error
CALLX   IOFINI,R5,JS.xx             ;Mark the job a runnable
RETURN                              ;Return to the monitor
```

3.2.9 ERL$ - Error Logging

Input:	R1	Pointer to DDB for this unit
	R3	Pointer to CSR for this unit
Output:	All registers are random	
Exit:	RETURN	

This routine is entered if the driver requests that an entry be made in the error log. This routine is only needed if the driver is going to do error logging. If the driver does not do error logging, the symbol ERL$xx should be globally equated to zero, as follows:

```
ERL$xx==0
```

The TBL assembly equates the global symbol LOG$xx to a special EMT code for logging errors for this driver. When the LOG$xx EMT is executed in a driver, the driver is interrupted and control passes to the ERL$xx entry point for that driver with all registers preserved. The ERL$xx routine logs the error and then resumes execution of the driver.

The monitor provides a subroutine (ERLDVR) for logging device specific information. This routine is described in section 3.5.14. The following code is an example of the call to the error logging routine:

```
ERL$xx::    CALLX  ERLDVR,R5        ;Call the monitor error logger
              .BYTE error           ;Error code (see KERNEL.MAC)
              .BYTE ddbsiz          ;Size of DDB in bytes
              .BYTE offset,count    ;Offset from CSR and word count
              .BYTE 0,0             ;End of argument list
            RETURN                  ;Return from EMT
```

3.2.10 SLP$xx - Sleep Check

Input:	R0	Unit number times 2
	R1	Pointer to DDB for this unit
	R4	Pointer to job's IOB entry for this channel
Output:	R0	Random
	R1	Random
	R4	Random
	C-bit	Set: Don't let the job sleep
		Clear: Let the job sleep
Exit:	RETURN	

This routine is entered before allowing a job that has this device open to sleep for a conditional sleep request. Conditional sleep is specified by a negative value for sleep time. If the device does not wish to allow the job to sleep, it can return with the C-bit set and the sleep request will not be honored.

This entry point is optional. If the driver supports this entry point, the symbol SLP.xx should be equated to 1 in the prefix file (xxPRE.MAC) used in the TBL assembly. If SLP.xx is not defined, this entry point will not be used.

3.2.11 UMR$.xx - Unibus Mapping Register Available

Input:	R0	Pointer to base root of DSQ list for disk device drivers. Random for non-disk drivers.
	R3	Pointer to CSR for disk drivers. Random for non-disk drivers.
	PR	Priority is PR5
Output:	None	
Exit:	RETURN	

When a driver that does DMA transfers requests a unibus mapping register (UMR), the request will be denied if there are no UMRs available at the time of the request. If a request is denied, the monitor will remember that a UMR is still needed. When a UMR becomes available the monitor will enter the UMR$xx entry point in all drivers that have this entry point.

Entry at the UMR$xx entry point does not indicate that the driver has been allocated a UMR. It only indicates that at least one UMR is now free. The driver should reissue the GETUMR call to try and allocate a UMR. The request may still fail because another driver may have already allocated the last free UMR before the request is made by this driver.

3.2.12 nnn$xx - Level Three Queue Reentry

Input:	All registers are random	
	PR	Priority is PR3
	C-bit	Clear
Output:	All registers are random	
	PR	Must be preserved
Exit:	JMPX	RTI3

In some cases it is desirable to split off some portion of the processing required in an interrupt service routine so that it may be performed at a lower priority. This capability is commonly known as a fork process. RSTS/E provides this capability using the Level Three Queue (L3Q).

The L3Q is a set of 32 bits, each of which signifies a separate function to be performed by the monitor. Fourteen of these bits are available for use by device drivers. Before the monitor returns to a user job, it checks L3Q. If a bit is set, the routine associated with that bit is entered. This process continues until no bits are set in L3Q, at which time control returns to the user job.

A device driver signifies that it wants to use the L3Q mechanism using the DEVICE macro in the prefix file used for the TBL assembly (see section 3.4.9). A three character name is specified in the DEVICE macro argument list for each bit required by the driver.

Each name is used to create two unique symbols: Qxxnnn and nnn$xx, where "xx" is the device name and "nnn" is the L3Q bit name. The Qxxnnn symbol refers to the associated bit in L3Q. The nnn$xx symbol is the name of the routine in the driver to enter when the associated bit is set in L3Q. These symbols are globalized by the TBL assembly.

The L3Q bit name may be any combination of three letters not already used for an entry point name in this driver. Common names are CON for a continuation point and DNE for I/O completion processing.

3.3 SYMBOLIC VALUES

Several symbols are used by device drivers. Each of these symbols represents a specific piece of information that is either used by the driver or is used in the TBL assembly when building the monitor tables.

The following symbols are used in the driver and must be specified at the beginning of each driver: STS.xx, FLG.xx, SIZ.xx and BUF.xx.

The following symbols are specified in the driver's xxPRE.MAC file. They are used when the monitor tables are created during the TBL assembly. The symbols are: CNT.xx, DDS.xx, and, optionally, CCC.xx, BFQ.xx, HOR.xx, SLP.xx and UMR.xx.

The TBL assembly produces several global symbols that are of use to the driver. These symbols are: ALT.xx, TIM.xx, IDX.xx, JS.xx, CSR.xx, DEV.xx, xxDDB and LOG$xx.

3.3.1 STS.xx - DDB Status Byte

Purpose: Define device status

File: xxDVR.MAC

Scope: Global

Example: STS.LP==DDRLO/400

STS.xx is equated to the desired value of DDSTS in the DDB. This byte describes the static characteristics of the device. See section 2.5.1 for a description of the DDB.

Although DDSTS is a high order byte in the DDB, STS.xx is set up as a low order byte. The usual practice is to combine the standard symbols representing the bit values in DDSTS and divide the result by 400_8.

3.3.2 FLG.xx - Device Dependent Flags

Purpose: Define device dependent flags for FLGTBL

File: xxDVR.MAC

Scope: Global

Example: FLG.PX==DDNFS!DDRLO!FLGPOS!FLGMOD!FLGFRC!376

FLG.xx defines the characteristics flags and handler index for this device. This value is used by the monitor when building the FLGTBL table. The low byte of this word contains the handler index for the device. The high byte contains a set of bits that describe the device characteristics. See section 2.5.5.1 for a description of the FLGTBL table and the characteristics flag bits.

The handler indices are the same for all systems. The values for all standard devices are defined in COMMON.MAC. User written drivers may use any handler index value that is not already assigned to a standard device driver. The suggested method is to start user device handler indices at 376_8 and decrease this value by two for each additional user written driver. Handler indices must be even.

3.3.3 SIZ.xx - Line Width

Purpose: Define line width for the device

File: xxDVR.MAC

Scope: Global

Example: SIZ.LP==0

SIZ.xx defines the line width for this device. It has three possible values:

1. 5*14.+1 signifies that line width does not apply to this device.
2. width+1 signifies that line width applies and is fixed to this size.
3. 0 signifies that line width applies and is variable. The real line width will be found in the DDHORC byte of the DDB.

3.3.4 BUF.xx - I/O Buffer Size

Purpose: Define default buffer size in BUFTBL

File: xxDVR.MAC

Scope: Global

Example: BUF.LP==132.

BUF.xx defines the default buffer size (in bytes) for this device. This value is entered into the appropriate word of the BUFTBL table by the TBL assembly. If the user does not specify a buffer size when the device is opened, this value will be loaded into FQBUFL in the FIRQB before the driver is entered for the open request. Allowable values for BUF.xx are 2-32766 (2-77776_8).

3.3.5 CNT.xx - Number of Units for Device

Purpose: Define the number of units of a device present on the system

File: xxPRE.MAC

Scope: Local

Example: CNT.LP=2

CNT.xx specifies the number of units for this device. It is used during the TBL assembly to build the DEVCNT table and to build a DDB for each unit of the device. It must be included in the prefix file included in the TBL assembly (see section 3.1).

3.3.6 DDS.xx - DDB Size

Purpose: Define the size of the DDB for this device

File: xxPRE.MAC

Scope: Global

Example: DDS.LP=40

DDS.xx specifies the size (in bytes) of each DDB for this device. It is used to set aside space in the monitor's read/write area during the TBL assembly. Although DDS.xx is defined as a local symbol in xxPRE.MAC, it is globalized in the TBL assembly, and therefore may be referenced from the driver and other parts of the monitor.

3.3.7 CCC.xx - ^C Flag

Purpose: Signify whether the device is ^C interruptable

File: xxPRE.MAC

Scope: Local

Example: CCC.LP=1

CCC.xx is an optional flag that signifies whether I/O to this device can be interrupted by a ^C. If CCC.xx is equated to 1, the device uses small buffers for all of its I/O and the user I/O request can be aborted when a ^C is typed. If an I/O request is aborted, the monitor simply clears the JFREDO bit in the user's JDB and unstalls the job. The driver is not notified of the abort. The TBL assembly will allocate a unique bit (JS.xx) in JBSTAT/JBWAIT for devices that are ^C interruptable (see section 3.3.15).

If CCC.xx is equated to 0, or is not defined, the device performs DMA transfers directly to user buffers and cannot be interrupted when a ^C is typed. If a ^C is typed, the JFCC or JF2cc bit will be set in the user's JDB and can be checked by the driver if necessary. The TBL assembly will equate JS.xx to bit 0 in JBSTAT/JBWAIT for devices that are not ^C interruptable. This bit corresponds to the bit JS.SY.

3.3.8 BFQ.xx - Buffer Quota

Purpose: Define small buffer quota

File: xxPRE.MAC

Scope: Global

Example: BFQ.LP=20.

BFQ.xx defines the desired small buffer quota for the device. This value is stored in the DDB at offset DDBUFC + BC and is used in the FREBUF subroutine call to check for small buffer availability (see section 3.5.1).

A device's small buffer quota is used to specify the number of small buffers that the device is "quaranteed" to have available for allocation. The monitor will not refuse a request for small buffers for a device that has not already exceeded its quota unless less than 20% of the small buffers are available (see section 3.5.1).

BFQ.xx is an optional parameter. If it is included in the xxPRE.MAC file, it will be assigned a value of zero. BFQ.xx only has meaning for devices that use small buffers to hold data during I/O.

Since BFQ.xx is stored in the buffer control area of the DDB, the DDB must have been set up to include a buffer control area and this area must be at the standard offset within the DDB (see section 2.5.1).

3.3.9 HOR.xx - Horizontal Line Width

Purpose: Specify horizontal line width for device

File: xxPRE.MAC

Scope: Local

Example: HOR.LP=132.

HOR.xx is an optional parameter that specifies the horizontal line width for this device. If HOR.xx is defined, its value, plus one, will be stored in the DDHORZ and DDHORC offsets in each DDB for the device. If HOR.xx is defined, the DDB must be of sufficient size to include DDHORZ and DDHORC and they must be at their standard offsets within the DDB (see section 2.5.1).

3.3.10 SLP.xx - Check Before Sleeping Flag

Purpose: Specify whether the driver needs notification before honoring a conditional sleep request.

File: xxPRE.MAC

Scope: Local

Example: SLP.LP=0

SLP.xx is an optional flag used to specify whether the driver has a SLP$xx entry point and wants to be notified before honoring a conditional sleep request for a user who has the device assigned. If SLP.xx is defined and is non-zero, the driver will be entered at its SLP$xx entry point before honoring conditional sleep requests. If SLP.xx is defined as 0, or is not defined, the driver does not need a SLP$xx entry point and will not be notified of conditional sleep requests. Conditional sleep requests are signified by the user by a negative value for the time to sleep.

3.3.11 UMR.xx - Notify Driver When UMR is Available

Purpose: Specify that the driver contains a UMR$xx entry point and should be notified when a unibus mapping register becomes available.

File: xxPRE.MAC

Scope: Local

Example: UMR.xx=0

UMR.xx is an optional flag used to specify whether the driver has a UMR$xx entry point and wants to be notified when a unibus mapping register (UMR) is free. If UMR.xx is defined and is non-zero, the driver will be entered at its UMR$xx entry point when a UMR becomes free if any driver failed in its last request for allocation of a UMR. Note that this driver may not have been the one that failed. Note also that entry at UMR.xx does not guarantee that a UMR is available. Another driver may have already allocated the last UMR before this driver is called.

3.3.12 ALT.xx - Alternate Device Name

Purpose: Equated to the alternate device name, if any, specified in the DEVICE macro for this device.

File: TBL.MAC

Scope: Global

ALT.xx is equated to a device's alternate name (synonym) if an alternate name was specified in the DEVICE macro in the xxPRE.MAC file (see section 3.4.9). This symbol is produced by the TBL assembly and will exist only if the device has an alternate name and that name is not already in use as the name of a previously defined device.

3.3.13 TIM.xx - Timeout Clock Setting

Purpose: Used to specify the amount of time to wait on a device operation request before taking special action.

File: TBL.MAC

Scope: Global

Example: MOV #5,TIM.LP(R0)

TIM.xx is the address of the first word in the timeout table (TIMTBL) assigned to this device. It is defined by the TBL assembly. If there is more than one unit configured for this device, the entries for the additional units follow TIM.xx in the table. They are accessed using the unit number times two as an offset from TIM.xx.

Each time an operaton is started, the word in TIMTBL corresponding to the desired unit should be set to the maximum number of seconds, plus one, that the operation could take to complete. When the operation completes, this word should be cleared. The device driver will be entered at its TMO$xx entry point if a device fails to complete the requested function within the allotted time.

3.3.14 IDX.xx - Driver Index

Purpose: Defines the driver index

File: TBL.MAC

Scope: Global

IDX.xx is the driver index code for this device driver. Each device driver is assigned a unique code by the TBL assembly. This index code is used to access information in many of the monitor's tables. It is stored in each DDB at offset DDIDX.

3.3.15 JS.xx - JBWAIT/JBSTAT Status Bit

Purpose: Bit to set in JBWAIT to stall a job while waiting on I/O from specified device. Bit to set in JBSTAT to destall a job previously stalled by setting the corresponding bit in JBWAIT.

File: TBL.MAC

Scope: Global

Example: CALLX IOFINI,R5,JS.LP

JS.xx is equated by the TBL assembly to a bit value to set in the caller's JBWAIT word when the job needs to be stalled by the driver. If the AND of a job's JBWAIT word with its JBSTAT word does not produce a non-zero result, the job is not runnable. The bit or bits set in JBWAIT signify what the job is waiting for.

When a driver is entered at its SER$xx entry point, the driver's JS.xx bit has already been set in the calling job's JBWAIT word to show that the job should wait on service by the device. When the service routine exists by jumping to IOEXIT, the JS.xx bit will be set in the calling job's JBSTAT word, making the job runnable again. If the service routine exits to IOREDO, the job is left in its non-runnable state.

3.3.16 CSR.xx - Pointer to CSR Address

Purpose: Point to the entry in CSRTBL for this device

File: TBL.MAC

Scope: Global

Example:
```
MOV    CSR.LP,R1    ;Get CSR address for LPO:
```

CSR.xx is equated by the TBL assembly to the entry in CSRTBL that corresponds to the CSR address for this device. CSRTBL is loaded with the starting CSR address for each unit of each standard device. It contains zero entries for a user written driver.

Since CSRTBL is contained in a read-only area of the monitor it cannot be changed by the driver and, therefore, is of limited use to a user written driver. It is included here for purposes of understanding standard device drivers.

User written device drivers should define the CSR address in a word in the xxDVR PSECT. This word can then be used in the same way CSR.xx is in standard drivers. This word should be globalized to allow easy change to the CSR address. See section 3.6.

3.3.17 DEV.xx - Pointer to DDB for Unit n

Purpose: Entry in DEVTBL where the DDB pointers for this device start.

File: TBL.MAC

Scope: Global

Example:
```
MOV    DEV.LP(R0),R3    ;Get DDB pointer for unit in R0
```

DEV.xx is equated by the TBL assembly to the address in DEVTBL where the DDB pointers for this device start. If this is a disk device, DEV.xx is equated to the address in UNTCNT where information for this device starts.

If the device has more than one unit, the entries for the additional units will immediately follow the entry for unit 0 at DEV.xx. By using the unit number times two as an offset from DEV.xx the address of the DDB (or UNTCNT entry) can be retrieved for the appropriate unit of a device.

3.3.18 xxDDDB - Address of Unit 0 DDB

Purpose: Point to first DDB for this device

File: TBL.MAC

Scope: Global

Example:
```
MOV    LPDDDB,R0      ;Get pointer to DDB for LP0:
```

xxDDDB is equated by the TBL assembly to the address of the beginning of the first DDB for this device. If the device supports more than one unit, the D DDB's for the additional units are located immediately following the unit 0 DDB. The DDB for a specific unit can be found by indexing from xxDDDB by the unit number times DDS.xx. Note that xxDDDB is a symbol that is equated to the address of the DDB. It is not a location that contains a pointer to the DDB.

3.3.19 LOG$xx - Enter Error Logging

Purpose: Initiate error logging

File: TBL.MAC

Scope: Global

Example: LOG$LP

LOG$xx is equated by the TBL assembly to a unique EMT instruction for each device. If the LOG$xx symbol is executed as an instruction, the driver will be entered at it's ERL$xx entry point. This routine will then log a device specific error using common error logging code within the monitor and return. LOG$xx is only used if the driver logs errors.

3.3.20 WAIT2T - Reenter After Two Clock Ticks

Purpose: Reenter at L3Q entry point after two clock ticks

File: Not applicable

Scope: Global

Example: BIS #QPXCON,WAIT2T

WAIT2T is a global location within the monitor. It is used to specify bits that are to be set in L3QUE after waiting for two clock ticks. This is very useful if a driver needs to wait for less than a second for some condition before continuing execution.

To use WAIT2T, OR the appropriate L3Q bit assigned to the driver into the global location WAIT2T. Only bits in L3QUE may be set. No bits can be set in L3QUE2. After two clock ticks have passed, the specified bit will be set in L3QUE and the driver will be reentered at the specified L3Q entry point.

3.4 SYSTEM MACROS

Several useful macros are defined in the COMMON.MAC, KERNEL.MAC and CHECK.MAC source files that come with the RSTS/E distribution. These macros should be used when writing a device driver to ensure standardization of coding and to help guarantee the integrity of device related monitor tables.

Some of these macros are described in the System Directives Manual. These macros are: TITLE, DEFORG, ORG, TMPORG, UNORG, INCLUDE, .DSECT, .BSECT, .EQUATE, GLOBAL, .BLKW0, JMPX, CALL, CALLX, CALLR, CALLRX, and RETURN. These macros are already described in the System Directives Manual and will not be discussed here.

Several other macros are available for use in device drivers. They are: .BR, .CALLR, PUSH, POP, .ASSUME, REGSAV, REGSCR, REGRES, DEVICE, BUFFER, GETUSR, PUTUSR, SETERR, ERROR, L3QSET, MAP, SPL, SPLC and CRASH. These macros are described in the following section.

3.4.1 .BR - Branch to Following Location

Format:	.BR	loc
Arguments:	loc	Location to branch to
Example:	.BR	DMPFIL

The .BR macro is used when a branch is required to a routine that immediately follows the current location. This macro verifies that the desired location immediately follows the .BR call. If it does not, an assembly error is generated. The .BR macro does not generate any executable code. This macro should be used when a routine would normally fall through into another routine rather than branching to it.

3.4.2 .CALLR - Call Following Routine and Return

Format:	.CALLR	loc
Arguments:	loc	Location to call
Example:	.CALLR	PRTAL

The .CALLR macro performs the same function as the .BR function. It is used when a CALLR is required to the location immediately following the .CALLR usage. This macro should be used when a routine would normally fall through into another routine rather than calling it with the CALLR macro.

3.4.3 PUSH - Push a Value on the Stack

Format:	PUSH	value
or:	PUSH	<list>
Arguments:	value	Value to be pushed on the stack
	list	List of values to be pushed on the stack
Example:	PUSH	<R0,R1>

The PUSH macro pushes a single value or a list of values onto the SP stack. If no argument is provided with this macro call, a zero will be pushed. This macro generates the following code for each value in the argument list:

```
MOV    value,-(SP)
```

3.4.4 POP - Pop a Value from the Stack

Format:	POP	destin
or:	POP	<list>
Arguments:	destin	Destination to pop top of stack into
	list	List of destinations to pop top of stack into
Example:	POP	<R1,R2>

The POP macro pops one or more values from the SP stack into the specified destination. The destination may be any of the standard instruction destination types. If no argument is provided with this macro call, the top element of the stack is discarded. This macro generates the following code for each value in the argument list:

```
MOV   (SP)+,destin
```

3.4.5 .ASSUME - Verify Assumption

Format:	.ASSUME	arg1 cond arg2
Arguments:	arg1	First value
	arg2	Second value
	cond	Assumed relation between arg1 and arg2
Example:	.ASSUME	XRBC EQ 2

The .ASSUME macro is used to verify assumptions, such as the value of certain symbols or the current program location. The condition codes speicfied by "cond" are the same as used in conditional branches. If the condition is specified by "cond" is not true for the two arguments, an assembly error is generated.

3.4.6 REGSAV - Save Registers R0-R5

Format:	REGSAV [inline]	
Arguments:	inline	"INLINE" specifies that the code to save R0-R5 should be generated inline at the current location. If this parameter is not specified, a call will be made to a routine to save the registers.

Example: REGSAV
or: REGSAV INLINE

The REGSAV macro causes registers R0 through R5 to be saved on the SP stack. These values can be restored using the REGRES macro, described in section 3.4.8.

REGSAV defines the local symbols TOS.R0, TOS.R1, TOS.R2, TOS.R3, TOS.R4, TOS.R5, TOS.PC, and TOS.PS. These symbols correspond to the offset from (SP) to the saved values of R0, R1, R2, R3, R4, and R5, as well as PC and the PSW, which will be already on the stack if REGSAV is called during interrupt processing. The INTSAV and INTSVX subroutines call this subroutine before dispatching to the interrupt service routine.

3.4.7 REGSCR - Save Registers Co-Routine

Format: REGSCR

Arguments: None

Example: REGSCR

The REGSCR macro is identical to the REGSAV macro except that the REGRES macro is not used to restore the registers. Instead, the REGSCR subroutine, which the REGSCR macro calls, returns to the caller as a co-routine. When the calling routine exits with an RTS PC, the saved values of R0-R5 will be automatically restored. The caller must be a subroutine that was called with a JSR PC.

REGSCR defines the local symbols TOS.R0 - TOS.R5 to the same values as used by the REGSAV macro (see section 3.4.6). The symbols TOS.PC and TOS.PS are not defined by this macro. Instead, the symbol TOS.RA is defined as the offset from (SP) that corresponds to the saved return address.

3.4.8 REGRES - Restore Registers

Format: REGRES

Arguments: None

Example: REGRES

The REGRES macro is used to restore the registers saved with the REGSAV macro (see section 3.4.6). It restores registers R0 through R5 from the stack.

3.4.9 DEVICE - Define Device Driver Information

Format: DEVICE name,[alt][,<13qbit>]

Arguments:		
	name	Two character device name
	alt	Synonym for device name (optional)
	13qbit	List of L3Q bit names to define for this device (optional)

Example: DEVICE DY,DX,<CON>

This macro is used during the TBL assembly to define the information needed for all device related monitor tables for this device. It is always contained in the xxPRE.MAC file that is used in the TBL assembly (see section 3.7). If the device name synonym conflicts with the name of a standard RSTS/E device or synonym already included in this system, the synonym will be ignored.

If a driver needs to establish additional entry points for performing extended processing for an interrupt service routine, it can do so by defining a bit in L3Q for each additional entry point required. See section 3.2.12 for a discussion of L3Q entry points. If any L3Q bit names are specified, they must follow the rules for L3Q bits specified in section 3.2.12.

If an L3Q bit name is entered in the form <13qbit,apr>, the contents of the location specified by "apr" will be used to load APR5 before dispatching to the L3Q entry point. This allows the use of coordinating drivers. For example, DEVICE KB,TT,<<FMS,FMS>> defines an L3Q bit named QKBFMS and specifies that the value in FMSAP5 should be used to load APR5 before dispatching to the L3Q entry point, FMS$KB. This allows non-phased device drivers to be larger than 4 K-words.

The DEVICE macro defines the following global symbols for the driver: JS.xx, IDX.xx, LOG$.xx, TIM.xx, CSR.xx, xxDDDB and, ALT.xx. See section 3.3 for more information on these symbols.

3.4.10 BUFFER - Allocate/Deallocate Small Buffers

```
Format:  BUFFER GETSML,clear,leave
   or:   MOV    smlptr,R4
         BUFFER RETSML
   or:   MOV    lrgptr,R4
         BUFFER RETURN
```

Arguments:

clear	Number of bytes in the small buffer to preset to 0 before returning from GETSML.
leave	Minimum number of small buffers to leave in the monitor pool after allocating the requested small buffers.
smlptr	Address of small buffer to return.
lrgptr	Address of large buffer header for buffer to return.

```
Example: BUFFER GETSML,,20.      ;Get a small buffer
   or:   MOV    BUFPTR,R4      ;Point to buffer to return
         BUFFER RETSML         ;Return the small buffer
   or:   MOV    BUFPTR,R4      ;Point to buffer header
         BUFFER RETURN         ;Return the large buffer
```

The BUFFER macro is used to allocate and deallocate monitor buffers. The first parameter of the BUFFER call specifies what type of function should be performed, as follows:

GETSML allocates a small buffer from the monitor pool. A pointer to the allocated buffer is returned in R4. The number of small buffers to leave in the monitor pool after allocating this one is specified by the "leave' parameter. This value is used to ensure that the monitor will never get into a situation where it is totally out of small buffers, causing a crash. A driver should typically specify a value of 20 (decimal) for "leave'.

When the driver requests allocation of a small buffer, the monitor performs several checks before granting the request. If the request is fulfilled, GETSML will return with the V-bit clear. If the request is denied, the V-bit will be set.

The "clear' parameter is used to specify the number of bytes to clear in the small buffer after it is allocated. Unless the driver specifically needs the buffer to start with a sequence of zero bytes, this parameter should not be specified or should be specified with a value of zero.

RETSML returns a large buffer to the monitor pool or XBUF. Register R4 points to the buffer to return. This buffer must have been previously allocated from the monitor small buffer pool using the GETSML function.

RETURN returns a large buffer to the monitor pool or XBUF. (See section 3.5.5 for information on allocating large buffers.) Register R4 is a "contorted' pointer to the buffer header of the buffer to return.

A contorted address is used to point to buffers that may be in either the small buffer pool or the extended cache area, XBUF. If the low order five bits of the address are zero, the pointer is the address of a buffer in the small buffer pool. If the low order five bits are non-zero, the pointer is an address in the extended buffer pool that has been rotated left seven bits to ensure that the low order bits are non-zero.

The RETURN function requires that the buffer starts with a buffer header. The buffer header has the following format:

Symbol *Offset*		*Offset*	*Symbol*
	Buffer size	0	BF.SIZ
	Offset to data in buffer	2	BF.OFF
	Link to next buffer	4	BF.LNK
	Size of data	6	BF.CNT

Offset	*Symbol*	*Description*
0	BF.SIZ	This word specifies the size of the buffer in bytes.
2	BF.OFF	This word specifies a byte offset from the beginning of the buffer (ie. BF.SIZ) to the data in the buffer. This offset is almost always 8 for buffers used by device drivers.
4	BF.LNK	This word contains a pointer to the next buffer header in a linked list. If the low order five bits of the pointer are non-zero, the pointer is the "contorted' address of the next buffer in the extended buffer pool, XBUF.
6	BF.CNT	This word specifies the number of bytes of data contained in this buffer.

3.4.11 GETUSR - Get Byte from User Buffer

Format: GETUSR

Arguments:
- R2 — Receives byte from user buffer
- R5 — Pointer to byte in user buffer to be retrieved
- R2 — Receives byte from user buffer

Example: GETUSR

The GETUSR macro is used to fetch a character from the user's buffer. It is normally used to copy data from a user's buffer into small buffers in the SER$xx routine of a driver.

Before the monitor enters a driver at its SER$xx entry point, it maps the buffer pointed to by XRLOC in the user's XRB using APR6 and returns a pointer to this buffer in R5. This pointer is used in the GETUSR call.

Each time the GETUSR macro is executed the character pointed to by R5 is retrieved from the user buffer. The value of R5 is automatically incremented after each use. If an attempt is made to access a character past the end of the portion of the user's buffer that is currently mapped, APR6 will be automatically updated to map the next portion of the buffer.

3.4.12 PUTUSR - Store Byte in User Buffer

Format: PUTUSR

Arguments:
- R2 — Character to store in user buffer
- R5 — Pointer to location in user buffer to receive character

Example:
```
MOV     #40,R2          ;Load a SPACE for PUTUSR
PUTUSR                  ;Put the SPACE in user buffer
```

The PUTUSR macro is used to store information in a user's buffer. It is normally used to transfer data from monitor buffers into the user's data buffer.

Before the monitor enters a driver at its SER$xx entry point, it maps the buffer pointed to by XRLOC in the user's XRB using APR6 and returns a pointer to this buffer in R5. This pointer is used in the PUTUSR call.

Each time the PUTUSR macro is executed the character contained in R2 is transferred to the user's buffer at the location pointed to by R5. The value of R5 is automatically incremented after each use. If an attempt is made to store a character past the end of the portion of the user's buffer that is currently mapped, APR6 will be automatically updated to map the next portion of the buffer.

3.4.13 SETERR - Post Error Code

Format:	SETERR	errcod,destin[,WORD]
Arguments:	errcod	RSTS/E error code to post to the destination.
	destin	Location to receive error code.
	WORD	The error code should be moved as a word.
Example:	SETERR	HNGDEV,@IOSTS

The SETERR macro moves a standard error code to the specified destination. The most common destination for an error code is the first byte of the calling job's FIRQB. This byte is pointed to by the location IOSTS in the monitor and can be referenced using @IOSTS.

If the third argument in the macro is the word "WORD', the error code will be moved to the destination as a word. This clears the high byte of the destination.

The error code is globalized. It will be resolved when the driver is linked with the monitor. The standard error codes are contained in the file ERR.STB and are listed in the *RSTS/E System Directives Manual.*

3.4.14 ERROR - Post Error Code and Exit

Format:	ERROR	errcod
Arguments:	errcod	RSTS/E error code to post to IOSTS
Example:	ERROR	PRVIOL

The ERROR macro is equivalent to using the SETERR macro to post an error to IOSTS, followed by a RETURN. The error code is globalized. It will be resolved when the driver is linked with the monitor. See section 3.4.13 for more information.

3.4.15 L3QSET - Set Bit in L3Q

Format:	L3QSET	bit[,bit2]
Arguments:	bit1:	Bit to set in L3QUE or L3QUE2
	bit2:	Optional bit to set in L3QUE
Example:	L3QSET	QDXCON

The L3QSET macro is used to set bits in the Level Three Queue (L3Q). The specified bit or bits can be any of the standard L3Q bits defined in KERNEL.MAC or they can be the L3Q bits defined for the driver in the DEVICE macro. To specify a bit in L3QUE, bit 15 should be set in "bit1'. If it is not set, the specified bit will be set in L3QUE2. See sections 3.2.12 and 2.2 for a discussion of the Level Three Queue.

3.4.16 MAP - Access Memory Management Registers

Format: MAP {source / PUSH / POP},[OFFSET=offset,]APR=aprval[,CODE][,DATA][,PIC]

Arguments:

source	Value to load into selected APR. If "PUSH' is used, the current value of the specified APR is pushed on the SP stack. If "POP' is used, the top of the SP stack is popped into the specified APR.
offset	Constant offset to add to the value specified in "source".
aprval	APR number
CODE	The I-space register should be used on processors that support I and D space.
DATA	The D-space register should be used on processors that support I and D space.
PIC	Generate position independent code.

Example:
```
MAP     LPDAP5,APR=6,DATA
or: MAP XBUAP5,OFFSET=100,APR=6,CODE,DATA
or: MAP PUSH,APR=6,DATA
or: MAP POP,APR=6,CODE
```

The MAP macro is used to access the memory management registers. It must be used whenever the memory management registers are accessed.

If "source' is a value, that value will be loaded into the specified APR register. If an offset is specified, that offset is added to the value loaded into the APR. If used, the offset value must be a constant.

If the source is specified as "PUSH', the current contents of the specified APR will be pushed onto the SP stack. If "POP' is specified, the contents of the SP stack will be popped and stored in the specified APR.

The only destination that can be specified for the MAP macro is the top of the stack or an APR. If you need to move the contents of an APR to a location other than the top of the stack, the following code can be used:

```
MAP   PUSH,APR=6,DATA
POP   destin
```

The only memory management register available to a driver is APR6. This register can be used to map such areas as the extended buffer area (XBUF), the FIP pool, or user buffers. Note that the monitor maps (using APR6) the user's XRB on a .SPEC, .READ, or .WRITE call before entering the driver.

Since RSTS/E supports I and D space on processors that support it, the MAP macro must specify which space to use. It can specify CODE, DATA, or both. By specifying PIC in the macro call, position independent code will be generated.

3.4.17 SPL - Set Processer Priority

Format: SPL prlvl

Arguments: prlvl Desired priority level

Example:
```
SPL   7
```

The SPL macro sets the processor status word (PSW) to a specific priority. The condition codes and trap bit contained in the PSW are not effected.

3.4.18 SPLC - Set Program Status Word

Format:	SPLC	prlvl
Arguments:	prlvl	Desired priority level
Example:	SPLC	7

The SPLC macro is identical to the SPL macro except that is also clears the condition codes and trap bit in the program status word. See section 3.4.17 for a description of the SPL macro.

3.4.19 CRASH - Crash the System

Format:	CRASH
Arguments:	None
Example:	CRASH

The CRASH macro is used to crash the monitor in an orderly fashion and cause an automatic restart. This macro should obviously be used with great discretion. A driver should always take any other action that is possible rather than crashing the entire system.

This macro generates a .WORD 107. This is a special reserved instruction used to note a software forced system crash. If monitor ODT has been loaded, ODT will be entered for an external breakpoint before the system crashes.

3.5 MONITOR SUBROUTINES

Several subroutines are provided within the monitor for use by device drivers. These subroutines control job status and small buffer usage and perform error logging.

The following subroutines will be described in this section: FREBUF, STORE, FETCH, CLRBUF, BUFFER, RETCHN, INTSAV, INTSVX, IOFINI, IOFINC, IOREDO, RTI3, RETDEV, ERLDVR, QUEUER, FNDJOB, UNLOCK, CLRRSQ, DMPJOB, MAPBUF, GETUMR, RHMADR and RELUMR.

3.5.1 FREBUF - Check Small Buffer Availability

Call:	CALLX	FREBUF,R5,BFQ.xx
Input:	BFQ.xx	Initial small buffer quota
	R1	Pointer to the DDB for this unit
Output:	R4	Random
	C-bit	Set: Insufficient buffers are available. Clear: Sufficient small buffers are available.
Example:	CALLX	FREBUF,R5,BFQ.LP

The FREBUF subroutine checks that enough small buffers are available to allocate at least as many small buffers as specified in the buffer quota, BFQ.xx. The C-bit shows the results of the check. If the C-bit is clear, sufficient buffers were available. Note that FREBUF only checks for availability, it does not actually allocate any buffers.

The request will only succeed if the following conditions are met: (l) There are more than 10 small buffers available in the monitor pool. (2) The device is under its buffer quota. (3) The device is over its quota, the system has at least 20% of its buffers available and this device has not already been allocated 25% of the total small buffers.

If there are not at least 10 small buffers available when FREBUF is called, the caller will be stalled until the system recovers from this critical shortage of buffers.

3.5.2 STORE - Store Character in Small Buffer

Call:	CALLX	STORE,R5,ddbufc+FP
Input:	ddbufc	Offset within the DDB to the small buffer control area for the buffer chain in use. This is DDBUFC, unless the DDB contains more than one buffer chain.
	R1	Pointer to DDB.
	R2	Byte to store in small buffer.
Output:	R4	Random
	C-bit	Set: Store failed due to small buffer shortage Clear: Store succeeded
Example:	CALLX	STORE,R5,DDBUFC+FP

The STORE subroutine is used to store characters in a small buffer chain. Each small buffer chain controlled by the driver has a small buffer control area in the DDB (see section 2.5.1.3). The STORE subroutine requires an argument which is the fill pointer for the buffer chain in the DDB.

When the store subroutine determines that the current small buffer is full, it will attempt to allocate a new one. If there are sufficient small buffers available in the pool, this allocation succeeds. The new small buffer is linked into the buffer chain and the character is stored in the new small buffer.

If there are not enough small buffers in the pool to honor the request, the allocation will fail and the character will not be stored. In this case the driver should stall the job by exitting through IOREDO if it is in the I/O service routine, or post an error or take device specific action if it is in the interrupt service routine.

3.5.3 FETCH - Get Character from Small Buffer

Call:	CALLX	FETCH,R5,ddbufc+EP
Input:	ddbufc	Offset within the DDB to the small buffer control area for the buffer chain in use. This is DDBUFC, unless the DDB contains more than one buffer chain.
	R1	Pointer to DDB
Output:	R2	Character fetched (if successful)
	R4	Random
	C-bit	Set: Fetch failed (no more data in small buffers) Clear: Fetch succeeded
Example:	CALLX	FETCH,R5,DDBUFC+EP

The FETCH subroutine retrieves a character from the specified small buffer chain (see section 2.5.1.3). The characters are stored in a first-in first-out basis. If there are no characters pending in the small buffer chain, FETCH will return with the C-bit set to signify failure. If a small buffer becomes empty, it will be returned to the small buffer pool.

This subroutine uses the BUFFER macro to return small buffers to the monitor pool as they become empty.

3.5.4 CLRBUF - Return All Small Buffers

Call:	CALLX	CLRBUF,R5,ddbufc+EP
Input:	ddbufc	Offset within the DDB to the small buffer control area for the buffer chain in use. This is DDBUFC, unless the DDB contains more than one buffer chain.
	R1	Pointer to DDB
Output:	R4	Random
Example:	CALLX	CLRBUF,R5,DDBUFC+EP

The CLRBUF subroutine returns all the small buffers in the specified small buffer chain. Any data stored in the small buffers will be lost. This subroutine is normally used to flush any pending input when a device is closed.

3.5.5 BUFFER - Allocate a Large Buffer

Call:	CALLX	BUFFER,R5,start
Input:	start	Global symbol corresponding to the buffer pool to start searching in for free space. The global symbols can be MONPOL, for the small buffer pool, LRGPOL, for the large buffer pool, and EXTPOL, for the large buffer pool that is permanently mapped for DMA transfers.
	R1	Requested buffer size (in bytes)
	R2	Number of buffers to leave in the monitor's pool (MONPOL)
Output:	R1	Size of buffer allocated. If the requested buffer size was not mod 40_8 from MONPOL or mod 100_8 from LRGPOL or EXTPOL, its size will be rounded up appropriately.

	R4	Pointer to the buffer allocated. If the least significant five bits are zero, the pointer is to a buffer in MONPOL. If not, the pointer is a "contorted" address (see section 3.4.10) into LRGPOL or EXTPOL and APR6 has been mapped to the base of the buffer. In either case, a null buffer header has been built at the beginning of the buffer (see section 3.4.11).
	C-bit	Set: The request failed due to lack of buffer space. Clear: The request succeeded.

Example: CALLX BUFFER,R5,LRGPOL

The BUFFER subroutine is used to allocate a buffer which is typically more than 16 words in size. It is allocated from the monitor pool or XBUF space unless a specific starting point in XBUF is specified.

Use the BUFFER (RETURN) macro to return a buffer allocated with this subroutine. See section 3.4.10 for more information.

3.5.6 RETCHN - Return All Large Buffers in a Chain

Call:	CALLX	RETCHN	
Input:	R4	Address of first buffer in the chain. If the least significant five bits of the address are non-zero, the address is a "contorted" pointer to a buffer in XBUF space. See section 3.4.10 for more information on contorted address poiners.	
Output:	R3	Random	
	R4	Random	
Example:	MOV	BUFPTR,R4	;Point to first buffer in chain
	CALLX	RETCHN	;Return all buffers in chain

The RETCHN subroutine returns all large buffers in a chain to their appropriate buffer pool. The individual buffers may belong to any of the buffer pools. Each buffer is headed by a standard buffer header (see section 3.4.10). The buffers are linked using the BF.OFF word in the headers.

3.5.7 INTSAV - Enter Interrupt Service Routine

Call:	CALLX	INTSAV,R5
	+	1200
	+	INT$xx
Input:	1200	The interrupt service routine is mapped using APR5, therefore its virtual base address corresponds to 120000_8. The number 1200_8 corresponds to this address.
	INT$xx	This word specifies the address of the interrupt service routine within the driver.
Output:	All registers preserved.	
Example:	CALLX	INTSAV,R5
	+	1200
	+	INT$LP

The INTSAV subroutine is used to map the interrupt sevice routine and save all registers before entering the interrupt service routine to process a device iterrupt. The call to INTSAV must be contained in the first four words of the xxDINT PSECT. In most cases the xxDINT PSECT contains only this call.

When the device interrupt is vectored to xxDINT, the call to the INTSAV subroutine will save R0 through R5, map the driver to a base address of 120000_8 using APR5 and enter the interrupt service routine at the driver's INT$xx entry point. See section 3.2.7 for information on the INT$xx entry point.

3.5.8 INTSVX - Enter Secondary Interrupt Routine

Call:	CALLX	INTSVX,R5
	+	xxDAP5
	+	entry
Input:	xxDAP5	This global symbol points to the APR5 base value specified in the INTSAV subroutine call.
	entry	This word specifies the entry point within the interrupt service routine to transfer to.
Output:	All registers preserved.	
Example:	CALLX	INTSVX,R5
	+	LPDINT+4
	+	INTLP2

The INTSVX subroutine is identical to the INTSAV subroutine, except that the first parameter is a pointer to the value to load into APR5, whereas the first parameter in the INTSAV subroutine is the actual value.

The INTSVX subroutine is used in device drivers that support multiple interrupt vectors and that choose not to combine all the vectors into one entry point (see section 3.1.1.2). Since the preferred method for supporting multiple vectors is to use only one entry point, the INTSVX subroutine is seldom needed.

3.5.9 IOFINI - I/O Finished

Call:	CALLX	IOFINI,R5,JS.xx
Input:	BFQ.xx	Initial small buffer quota.
	count	Minimum number of small buffers available in quota to allow IOFINI.
	JS.xx	JBSTAT bit to set to make the job runnable.
	R1	Points to the DDB for this unit.
Output:	R4	Job number times 2
Example:	CALLX	IOFINI,R5,JS.LP

The IOFINI subroutine is used by an interrupt service routine to signify that the associated job should be made runnable. The bit corresponding to JS.xx (which was set in JBWAIT to stall the job) will be set in the job's JBSTAT word. The next time the scheduler is envoked the job will be eligible for execution. If there are no other jobs currently executing, the scheduler will be envoked immediately. The call to IOFINI is normally followed by the RETURN that exits from the interrupt service routine.

Note that, if the job is marked for I/O redo (JFREDO set in JDFLG), the driver will be reentered at its SER$xx entry point when the job is scheduled for execution. The user program will not be executed until the I/O service routine exits to IOEXIT. See sections 3.2.5, 3.3.15, and 3.5.11.

3.5.10 IOFINC - I/O Conditionally Finished

Call:	CALLX	IOFINC,R5<BFC.xx-count,JS.xx>
Input:	BFQ.xx	Initial small buffer quota.
	count	Minimum number of small buffers available in quota to allow IOFINI.
	JS.xx	JBSTAT bit to set to make the job runnable.
	R1	Points to the DDB for this unit.
Output:	R4	Job number times 2
Example:	CALLX	IOFINC,R5,<BFQ.LP-4,JS.LP>

The IOFINC subroutine performs the same function as IOFINI, but only if a specified number of small buffers are "guaranteed" to be available to the driver. It is useful for continuing execution of the job (or performing an I/O redo) before the driver's small buffer chain is totally empty. This tends to maintain continuous output on the device.

The INFINC subroutine checks the number of small buffers currently allocated to the device. If the device can allocate the number of small buffers specified by "count" and not exceed its buffer quota, IOFINI will be called to set the specified bits in JBSTAT. See section 3.5.9.

3.5.11 IOREDO - Stall for I/O Redo

Call:	JMPX	IOREDO
Input:	None	
Output:	None. This is an exit routine.	
Example:	JMPX	IOREDO

The IOREDO is used when the I/O service routine (SER$xx) needs to be reentered at a later point. This is normally caused by a need to wait for small buffer availability. If the I/O service routine exits to this routine, the JFREDO bit is set in the job's JDFLG word and the job is stalled.

When the interrupt service routine exits through IOFINI the driver will be made resident, if necessary, and reentered at its SER$xx entry point. The C-bit will be set and the V-bit will be cleared to signify that this is an I/O redo.

This exit should be taken by the I/O service routine any time it needs to wait for something that the interrupt service routine has control over. The normal use of this exit is to wait until the interrupt service routine needs the I/O service routine to move data between small buffers and the user's buffer.

3.5.12 RTI3 - Exit and Check Level Three Queue

Call:	JMPX	RTI3
Input:	None	
Output:	None. This is an exit routine.	
Example:	JMPX	RTI3

The RTI3 routine is used as the normal exit point for the special service entry point, SPC$xx. It restores registers R0 through R5 and checks L3Q for any pending requests. If requests are found, it dispatches to the appropriate routine to service the request. Many of the exits from other entry points in a driver enter RTI3 automatically before returning to the caller.

5.3.13 RETDEV - Return an Open Device

Call:	CALLX	RETDEV
Input:	R1	Pointer to DDB for this unit
	R5	Pointer to job's IOB entry for this channel
Output:	R2	Random
Example:	CALLX	RETDEV

OThe RETDEV subroutine is used in the open routine (OPN$xx) of a device driver if the open fails. The RETDEV subroutine cancels all setup for the open that was previously done by the monitor. RETDEV must be called any time an open fails for device dependent reasons.

3.5.14 ERLDVR - Enter Error Log Entry

Call:	CALLX	ERLDVR,R5
	.BYTE	errcod
	.BYTE	ddbsiz
	.BYTE	offset,regcnt
	...	
	.BYTE	0,0
Input:	errcod	Error code to enter in error log
	ddbsiz	Size of DDB
	offset	Offset from CSR to first register to log
	regcnt	Number of registers to log
	R1	Pointer to DDB
	R3	Pointer to CSR for this device
Output:	R0	Pointer to timeout flag in error log table
	R1	Random
	R2	Pointer to terminator in error log table
	R3	Random
	R5	Pointer to error logging control table
Example:	CALLX	ERLDVR,R5
	.BYTE	377
	.BYTE	DDS.AR
	.BYTE	0,8.
	.BYTE	0,0

The ERLDVR subroutine is used to log errors. The TBL assembly defines a macro instruction, LOG$xx, which can be used to initiate eror logging. When the LOG$xx instruction is executed, the driver's error logging routine is entered at ERL$xx. The code at ERL$xx then calls the ERLDVR routine and passes it a list of information to log.

The first argument to ERLDVR is an error code to identify the type of error. The current error codes are defined in KERNEL.MAC as ERC$xx. In addition to the codes defined by KERNEL, code 55 is used for unrecognized messages and code 56 is used for SHUTUP requests. A user written driver can use any code not already in use up to a value of 61.

The second argument specifies the size of the DDB, in bytes. The entire DDB pointed to by R1 will be copied to the error log.

The following arguments are a list of byte pairs. The first byte in each pair is an offset from the beginning of the CSR (pointed to by R3) to the first device register to copy into the error log. The second byte specifies the number of registers to copy.

If the device supports the RH70 massbus interface, there can be at most three byte pairs. Otherwise, there can be up to four byte pairs. The last byte pair is followed by a byte pair of zeroes. The maximum number of byte pairs includes the special entries described below.

ERLDVR returns a pointer to a location to use as a timeout flag in R0. The pvalue at this location is returned as zero. If the error is due to a timeout, this value should be made non-zero.

ERLDVR also returns a pointer to the end of the control table. This pointer can be used to modify the error log to include information about a device that is not normally available in its device registers. This information is entered using a special routine in the driver. A pointer to this special routine is appended to the control table, as follows:

ERLDVR returns with a pointer to the end of the control table in R2. Store a number in the byte pointed to by R2 that specifies the number of words that will be supplied by the special routine. Store a -4 in the following byte. Store the address of the special routine in the following word. Follow this word with a -1.

When the monitor error logging routine is entering information in the error log message, it will call the specified special routine. R5 will point to the next available entry in the message buffer. The special routine then loads each succeeding word in the message buffer with the appropriate information and returns with R5 pointing to the next available word in the message buffer.

The following example shows the use of this special capability:

```
ERL$DX::  CALLX     ERLDVR,R5          ;Call error logger
            .BYTE   ERC$DX             ;Error code is DX error
            .BYTE   DDS.DX             ;Size of DDB for DX driver
            .BYTE   0,2                ;Offset from CSR=0, Registers=2
            .BYTE   0,0                ;End of table
          COM       (R0)               ;Set timeout indicator
          MOV       #-4*400+1,(R2)+    ;Set word count and -4
          MOV       #ERLDXM,(R2)+      ;Point to special data log routine
          MOV       #-1,(R2)           ;Reterminate the control table
          RETURN                       ;Return from error log routine
;+
; ERLDXM - Special error logging data
;
;         R3 -> CSR
;         R5 -> Error log message buffer
;
;         ...
;
;         RETURN
;
;         R5 -> New position in error log message buffer
;-
ERLDXM:   MOV       #17,(R3)           ;Function: READ ERROR REGISTER
   10$:   TSTB      (R3)               ;Check low byte of CSR
          BPL       10$                ;Wait for TR to set
          MOV       2(R3),(R5)+        ;Store error register in error log
          RETURN                       ;Return from special error logger
```

3.5.15 QUEUER - Enter Item in Queue

Call:	CALLX	QUEUER,R5,root
Input:	root	Address of location that contains a pointer to the first element in the queue.
	R4	Pointer to the link word of the item to enter into the queue.
Output:	R4	Preserved
	C-bit	Set: This item is the first in the queue
		Clear: This item is not the first in the queue
Example:	MOV	#ITEM+LINK,R4
	CALLX	QUEUER,R5,DXQROT

The QUEUER subroutine is used to enter an item into a specified queue. The queue is ordered first-in, first-out. The first element of the queue is pointed to by a root word. The link word in the last entry in the queue is zero.

To use the queuing mechanism, the driver needs to set aside a word to use as the root of the queue. This word is typically contained in the driver's xxDCTL PSECT (see section 3.1.1.3). Any item that is to be entered into the queue must contain a word to use as a link to the following entries in the queue. The typical structures Oused for queue elements are DDBs, the user's OXRB (contained in WRK), and DSQs.

To remove the top element from the queue, move the contents of the link word of the first element in the queue to the root word.

3.5.16 FNDJOB - Force a Job Into Memory

Call:	CALLX	FNDJOB	
Input:	R0	Job number times 2 of job to find	
	R3	L3Q bits to set on residency if non-resident	
	R5	Pointer to user's buffer (unmapped)	
Output:	R0	Random	
	R2	Physical location of job, bits <16:21>	
	R3	Physical location of job, bits <0:15>	
	R5	Pointer to user's buffer (mapped by APR6)	
	C-bit	Set: The job was non-resident and is coming into memory.	
		Clear: The job is resident.	
Exit:	Non-skip	A fatal error occurred when loading the job image, its associated RTS or LIBs.	
	Skip	The job image and its associated RTS and LIBs are OK.	
Example:	MOV	#1*2,R0	;Force job 1 into memory
	MOV	#QDXCON,R3	;Set QDXCON in L3Q on residency
	CALLX	FNDJOB	;Get the job into memory
	BR	FTLERR	;Error in residency attempt
	BCS	WAIT	;Wait for residency
	...	;Continue with job resident	

The FNDJOB subroutine is used to ensure residency of a job before the driver attempts to perform a DMA transfer to or from a buffer in the job image. If the specified job is already in memory, the value passed in R5 will be used to map the job's buffer using APR6.

If the job is resident when FNDJOB is called, the job will be locked in memory (LCK will be set in M.CTRL in the job's MCB). FNDJOB should only be called once for each time the job is unlocked (see section 3.5.17). If any bits are set in M.CTRL when FNDJOB is called (including the LCK bit set by FNDJOB), the job will be considered to be non-resident.

The FNDJOB subroutine has two possible return points. If an error occurs while loading the job, its runtime system or any of its libraries, FNDJOB will return to the location immediately following the call. This location should contain a branch to a routine to handle a swap/load error. If no error occurs, FNDJOB will skip the word immediately following the call and return to the following location.

If the specified job is not already resident, the specified bits will be set in L3Q when the job is made resident. This allows the driver to exit and continue processing (at an L3Q entry point) after the job becomes resident. If a driver has to wait for job residency, the FNDJOB routine should be called again after the job is resident so that swap and load errors can be checked and so that the user's buffer can be mapped.

3.5.17 UNLOCK - Unlock a Job's Memory

Call:	CALLX	UNLOCK	
Input:	R0	Job number times 2	
Output:	R2	Random	
Example:	MOV	JOBNUM,R0	;Get desired job number *2
	CALLX	UNLOCK	;Unlock it

The UNLOCK subroutine is used to mark a job's memory as being available for swapping out. It is used in synchronous I/O drivers that must control the residencey of a job while performing I/O.

This subroutine only marks the job as being available for swapping out. It will not be actually swapped unless the memory is needed for another job.

3.5.18 CLRRSQ - Clear Residency Quantum

Call:	CALLX	CLRRSQ	
Input:	R1	Pointer to Job Data Block (JDB) for job	
Output:	None		
Example:	MOV	JOBTBL(R0),R1	;Get JDB pointer for job
	CALLX	CLRRSQ	

The CLRRSQ routine clears the residency quantum for the specified job. If any other job is waiting for residency, the specified job will be swapped out to make room.

3.5.19 DMPJOB - Dump the Current Job

Call:	CALLX	DMPJOB
Input:	None.	
Output:	R0	Random
	R1	Random
	R2	Random
	R5	Random
	APR6	Random
Example:	CALLX	DMPJOB

The DMPJOB routine stops the execution of the currently executing job. It clears its residency quantum and unlocks its memory, making it eligible to be swapped out if necessary.

3.5.20 MAPBUF - Map a Buffer

Call:	CALLX	MAPBUF	
Input:	R4	Contorted address	
Output:	R3	Mapped address	
	APR6	Altered if address is in non-monitor buffer	
Example:	MOV	BUFPTR,R4	;Point to the large buffer
	CALLX	MAPBUF	;Map the buffer and point to it

The MAPBUF subroutine ensures that the large buffer pointed to by R4 is properly mapped by memory management. If the least significant five bits of the pointer are zero, it points to a monitor buffer; no mapping is necessary. If the least significant five bits are non-zero, it is a "contorted" pointer to a buffer in XBUF. This buffer will be mapped using APR6. The original pointer will be modified to point to the base of the mapped buffer.

3.5.21 GETUMR - Get a Unibus Mapping Register

Call:	CALL	@GETUMR
Input:	R0	Negative of word count to transfer
	R1	MSB of 22-bit bus address
	R2	LSB of 22-bit bus address
	R4	Pointer to DDB or DSQ
Output:	R1	Most significant two bits of unibus address for use in DMA transfer. Unchanged if request failed or processor does not support UMRs.
	R2	Least significant 16 bits of unibus address for use in DMA transfer. Unchanged if request failed or processor does not support UMRs.
	C-bit	Clear: UMR was allocated. Set: Insufficient UMRs were available or UMRs are not supported. No UMRs were allocated.
Example:	GLOBAL	GETUMR
	CALL	@GETUMR

Processors with 22-bit addressing can address up to 4 megabytes of physical memory. Since unibus device controllers that do DMA transfers only support 18-bit addressing, a mechanism is necessary for mapping the 18-bit addresses into 22-bit adresses during DMA transfer. This mechanism is provided by the unibus mapping registers (UMRs).

All device drivers that perform DMA transfers directly to or from a user buffer need to initialize unibus mapping registers when running on processors that support unibus mapping. The GETUMR routine provides the capability for doing this initialization in a common way on all processor types.

If GETUMR is called on a processor that does not support unibus mapping, it will return immediately. If the processor supports unibus mapping, GETUMR will allocate the number of unibus mapping registers necessary to map the buffer for the transfer. Note that the call to GETUMR uses indirect addressing. This allows the code for GETUMR to be replaced by a RETURN and the remaining code to be converted to small buffers on processors that do not support unibus mapping.

If allocation of a UMR is requested when an insufficient number of UMRs are available, the request will fail (shown by the C-bit). The driver should stall the calling job (by exiting to IOREDO) and wait for a UMR to become available.

If a GETUMR request fails, the monitor will enter the driver at its UMR$xx entry point when a UMR becomes available. Note that an entry at the driver's UMR$xx entry point does not guarantee that a UMR is available. All drivers that use UMRs will be entered if a UMR becomes free. A previously entered driver may have allocated the UMR by the time a later driver is entered.

Once allocated, a UMR should be released as soon as possible. The RELUMR routine releases UMRs allocated by a previous GETUMR or RHMADR call (see section 3.5.22 and 3.5.23).

A portion of the large buffer pool is permanently mapped by UMRs. This permanently mapped pool is named EXTPOL. If I/O takes place using a large buffer from this pool, no special settup of UMRs is required. It has already been done by the monitor.

The following code will convert a contorted buffer address (contained in R1) to an 18-bit address suitable for loading into a device register:

```
ASHC    #-7,R1          ;Convert contorted address to APR value
ADD     EXTPOL,R1       ;Adjust for base of EXTPOL buffer pool
ASHC    #6,R0           ;Convert to 18-bit unibus address
                        ;MSB is in R0, LSB is in R1
GLOBAL  <EXTPOL>
```

The GETUMR routine is only used for devices that do not use the RH70 and RH11 massbus interfaces. If the device uses the RH70 or RH11, the RHMADR routine (see section 3.5.22) should be used instead of GETUMR.

3.5.22 RHMADR - Initialize Unibus Mapping for RH70 or RH11

Call:	CALLX	#RHMADR
Input:	R0	Negative of word count to transfer
	R1	MSB of 22-bit bus address
	R2	LSB of 22-bit bus address
	R3	Address of RH70 CSR
	R4	Pointer to DDB or DSQ
Output:	Extended bus address loaded into RH70 or RH11	
	C-bit	Clear: UMR was allocated.
		Set: Insufficient UMRs were available. None were allocated.
Example:	GLOBAL	RHMADR
	CALL	@RHMADR

The RHMADR routine is identical to the GETUMR routine (see section 3.5.21), except that it is used for devices which use the RH70 and RH11 massbus interfaces.

3.5.23 RELUMR - Return a Unibus Mapping Register

Call:	CALLX	@RELUMR
Input:	R4	Pointer to DDB or DSQ
Output:	None	
Example:	.GLOBL	RELUMR
	CALL	@RELUMR

The RELUMR routine returns the unibus mapping registers (UMRs) allocated by the GETUMR or RHMADR routines. Register R4 must contain the same pointer to the DDB or DSQ that was used when the UMR was originally allocated.

3.6 CSR AND VECTOR ASSIGNMENT

The CSR and vector addresses for all standard device drivers are stored in tables within the monitor. These tables may be modified by the HARDWARE option of INIT if any CSR or vector addresses need to be changed. All access to the CSR or vector information for a standard device driver is made through these tables.

The HARDWARE option uses an additional table of information about every possible standard device when accesssing the CSR the vector tables. Since this table obviously does not contain information about nonstandard devices the HARDWARE option cannot be used to change information about nonstandard devices. A different approach is needed for these devices.

The suggested approach for defining the CSR addresses for a user written device driver is to allocate a word in the xxDVR PSECT for each unit of the device. The CSR address for the associated device can be specified in each of these words. The contents of these words can then be patched if a particular device is at a nonstandard CSR address.

All access to the device registers should be made using the CSR addresses defined in the driver. For example:

```
xxCSR::   .WORD    170400              ;Table of CSR addresses for "xx"
          .WORD    170400+20
            ...
SER$xx::  MOV      xxCSR(R0),R3        ;Get CSR address for this unit
          MOV      (R3),R4             ;Access CSR for this unit
            ...
```

The suggested approach for defining vector addresses for a user written driver is very similar to that used for CSR addresses. Allocate two words in the xxDVR PSECT for each unit of the device. The first word contains the vector address. The second word contains the desired PSW and unit number. The unit number is contained in the least significant four bits of the PSW value.

The driver will be entered at its ASN$xx entry point when the device is first opened or assigned. At this time the vector locations in the monitor can be initialized using the table of information. The following example shows one implementation of this approach:

If the device only requires one vector location:

```
xxVEC::   .WORD    170                 ;Vector address for device
          .WORD    PR5+0               ;PSW and unit number
          ...
                                       ; ASSIGN entry point
ASN$xx::  PUSH     R2
          MOV      xxVEC,R2            ;Get pointer to vector table
          MOV      #xxDINT,(R2)+       ;Load vector with xxDINT address
          MOV      xxVEC+2,(R2)        ;PSW and unit number in 2nd word
          POP      R2
          ...      ;Continue with ASN$xx code
```

If the device requires more than one vector location:

```
                ; Table of vector locations, PSW values and unit numbers
                ; Each vector requires two words. The table is terminated
                ; by a 0 word.
xxVEC::         .WORD   170             ;Unit 0
                .WORD   PR5+0
                .WORD   174             ;Unit 1
                .WORD   PR5+1
                .WORD   0               ;Terminate table with 0 word
                ...
                ; ASSIGN entry point
ASN$xx::        PUSH    <R2,R3>         ;All registers must be preserved
                MOV     #xxVEC,R2       ;Point to base of vector table
2$:             MOV     (R2)+,R3        ;Get pointer to vector location
                BEQ     4$              ;End of table
                MOV     #xxDINT,(R3)+   ;Set vector to xxDINT
                MOV     (R2)+,(R3)      ;PSW+unit in 2nd word of vector
                BR      2$              ;Repeat for each vector in table
4$:             POP     <R3,R2>
                ...                     ;Continue with ASN$xx code
```

3.7 INSTALLING THE DRIVER

Installing a user written device driver in RSTS/E requires four steps: (l) Assemble the driver (2) Assemble TBL to include the driver information (3) Link all portions of the monitor (4) Build the monitor with SILUS.

To assemble the driver, type the following, where xxDVR.MAC is the name of the source file for the driver:

```
****$RUN MACRO
*****xxDVR,xxDVR/C=COMMON,KERNEL,xxDVR
```

The driver source file (xxDVR.MAC) must contain the definitions for the symbols STS.xx, FLG.xx, SIZ.xx and BUF.xx (see section 3.3). It must contain the interrupt mapping code in the xxDINT PSECT (see section 3.1.1.2). It must also contain the driver code in the xxDVR PSECT (see section 3.1.1.1).

Assembling TBL requires a prefix file that defines the information needed by TBL to build monitor tables to include the user written driver. This file, xxPRE.MAC, must contain the following code (see sectin 3.3 and 3.4.9):

```
DEVICE xx,[alt][,<13qbits>]
CNT.xx= number of units
DDS.xx= size of DDB in bytes
CCC.xx= 1 if device is^C interruptable
        0 if not
BFQ.xx= buffer quota (optional)
HOR.xx=width (optional)
SLP.xx= 1 if driver should be notified of conditional SLEEP requests
        0 if not (optional)
UMR.xx=1 if driver uses unibus mapping registers (UMRs)
        0 if not (optional)
```

The assembly of the TBL.MAC file in the SYSGEN.CTL command file must be changed to include the xxPRE file for the new driver. The link of the monitor must also be changed to include the new driver code.

The assembly of TBL.MAC is changed by inserting the name of the prefix file for the driver (xxPRE.MAC) immediately before the TBL.MAC file in the assembly line. The resulting command line is changed

from:

```
$R MACRO.SAV
TBL,TBL/C=IN:COMMON,KERNEL,DK:CONFIG,IN:CHECK,TBL
```

to:

```
$R MACRO.SAV
TBL,TBL/C=IN:COMMON,KERNEL,DK:CONFIG,IN:CHECK,DK:xxPRE,IN:TBL
```

The procedure for linking the monitor (RSTS.SAV) is changed by inserting the name of the driver object module (xxDVR.OBJ) into the link command stream immediately after the terminal driver root module (TTDINT). The resulting command line is changed

from:

```
$R LINK.SAV
RSTS/Z,RSTS/A/W,RSTS=TBL,$ERR.STB/X/B:O/E:#12500/U:#1000/I/C
TTDINT/C
IN:RSTS
MORBUF
...
```

to:

```
$R LINK.SAV
RSTS/Z,RSTS/A/W,RSTS=TBL,$ERR.STB/X/B:0/E:#12500/U:#1000/I/C
TTDINT/C
xxDVR/C
IN:RSTS
MORBUF
...
```

The remainder of the SYSGEN.CTL file remains unchanged. The following EDIT commands will make all the required modifications to the SYSGEN.CTL file. When the sysgen process prints "If you have any editing changes, stop now and do them then type .R SYSBAT." Stop and type the following commands (change "xx" to the appropriate driver name):

```
.R EDIT.SAV
#EBSYSGEN.CTL$FTBL.OBJ$0AV$$
TBL.OBJ,TTDINT.OBJ,TTDVR.OBJ/DE:NOWARN
#K$$
*F,TBL/C$FCHECK,$V$$
TBL,TBL/C=IN:COMMON,KERNEL,DK:CONFIG,IN:CHECK,TBL
*IDK:xxPRE,IN:$V$$
TBL,TBL/C=IN:COMMON,KERNEL,DK:CONFIG,IN:CHECK,DK:xxPRE,IN:TBL
*FIN:RSTS$V$$
IN:RSTS
*IxxDVR/C$$
$$
*EX$$
.R SYSBAT
```

Once the monitor is built, it is installed and used in the normal fashion. See the RSTS/E System Generation Manual for more information on the sysgen procedure. Also see section 3.8 for information on startup options for debugging.

If the monitor needs to be rebuilt during the debugging process, the monitor files do not need to be reassembled unless the prefix file for TBL (xxPRE.MAC) is changed. Only the commands in SYSGEN.CTL following the assemblies need to be repeated to rebuild the monitor.

Note that the SIL that is being rebuilt cannot have the same name as the currently installed SIL. Install a different SIL while rebuilding the SIL used for developing the device driver.

3.8 DEBUGGING A DEVICE DRIVER

RSTS/E provides two versions of ODT: one for INIT and one for use during timesharing. Since INIT does not support any user written drivers, its debugger is not normally required. On the other hand, the ODT available from the monitor is invaluable for debugging user written drivers.

To load ODT under INIT type "ODT" at the OPTION question. INIT will immediately enter ODT and display a prompt of a letter followed by an underline (eg. K_). The letter signifies the memory management mode currently in effect (see M command, section 3.8.4), as follows:

- K Kernel. I & D space is not in use or is not available.
- I Kernel I-space.
- D Kernel D-space.
- U User I-space.
- X User defined mapping (see control table Y in section 3.8.3).

If for some reason the system crashes before reaching the OPTION question, ODT can be envoked by setting the console switch register to the RAD50 equivalent of "ODT". ODT will then be entered immediately after starting INIT.

The most useful debugging tool for user written drivers is the monitor version of ODT. Monitor ODT is envoked by the "Memory allocation changes" option of the START and DEFAULT commands in INIT. (ODT.SYS is contained on the system generation kit and must be copied to [0,1]ODT.SYS before being used.

If invoked by the START command, ODT will only be loaded for this timesharing session. If envoked by the DEFAULT command, ODT will be loaded every time timesharing is started.

The ODT option requires specification of the address where ODT will be loaded during timesharing. It is normally loaded immediately after the monitor and resident runtime system, but can be loaded at other locations if there is a conflict with resident runtime systems or libraries. ODT requires 3 K-word of memory.

Once loaded, monitor ODT will be entered any time a ↑P is typed on the console terminal or whenever a breakpoint is encountered in a monitor routine. (The character used to enter ODT from the console can be changed to something other than ↑P by changing location $$ODTP to the desired character code.) A breakpoint will also be entered before the system crashes.

When entered by ↑P or a crash, ODT will assume it was entered with an external breakpoint and will print "BEr,a", where "r" is an optional relocation register and "a" is the address where the breakpoint occurred. All ODT commands may be used at this point.

The remainder of this section will describe the commands available from monitor ODT that are applicable to debugging device drivers. In many cases the monitor ODT commands are identical to the ODT debugger described in the *IAS/RSX-11 ODT Reference Manual* supplied in the RSTS/E documentation kit. An understanding of the standard ODT debugger is assumed for the following discussion.

3.8.1 Notation

All values in ODT are considered to be unsigned, octal numbers in the range 0-177777_8. If a decimal point is appended to a number, it is considered to be a decimal value. If the number is preceded by a minus sign, the two's complement of the number is used. If a value greater than 177777_8 is entered, only the least significant 16 bits of the value will be used.

There are three special characters which can be used interchangably with numeric values. These special characters have the following meanings:

C Equivalent to the value stored in control register $C.

Q Equivalent to the last quantity displayed. This quantity is stored in control register $Q.

. Equivalent to the address of the last or currently open location.

In the descriptions which follow, several abbreviations are used for specific values in commands or expressions. These abbreviations have the following meanings:

v Represents a 16 bit value. This value can be a number or an expression.

r Represents a number in the range 0-7. These numbers are used to specify register numbers for control registers.

n Represents a number in the range 0-7. These numbers are used to specify relocation registers and control table entries.

$n Represents the value of processor registers R0-R7.

$c Represents one of the control register names (see section 3.8.2)

$nt Represents an element of one of the control tables (see section 3.8.3). "n" specifies which element of the table, "t", to use.

Any value other than a register number or control table entry can be an expression. Expressions consist of values or constant registers combined by any of the following operators:

+ Add the following argument to the preceding value.

- Subtract the following argument from the preceding value.

* Multiply the preceding value by 50_8 and add the following argument.

! Shift the preceding value the number of bits specified by the following argument. If the following argument is positive, the preceding value will be shifted to the left, otherwise it will be shifted to the right.

& Perform a logical AND between the preceding value and the following argument.

<sp> A space is treated like a "+" unless it is preceded or followed by another operator.

3.8.2 Control Registers

Control registers are used to define or access information used by ODT. To access information in a control register, use the register name as a value in an open command. For example, $C/ will display the current value of the constant register.

To change the value in a control register, open the register with the "/" command, enter a new value and press RETURN. For example, the following sequence will change the constant register from a value of 0 to a value of 2:

$C/ 000000 2<CR>

The control registers have the following meanings:

$A This register is used as the argument in the search commands (see the W, N and E commands). Its value can be explicitly set using this register or implicitly set through the W, N or E commands.

$C This register contains the value used as the constant register, "C" (see section 3.8.1). This register is initialy loaded with ODT's starting address.

$F This register specifies the print format for addresses displayed by ODT. If it is set to zero (default), all addresses will be displayed relative to a relocation register, if possible. If it is non-zero, all addresses will be displayed as absolute values.

$H This register is used to specify the ending address for commands that access a range of locations in memory. These commands include W, N, E, L and F. The value of this register can be explicitly set or implicitly set through in the W, N, E, L or F commands.

$L This register is used to specify the starting address for commands that access a range of locations in memory. These commands include W, N, E, L and F.

$M This register is used as a mask in the search commands (see the W, N and E commands). When a search command is executed, the contents of the argument register ($A) and the value from memory will each be ANDed with this mask before being compared. The value of this register can be explicitly set or implicitly through the W, N, E, or L commands.

$P This register specifies the priority ODT should run at. If it is set to 377_8, ODT will run at the same priority that was in effect at the time of the breakpoint. If it is not set to 377_8, the register's contents will be used for ODT's run priority. The priority should range from 0-7 or be set to 377_8.

$Q This register is set to the last value displayed by ODT.

$S This register is the processor status word (PSW) at the time of the last breakpoint. It is loaded with the current PSW each time a breakpoint occurs. It may be modified to change the priority the driver or other monitor routine will run at when it continues execution.

$= This register specifies the print format for numeric values other than addresses. If it is set to zero (default), values are printed in octal. If it is negative, values will be printed in signed decimal. If it is positive, values will be printed in unsigned decimal.

3.8.3 Control Tables

Control tables are used to define or access tables of information used by ODT. The contents of a control table can be accessed by specifying the desired table element number between the "$" and the table name. For example, $2R/ will display the current value of relocation register two.

The control tables have the following meanings:

$nB This table contains the locations corresponding to the eight breakpoints, 0-7. It corresponds to the values specified in the B command.

$nD This table contains the action routine number to execute when the corresponding breakpoint is entered. It corresponds to the argument on the D command.

$nG This table contains the proceed counts for each corresponding breakpoint. It corresponds to the values specified in the P command.

$nR This table contains the relocation address for each corresponding relocation register. If a relocation register is not in use its value will be -1.

$nT This table contains four words which specify the address of the console device DL-11 interface to use for ODT. These values should not be changed as individual table entries. Use the T command to change the console device address, if necessary. These words have the following meaning:

$0T: Keyboard CSR
$1T: Keyboard data buffer
$2T: Printer CSR
$3T: Printer data buffer

$nE This table contains memory management register pointers that are saved when a breakpoint occurs and restored when execution continues. The registers have the following meaning:

$0E: APR6 value used for memory management work.
$1E: Saved KDSAR6 value.
$2E: APR6 value used for ODT.
$3E: Pointer to the saved APR6 value.
$4E: Pointer to the saved DAPR6 value.
$5E: Saved MMUSR3 value.
$6E-$12E Saved values for trap vectors.

$nI This table contains memory management register pointers that are saved when a breakpoint is set.

$nV This table contains the kernel mode I space page address register (PAR) values currently in use by the monitor. Table entries $0V through $7V correspond to KISAR0 through KISAR7, respectively. Table entries $10V through $17V correspond to KISDR0 through KISDR7, respectively.

$nW This table contains the kernel mode D space page address register (PAR) values currently in use by the monitor. Table entries $0W through $7W correspond to KDSAR0 through KDSAR7, respectively.

nX This table contains the user mode, I-space page address register (PAR) values currently in use by the monitor. Table entries $0X through $7X correspond to UISAR0 through UISAR7, respectively.

$nY This table contains page address register (PAR) values to use with the Mx command (see section 3.8.4) for user defined memory management mapping. Table entries $0Y through $7Y correspond to through APR7, respectively.

3.8.4 Commands

The commands for monitor ODT have the following meanings:

n,v The contents of the relocation register specified by "n" is added to the value "v" to form a new value.

v/ Open location "v" as a word and display its contents.

v\ Open location "v" as a byte and display its contents.

v' Open location "v" as a byte and display its contents as an ASCII character.

v'' Open location "v" as a word and display its contents as two ASCII characters.

v% Open location "v" as a word and display its contents as three RAD50 characters.

<CR> Close the currently open location. If the carriage return is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

<LF> Close the currently open location and open the following one. If the line feed is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

Close the currently open location and open the preceding one. If the uparrow is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

_ Close the currently open location and use its contents as a PC relative offset to a new location to open. If the underline is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

@ Close the currently open location and use its contents as the new location to open. If the @ is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

> Close the currently open location and use the low order byte of its contents as a PC relative offset to the new location to open. If the > is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

< Close the currently open location and reopen the last explicitly opened location. If the < is preceded by a value, this value will be stored in the currently open word or byte before it is closed.

= Print the value preceding the equal sign using the print format specified in the $= register. A useful technique for displaying a negative number as the equivalent positive number is to type -Q= after displaying the negative number.

nA Execute action routine "n".

v;nB Set breakpoint "n" at location "v".
v;B Set breakpoint at location "v". Let ODT pick the breakpoint number.
nB Remove breakpoint number "n".
B Remove all breakpoints.

n1;n2D Execute action routine "n1" when breakpoint "n2" is entered.
nD Remove the action routine associated with breakpoint "n".
D Remove action routines from all breakpoints.

E Print all locations between the limits specified by $L and $H that have the same effective address as the contents of $A using the mask $M. If the E command is preceded by one or two arguments in the form v1;E, ;v2E or v1;v2E, $L will be set to "v1" and $H will be set to "v2" before executing the E command. The contents of a location are considered to be the same effective address if any of the following conditions are met in relation to $A: (l) They are equal. (2) The PC-relative address is equal. (3) The branch displacement based on the low byte is equal.

F Fill all memory locations between the limits specified by registers $L and $H with the value contained in register $A. If the F command is preceded by a value (nF), that value is loaded into $A before executing the F command.

G Start execution at the location specified by register $7. If the G command is preceded by a value (nG), that value is loaded into $7 before executing the G command.

k Determine which relocation register is closest to (but not greater than) the address contained in the currently open word. Use this relocation register to print the value of the currently open location as an offset from the relocation register.
nK Print the address contained in the currently open word as an offset from relocation register "n".
v;nK Print the address specified by "v" as an offset from relocation register "n".

L Print all locations between the limits specified by $L and $H. Eight values are printed per line. The value of $L used for the print is equivalent to $L ANDed with 177770_8. If the L command is entered in the form v1;v2;v3L, "v1" specifies the desired output device (0 or null for console device, 1 for LP11), "v2" specifies $L if non-null, "v3" specifies $H if non-null.

Ms Use memory management mode "s" for accessing memory locations, where "s" is one of the following:

- K Kernel. I & D space is not in use or is not supported. Same as mode I when D space is not supported.
- I Kernel I space.
- D Kernel D space.
- U User I space.
- X User defined mapping (see control table Y in section 3.8.3).

N The N command is identical to the W command except that only those locations that do not meet the test for the W command will be printed.

vO Print the PC-relative offset and branch displacement from the currently opened word to location "v".
v1;v2O Print the PC-relative offset and branch displacement from address "v1" to address "v2".

P Proceed from the current breakpoint. If the P command is preceded by a value (nP), that value specifies the number or times to proceed through the current breakpoint before stopping again.

v;nR Set relocation register "n" to value "v".
v;R Set relocation register 0 to value "v"
nR Clear relocation register "n".
R Clear all relocation registers.

S Execute one instruction and enter breakpoint 8. If the S command is preceded by a value (nS), the specified number of instructions will be executed before stopping.

vT Use the device located at location "v" in the I/O page as the console terminal. If the T command is not preceded by a value, the console terminal will be reset back to the standard assignment of 177560_8. A global location (.ODTKB) can be patched to specify a different value to reset the console back to if necessary. The console terminal must use a DL-11 interface. DZ-11, DJ-11 and DH-11 interfaces are not supported.

W Print all locations between the limits specified by $L and $H that are equal to the value in $A using the mask $M. If the W command is preceded by one or two arguments in the form v1;W, ;v2W or v1;v2W, $L will be set to "v1" and $H will be set to "v2" before executing the E command.

n(..) The (command is used to define action routine "n". This action routine is a sequence of up to 31 bytes of ODT commands terminated by a ")". The action routine can be executed by the A command or at the time of a breakpoint, by the D command.

n) Display action routine "n".

v[..] Execute the ODT commands within the brackets if the value of expression "v" is non-zero.

1;v[..] Execute the ODT commands within the brackets if the value of expression "v" is zero.

<RO> A rubout will abort any command currently in process.

Chapter 4

RESIDENT LIBRARIES

AND RUNTIME SYSTEMS

Resident libraries and runtime systems provide a means of sharing information and program code that would normally be contained in each copy of a program. For example, if twenty people are all using an order entry program, they can share one copy of the program instead of having twenty separate copies. This can drastically reduce swapping and memory requirements and improve overall system performance.

Another common example is the execution support code used by all programs written in the same language. By putting this code in a resident library or runtime system, only one copy of the code is required, regardless of the number of programs using the language.

For example, the BASIC-PLUS runtime system contains all the code needed to compile and execute BASIC-PLUS programs. Only one copy of this code is needed, regardless of the number of programs using BASIC-PLUS. The BASIC2 runtime system contains all the code needed to execute BASIC-PLUS-2 programs and provide RSX emulation. The BASICS resident library contains the same execution support code but is used in conjunction with the RSX runtime system.

The creation and use of resident libraries and runtime systems is described in the RSTS/E manual set. The *RSTS/E Programmer's Utilities Manual* and the *RSTS/E Task Builder Reference Manual* contain information on building and accessing resident libraries and runtime systems. The *RSTS/E System Directives Manual* and the *RSTS/E Programming Manual* contain information on how the monitor supports resident libraries. The *RSTS/E System Manager's Guide* contains information about adding and removing resident libraries and runtime systems.

This chapter provides information not currently available in any of the documents described above. It concentrates on special ways to use resident libraries and runtime systems that are not discussed in the standard documentation.

Although resident libraries and runtime systems are similar in many ways, they also also have several distinct differences. In general, resident libraries are used used to extend the functionality of a program. The resident library is under the control of the program and is accessed as needed. Runtime systems are used to extend the functionality of the monitor. The program is under the control of the runtime system, which interfaces to the monitor on the program's behalf.

The decision of whether to use a resident library or a runtime system to make a program's code sharable depends on several factors. The following table lists some of the decision criteria necessary:

Use a resident library if:

The program needs to use the capabilities provided by an existing runtime system.

The program does not use any of the capabilities of an existing runtime system, but the sharable code will be under the control of the program (ie. it is support code, subroutines or data used by a program). A runtime system could also be used in this case but would generally be more difficult to implement.

The virtual address space used by a program needs to be reduced by using memory resident overlays.

Use a runtime system if:

The sharable code controls the execution of a program.

The sharable code emulated another operating system or environment.

The sharable code or user program uses EMT, TRAP, IOT or BPT instructions in a non-standard way.

The sharable code requires control as a keyboard monitor.

Chapter 2 of the *RSTS/E System Directives Manual* should be read and understood before continuing your reading in this section. This chapter provides an explanation of the relationship between virtual and physical memory addresses, the use of memory management registers, the user program area and runtime system pseudo vectors.

4.1 ADDRESS SPACE USAGE

A "job" in RSTS/E consists of a user program, a runtime system and one or more optional resident libraries. User program addresses always start at virtual address 0. Runtime system addresses always end at virtual address 177774_8. The virtual address space betwen the end of the user program and the beginning of the runtime system is used by resident libraries or is unused.

The virtual addresses used by a job are mapped into physical memory locations by the memory management hardware. Each segment of a job (ie. the user program, resident library or the runtime system) is contained in a separate section of physical memory. Although the contents of each section of a job are contiguous in memory, the individual sections are not necessarily contiguous.

Because the PDP-11 memory management hardware maps memory in 4 K-word increments, a portion of the virtual address space may not be available. Any remaining virtual address space between the end of the user program or resident library and its next 4 K-word boundary will be unused. The same is true for the virtual address space between the beginning of a 4 K-word boundary and the beginning of a runtime system.

Resident libraries are mapped as needed using the memory management hardware. For this reason, they also use virtual address space in 4 K-word increments. If a resident library uses memory resident overlays, each overlay region uses one or more additional 4 K-word segments.

4.1.1 Address Requirements for a Resident Library

Resident libraries can be built to run at specific virtual addresses or they can be position independent. Each has its own advantages.

Resident libraries designed to run at a specific address are easy to build but they restrict the flexibility of the application using them. If a library is built to run at a specific address, it cannot be used in the same program with another library built to use the same address range. The size of the program is also restricted by the starting address of the library, since the program's addresses cannot overlap those of the library.

Position independent resident libraries do not have the problems of non-position independent libraries but they are somewhat more difficult to write. Certain addressing modes cannot be used in position independent code. Although this makes programming slightly more difficult, position independent code should be used whenever possible because it minimizes the address space problems inherent in position dependent code.

Any application that can be written position dependent can also be written position independent. (See section 4.1.3 for more information on position independent code.)

4.1.2 Address Requirements for a Runtime System

Runtime systems have very specific address requirements. The last 18 words of the runtime system are a pseudo vector region used to interface between the monitor and the runtime system. The runtime system must be built such that the pseudo vector region ends at location 177774_8. The runtime system code immediately precedes the pseudo vector region.

The pseudo vector region is used to provide information to the monitor and to provide pointers to routines that perform specific functions at the monitor's request. The ordering of the contents of this region is important. Each pointer or piece of information is accessed at a fixed address at the high end of the virtual address space.

A complete description of the pseudo vector region is provided in the *RSTS/E System Directives Manual*. The contents of the pseudo vector region will not be described in detail here. See the *RSTS/E System Directives Manual* for more information. The following table should serve as a quick reference to the pseudo vectors (addresses are in octal):

Symbol	*Address*	*Description*
P.FLAG	177732	Flags describing the runtime system: PF.EMT Monitor EMTs require a special prefix (specified in the low byte of this word). PF.CSZ Compute initial memory requirements, in K-words, as (filesize + 3)/4. PF.REM Remove the runtime system from memory when its user count becomes zero. PF.NER Do not log errors occurring within the runtime system. PF.RW Write access is allowed in this RTS. PF.1US Only one job may use this runtime system at a time. PF.KBM The runtime system contains a keyboard monitor.
P.DEXT	177734	Default extension (in RAD50) for files running under this runtime system.
P.ISIZ	177736	Unused. Reserved for future use.
P.MSIZ	177740	Minimum size of user job image (in K-words).
P.FIS	177742	Pointer to FIS exception trap handling routine.
P.CRAS	177744	Pointer to routine to handle crash recovery system startup (system default runtime system only).
P.STRT	177746	Pointer to routine to handle normal system startup (system default runtime system only).
P.NEW	177750	Pointer to routine to handle control of a job when a run request is not desired. The contents of XRB + 2 and XRB + 4 plus the JFNOPR bit in the keyword (KEY) are used to specify why the runtime system was entered, as follows: New Job on System: JFNOPR Set to 1. XRB + 2 0 XRB + 4 0 Switch to Runtime System with Logged Out Job: JFNOPR Set to 1. XRB + 2:4 Calling runtime system name (in RAD50). Normal Switch to Runtime System: JFNOPR Cleared to 0. XRB + 2:4 Calling runtime system name (in RAD50).
P.RUN	177752	Pointer to routine to handle RUN requests. The XRB, FIRQB and KEY areas contain information about the RUN request.

P.BAD 177754 Pointer to routine to handle unexpected error conditions and traps. These errors are:

- B.4 Trap to 4 (odd address).
- B.10 Trap to 10 (reserved instruction trap).
- B.250 Trap to 250 (memory management violation).
- B.STAK Stack overflow.
- B.SWAP Fatal disk error during swap.
- B.PRTY Memory parity error.

P.BPT 177756 Pointer to routine to handle BPT instruction and T-bit traps.

P.IOT 177760 Pointer to routine to handle IOT instruction traps.

P.EMT 777762 Pointer to routine to handle EMT instruction traps. Control will not be passed to the runtime system for EMT instructions that are valid RSTS/E monitor calls unless the runtime system specifies that a special prefix is required for RSTS/E monitor calls (see PF.PFX in P.FLAG).

P.TRAP 177764 Pointer to routine to handle TRAP instruction traps.

P.FPP 177766 Pointer to routine to handle floating point processor exception traps.

P.CC 177770 Pointer to routine to handle interruption by a single ↑C.

P.2CC 177772 Pointer to routine to handle interruption by two or more ↑Cs.

P.SIZE 177774 Maximum size of user job image (in K-words).

4.1.3 Writing Position Independent Code

The taskbuilder typically builds a program so that it can only be loaded and executed at a specified virtual address. If more than one resident library is built for a specified virtual address range, the libraries cannot be used in the same program.

It is possible to write code so that is does not require being loaded at any specific position. Code written in this way is called "position independent", or "PIC". If the resident libraries are position independent, the taskbuilder will automatically build them in different virtual address ranges. This allows the libraries to be taskbuilt with little or no impact on the main program or other libraries.

To code a resident library so that it is position independent requires the proper use of instruction addressing modes. If the following rules are followed, all memory references will be to fixed locations outside of the library or to locations within the library that are referenced using offsets from the current location counter. The actual location of the library is unimportant. (See Appendix G of the MACRO-11 Language Reference Manual for more information on writing position independent code.)

1. All code must be in PSECTs. ASECTs cannot be used, except to define fixed data values and offsets.
2. All addressing modes that only involve registers are position independent:

```
MOV     R0,R1
MOV     (R0),(R1)+
```

3. Relative mode addressing is position independent when a relocatable address is referenced from a relocatable instruction:

```
MOV     #1,DESTIN       ;DESTIN is a relocatable address
CLR     DESTIN          ;DESTIN is a relocatable address
```

4. Absolute mode addressing is position independent when the address is fixed, regardless of the location of the instruction:

```
MOV     #1,@#FIRQB      ;FIRQB is always at a fixed location
CLR     @#XRB           ;XRB is always at a fixed location
MOV     @#FIRQB,R0      ;FIRQB is always at a fixed location
```

5. Immediate mode references are position independent when the value is fixed, regardless of the location of the instruction:

```
MOV     #1,R0           ;Absolute values are always fixed
MOV     #FIRQB,R0       ;FIRQB is always at a fixed location
```

6. Indexed mode references are position independent if the index value is fixed, regardless of the location of the instruction:

```
MOV     R0,2(R1)        ;A non-symbolic value is always fixed
MOV     XRB(R3),R0      ;XRB is always at a fixed location
```

The following instructions are NOT position independent because they violate one of the rules above:

```
        .ASECT                  ;Violation of rule 1:  Code within an
        MOV     R0,R1           ;ASECT is not position independent.

        TST     FIRQB+22        ;Violation of rule 3: FIRQB+22 is not a
                                ; relocatable address.  Use TST @#FIRQB+22
                                ; instead.

LOC:    MOV     @#LOC,R0        ;Violation of rule 4:  The value of LOC depends
                                ; on the location of the instruction.

LABEL:  MOV     #LABEL,R0       ;Violation of rule 5: The value of LABEL depends
                                ; on the location of the instruction.
```

As the example above shows (violation of rule 5), instructions that reference a relocatable location using an absolute addressing mode are not position independent. MOV instructions that are not position independent, such as:

```
LOC:    MOV     #LOC,R0
```

can be made position independent using the MOVPIC macro provided in COMMON.MAC. This macro generates the following code:

```
LOC:    MOVPIC #LOC,R0
         MOV     PC,R0          ;This code is generated by the macro.  It calculates
         ADD     #LOC-.,R0      ; the actual address of LOC using an offset from
                                ; the current instruction location.
```

4.2 SPECIAL LIBRARIES AND RUNTIME SYSTEMS

User written resident libraries are normally included in a user program without any special mapping requirements. The entire library is included in the program and used as needed. This section describes some additional ways to use resident libraries. Use of these techniques allows efficient use of virtual address space and system resources.

Runtime systems are normally used to control execution of a program or to emulate another operating environment. A different technique that is very effective involves using runtime systems as coordinating programs that share a common data area. Vastly improved system performance can be achieved using this technique.

4.2.1 Memory Resident Overlays Within a Library

A resident library normally consists of one or more subroutines that are mapped by the main program and called as needed. The entire library may be mapped and accessed at once, or an individual section may be mapped and accessed while the remainder of the library is inaccessable.

Mapping a small section of a resident library reduces the amount of virtual address space required for the library but sometimes makes creating the library more difficult. Libraries which must be completely mapped in order to be used are easier to write but can take excessive amounts of virtual address space from a program.

Care should be excercised when deciding to use memory resident overlays within a resident library. Careless use can significantly impact the virtual address requirements of a resident library. Virtual addresses are allocated to the resident library and each memory resident overlay region within it in 4 K-word increments. Any address space that is unused within the root segment and any overlay regions will be wasted.

Since the resident library always uses at least 4 K-words of virtual address space, there is no need to use memory resident overlays if the entire library will fit within 4 K-words. Similarly, since a resident library with a memory resident overlay uses at least 8 K-words of virtual address space, there is no need to use a memory resident overlay if the entire library and its subroutines will fit within 8 K-words.

The best application of memory resident overlays in resident libraries is when a library requires more than 8 K-words of virtual address space and when the code of a library consists of a root segment that approaches 4 K-words in length and that references more overlayed subroutines than will fit within an additional 4 K-word region.

4.2.2 Common Data Areas

Resident libraries can be used to share data as well as program code. The data can be a table of constant values, information read and written by more than one program, or information written by one program and read by another. The information can be read-only or read-write.

The most valuable use of libraries that serve as common data areas is for sharing read-write data between two or more programs. Programs that would normally send large amounts of data using send receive or a disk workfile can be designed to use a shared memory area to pass this information. This can drastically reduce the overhead required for send/receive character handling and disk I/O.

BASIC-PLUS-2 (BP2) programs can use resident common areas as easily as MACRO. The resident common area is included by naming a COMMON area in the BP2 program to the same name as the .PSECT name in the MACRO program which defines the resident common area. The following example shows the use of this capability:

Macro source (COMBUF.MAC):

```
        .TITLE  COMBUF  Resident Library Common Data Area
        .PSECT  COMBUF  GBL,D,RW
LOCK:   .WORD   -1      ;Semaphore: 0 = Locked
BYTCNT: .WORD   0       ;Buffer byte count
BUFFER: .BLKB   128.    ;128 byte buffer
        .END
```

BASIC-PLUS-2 source (REDBUF.BP2):

```
1       EXTEND
10      COMMON (COMBUF) LOCK%,          ! Semaphore: 0 = Locked  &
                        BYTCNT%,        ! Buffer byte count  &
                        BUFFER$=128%    ! 128 byte buffer
20      IF LOCK%>=0%    &
          THEN  PRINT "%The buffer is locked. Please wait..."  &
\               SLEEP 2%  UNTIL LOCK%<0%
30      PRINT LEFT(BUFFER$,BYTCNT%)
40      END
```

Once both tasks have been assembled, they are taskbuilt using the following commands:

For the resident library:

```
TKB>COMBUF/-HD/PI,COMBUF,COMBUF=COMBUF
TKB>/
TKB>ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=BUFFER:0:4000
TKB>//
```

For the BASIC-PLUS-2 program accessing the resident library:

```
TKB>REDBUF,REDBUF=REDBUF,LB:BP2COM/LB
TKB>/
TKB>ENTER OPTIONS:
TKB>RESLIB=COMBUF/RW:6
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>//
```

All read-write resident libraries must be task built as position independent (the /PI taskbuilder switch). This causes the .PSECT name to be included in the symbol table along with the global symbols. Assuming the common area does not make any absolute references to any locations within itself, no special considerations need to be observed to make the common area position independent since it does not contain any instructions.

There are several disadvantages that go along with using resident libraries as common data areas. The first is that the resident library uses at least 4 K-words of the virtual address space available to the program. This can be a serious problem for large programs or programs that use several non-clustered resident libraries.

Another problem is the lack of locking or semaphore mechanisms available between jobs. The programs must include their own code for restricting access to the read-write data while it is being updated. The following example shows the use of a user written locking semaphore:

```
        ; This code is within the read-write resident library:
LOCK:   .WORD   -1              ;Semaphore: 0 = Locked

        ; This code is accessed by each program that needs to lock the
        ; common data area.  The data area should be locked before it
        ; is changed or when the data should not be changed while it is
        ; being accessed.
        .GLOBL  LOCK
LOCKIT::INC     LOCK            ;Attempt to set the lock
        BEQ     2$              ;Lock succeeded
        DEC     LOCK            ;Area was already locked, wait and try again later
        MOV     #2,@#XRB        ;Specify a 2 second sleep
        .SLEEP                  ;Sleep for 2 seconds
        BR      LOCKIT          ;Try to lock the area again
2$:     RETURN                  ;Return with area locked

        ;This code unlocks the area locked by LOCKIT.  It must be called
        ;each time an area is locked.
        .GLOBL  LOCK
UNLOCK::DEC     LOCK            ;Unlock the data area
        RETURN                  ;Return with data area unlocked
```

The only way to notify another program about a change in information in the common data area, other than the continual checking used in the preceding example, is to use small message send receive. This works well when a program needs to wait a relatively long period of time for another program to take some action before accessing the common data area. Additional information about the status of the common data area can be passed in the parameter area of the small message without incurring additional overhead.

4.2.3 Coordinating Runtime Systems

The monitor call that passes control between runtime systems has a parameter that allows control to pass from one runtime system to another without effecting the contents of the low segment. This capability can be used to develop coordinating runtime systems.

Programs of virtually unlimited size can be written by building a set of runtime systems that share data in a common low segment. Each runtime system can perform a specific function on the data in the low segment and then pass control to another runtime system.

This technique can typically be applied to an order entry system. One runtime system performs all screen handling and data collection. Once the data is collected, the first runtime system transfers control to another runtime system which processes the order using the collected data. The second runtime system then returns control to the first runtime system, which continues with the next order. This technique can obviously be expanded to more than two coordinating runtime systems.

Any number of people could be using this order entry system simultaneously. All of the code will be shared. Each user will have only a small data area to swap. Since each user's job image is so small, swapping would be minimal or non-existent.

Many of the same advantages of coordinated runtime systems can be achieved using resident libraries by combining the coordinating "programs" into one large library and accesssing the proper entry in the library from a small user program. The disadvantage of this approach is that the programs become less autonomous and, therefore, more difficult to support if they are combined into one large library.

4.3 BUILDING A RESIDENT LIBRARY

The *RSTS/E Task Builder Reference Manual* contains a thorough description of the use of the task builder to build and reference resident libraries. The *RSTS/E Programmer's Utilities Manual* describes the use of the utilities used in the examples.

4.3.1 Building a Position Dependent Library

In the following example, the resident library will be built to use fixed virtual addresses of 140000_8-157777_8. This is the virtual address range mapped by APR6. The files FILEA.OBJ and FILEB.OBJ are object modules which contain the resident library code.

```
>TKB
TKB>LIB/-HD,LIB,LIB=FILEA,FILEB
TKB>/
TKB>ENTER OPTIONS:
TKB>PAR=LIB:140000:20000
TKB>STACK=0
TKB>//

RUN $MAKSIL
MAKSIL V8.0-04 RSTS V8.0-04 Northwest Digital
Resident Library name? LIB
Task-built Resident Library input file <LIB.TSK>?
Include symbol table (Yes/No) <YES>?
Symbol table input file <LIB.LIB>?
Resident Library output file <LIB.LIB>?
LIB built in 1 K-words, 2 symbols in the directory
LIB.TSK renamed to LIB.TSK<40>

>PIP LIB.STB<40>/RE
```

4.3.2 Building a Position Independent Library

In the following example, the resident library will be built position independent. It can use any virtual address range when it is bound to a controlling task. The files FILEA.OBJ and FILEB.OBJ are object modules which contain the resident library code. All code was written using position independent coding techniques (see section 4.1.3).

4.3.3 Building a Read-Write Resident Library

In the following example, the resident library contains only common data areas. Each .PSECT will be included in the symbol table file (.STB) and can be referenced as a resident library or as a named COMMON in BASIC-PLUS-2, COBOL or FORTRAN-FOUR-PLUS. The file FILEA.OBJ contains the definition of all .PSECTs and the allocation of space for data within each PSECT.

```
TKB>COM/-HD/PI,COM,COM=FILEA
TKB>/
TKB>ENTER OPTIONS:
TKB>PAR=COM:0:4000
TKB>STACK=0
TKB>//

>RUN $MAKSIL
MAKSIL V8.0-04 RSTS V8.0-04 Northwest Digital
Resident Library name?  LIB
Task-built Resident Library input file <LIB.TSK>?
Include symbol table (Yes/No)<YES>?
Symbol table input file <LIB.STB>?
Resident Library output file <LIB.LIB>?
LIB built in 1 K-words, 1 symbols in the directory
LIB.TSK renamed to LIB.TSK<40>

>PIP LIB.STB<40>/RE
```

Note that the resident library was built using the /PI switch, which indicates position independent code. The use of this switch causes the name of each .PSECT to be included in the symbol table file. The .PSECT name is used to reference the start of a named COMMON area. All access to the data within the common area is by relative offset from the beginning of the COMMON area.

4.4 BUILDING A RUNTIME SYSTEM

Building a runtime system is described in the RSTS/E Programmer's Utilities Manual. However, some additional discussion is in order.

The only major conditions placed on a runtime system by the monitor are that the last 18 words of the runtime system must contain the pseudo vector area and that the last virtual address used by the runtime system must be 177774_8.

The first of these conditions is met in one of two ways. One method is to include the entire runtime system within one PSECT with the pseudo vectors as the last code in the section. A second method is to use multiple PSECTs and to order the PSECTs such that the pseudo vector code is contained the last PSECT.

Bear in mind that the taskbuilder normally arranges PSECTs in alphabetical order. To ensure that the pseudo vector area is the last PSECT in the module, the pseudo vector area should be defined in a PSECT with a name of .99998 (a name that will always sort higher than any other PSECT name except .99999, which is used by the runtime system debugger, RTSODT) or the /SQ switch should be used in the taskbuilder to order the PSECTs in the same order in which they were defined.

The second condition, that of ensuring that the runtime system ends at location 177774_8, is a little more difficult to meet using the taskbuilder. Since the taskbuilder does not have an option to allow the specification of a desired ending address, an extend section must be used to move the runtime system code to the proper address so that it will end at 177774_8.

Aligning a runtime system against the proper upper boundary is a two step process. The runtime system is first taskbuilt using a taskbuild command file without regard to the ending address. The MAKSIL program is then run using the task just built.

MAKSIL will modify the taskbuild command file to extend the .99998 .PSECT (using the EXTSCT option in the taskbuilder) to the proper size to cause the task to end at location 177774_8. When the task is rebuilt using the updated command file it will be aligned at the proper boundary. The MAKSIL program is now run again and the runtime system SIL image is produced.

The following example shows the steps involved in building a runtime system. Note that two passes through the taskbuilder and MAKSIL are normally required in order to force the pseudo vector to end at location 177774_8.

```
>TKB @EXAMPL
>RUN $MAKSIL
MAKSIL V8.0-04 RSTS V8.0-04 Northwest Digital
Resident Library name?  EXAMPL/RTS
Task-built Run-Time System input file <EXAMPL.TSK>?
The run-time system is not aligned
Edit mode (Yes/No) <YES>?
Task-builder command input file <EXAMPL.CMD>?
The task-builder commands have been changed as follows
        PAR=EXAMPL:160000:020000        PAR=EXAMPL:160000:020000
        STACK=3072                      STACK=3072
                                        EXTSCT=.99998:001726

EXAMPL will load in a 4 K-word partition using 1 K-words physical memory.
001726 (octal) bytes may be used for expansion.

Corrected command file name <EXAMPL.CMD>?
Please task build again using EXAMPL.CMD

>TKB @EXAMPL
>RUN $MAKSIL
MAKSIL V8.0-04 RSTS V8.0-04 Northwest Digital
Resident Library name?  EXAMPL/RTS
Task-built Run-Time System input file <EXAMPL.TSK>?
The run-time system is correctly aligned
Edit mode (Yes/No)<Yes>? NO
Include symbol table (Yes/No) <YES>?
Symbol table input file <EXAMPL.STB>?
Run-Time System output file <SY:[0,1]EXAMPL.RTS>?
EXAMPL built in 1 K-words, 1 symbols in the directory
EXAMPL.TSK renamed to EXAMPL.TSK<40>
```

4.5 DEBUGGING A RUNTIME SYSTEM

The RSX object library (LB:SYSLIB.OLB) contains a version of ODT called RTSODT. RTSODT is designed specifically for debugging runtime systems. It is taskbuilt with the runtime system object modules and becomes part of the runtime system.

The commands available in RTSODT are identical to those available in ODT. Please refer to section 3.8 and the ODT Reference Manual for information on these commands.

RTSODT contains its own psuedo vector region. When taskbuilt with the runtime system code, the pseudo vectors for RTSODT immediately follow those of the runtime system. Thus, the monitor uses ODT's pseudo vectors while ODT uses the runtime system's pseudo vectors.

When the monitor enters the runtime system at any of the entry points specified in the pseudo vector region, RTSODT will stop execution with a breakpoint and wait for commands. If the procede command (P) is issued, the pseudo vector entry point for the runtime system will be entered and execution will continue normally.

If you want an entry point to continue straight through to the runtime system without breakpointing, the following patch should be applied at RTSODT command level. The patch should be repeated as necessary for all entry points that are to be disabled:

```
_P.????/xxxxxx @          (P.???? is the name of the pseudo vector)
_xxxxxx/000003 240
_
```

When RTSODT is include in a runtime system, the following global symbols must be defined in the runtime system code:

Symbol	*Description*
O.FLAG	Value to use for P.FLAG. This value is normally equated to PF.REM!PF.NER!PF.1US. The PF.CSZ,PF.EMT and PF.KBM bits can also be set, if needed. If the PF.RW bit is not set, the runtime system must always be loaded with the /RW switch.
O.DEXT	Value to use for P.DEXT. This value should normally be the same as that used for P.DEXT in the runtime system code.
O.MSIZ	Value to use for P.MSIZ. This value should normally be the same as that used for P.SIZE in the runtime system code. Since RTSODT will add 4056 bytes to the runtime system size, the value of P.SIZE used during debugging may have to be less than that used during normal operation.

The command file used to taskbuild the runtime system must be modified to include a reference to RTSODT in the system object library (LB:SYSLIB.OLB). This is done by adding the object module specification "LB:SYSLIB/LB:RTSODT" after the last object module to include in the taskbuild.

The order of the normal runtime system object modules must be the same as used to taskbuild the runtime system without RTSODT. The runtime system pseudo vector region must remain at the highest address in the runtime system code. The pseudo vector region of RTSODT is taskbuilt immediately after the runtime system pseudo vector region.

The partition size specified with the PAR option of the taskbuilder will have to be changed if the additional size caused by including RTSODT extends the task image size beyond that specified by the existing PAR option. The STACK option may also have to be changed to compensate for the additional task size caused by adding RTSODT.

The following example shows the changes required for taskbuilding with RTSODT. UWRTS.OBJ contains the object module code for the runtime system.

```
>TKB
TKB>UWRTS/-HD,UWRTS,UWRTS=UWRTS,LB:SYSLIB/LB:RTSODT
TKB>/
TKB>PAR=UWRTS:160000:020000        (This value varies with each program)
TKB>STACK=0                        (This value varies with each program)
TKB>//
```

The first user to enter a runtime system containing RTSODT obtains control of the debugger. Subsequent users are ignored by RTSODT, which is transparent to the user. If PF.1US is set in P.FLAG, or if the /1USER switch is used when the runtime system is loaded, only one user will be allowed access to the runtime system.