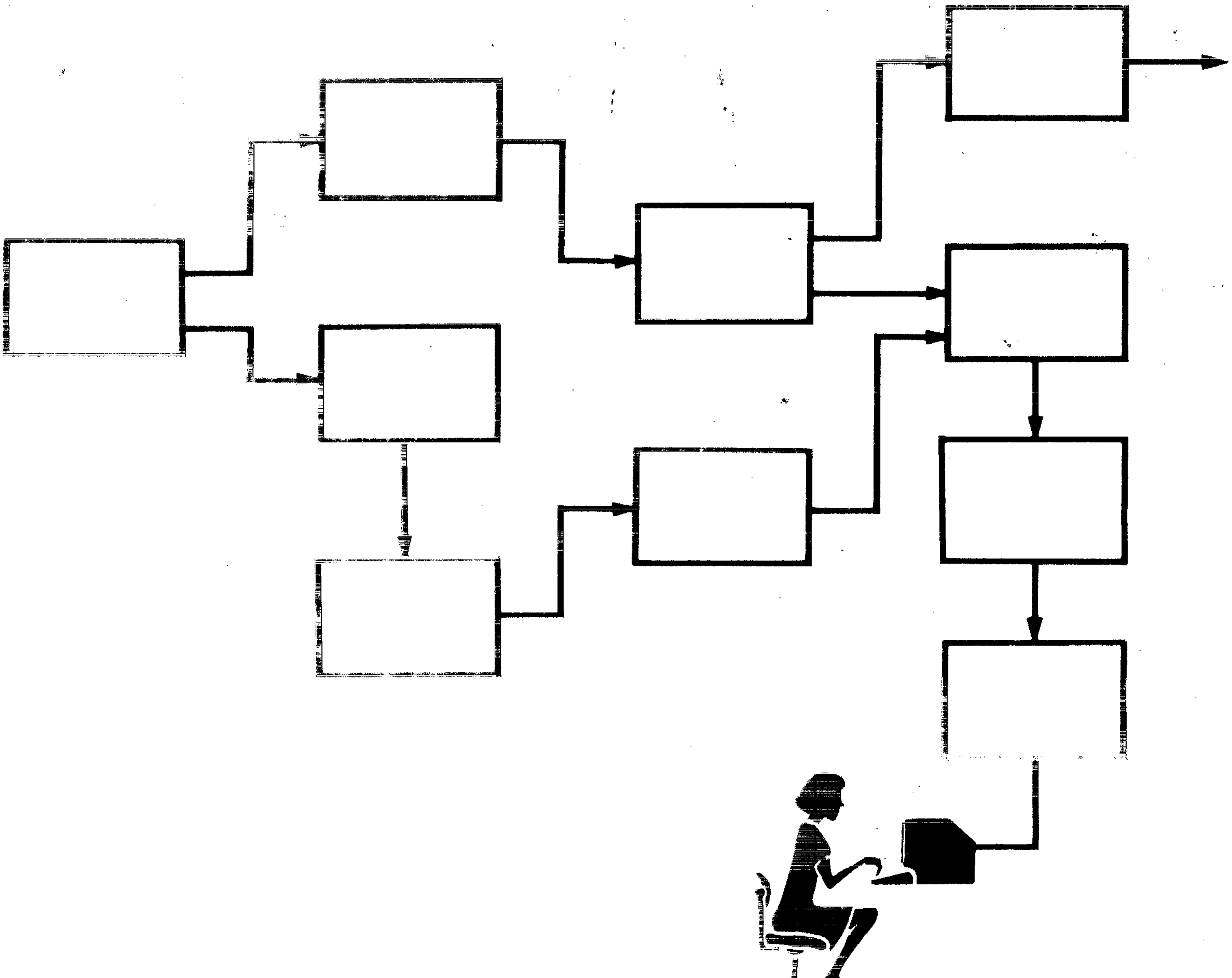
# RSTS/E V8.0

## INTERNALS MANUAL



# RSTS/E
# INTERNALS MANUAL

Order No. AA-CL35A-TE

September 1984

This manual describes the internal structure of the RSTS/E V8.0 operating system.

| | |
|---|---|
| OPERATING SYSTEM AND VERSION: | RSTS/E V8.0 |
| SOFTWARE VERSION: | RSTS/E V8.0 |

digital equipment corporation, maynard, massachusetts

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

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

DEC
DECnet
DECsystem-10
DECSYSTEM20
DECUS
DECwriter
DIBOL
Edusystem
IAS
MASSBUS
PDP
PDT
RSTS
RSX
RT
UNIBUS
VAX
VMS
VT

# TABLE OF CONTENTS

## Preface

## PART I: OVERVIEW OF RSTS/E FUNCTIONALITY

### Chapter 1: Overview of RSTS/E Functionality

## PART II: THE MONITOR

### Chapter 2: Monitor Structure

### Chapter 3: Job Control

**Chapter 6: Device Control**

## PART III: DISK I/O AND THE FILE PROCESSOR

### Chapter 7: Disk Terminology

### Chapter 8: On-Disk Structures

**Chapter 13: Terminal Service Structure**

**Chapter 14: The Human Interface**

## PART V: SHARED COMMON CODE

**Chapter 15: Run-Time Systems**

**Chapter 16: Resident Libraries**

## PART VI: SYSTEM GENERATION AND INITIALIZATION

**Chapter 17: System Generation**

Chapter 18: System Initialization

## Figures

# Preface

The *RSTS/E V8.0 Internals Manual* provides developers and maintenance programmers with a comprehensive technical overview of the RSTS/E monitor. The manual is a functional definition of the monitor, presenting the philosophy, functionality and structure of RSTS/E as a total unit. The emphasis, therefore, is on how the various elements of the system fit and function together. In addition, some historical perspective has been included to add the element of "why" as well as "how."

While this document includes general descriptions of major data structures and procedures, low-level design detail has been avoided. Numeric offsets within data structure diagrams have also been omitted wherever possible. The layout of bits and bytes and the exact details of various algorithms and procedures can change frequently from version to version, whereas a technical overview document should remain reasonably constant over the lifetime of the system as a whole.

This document is divided into six sections, or "parts." Each part deals with a functionally separate aspect of the RSTS/E operating system.

| | |
|---|---|
| Part I: | Overview of RSTS/E Functionality |
| Part II: | The Resident Monitor |
| Part III: | Disk I/O and the File Processor |
| Part IV: | Terminal Service and the Human Interface |
| Part V: | Shared Common Code |
| Part VI: | System Generation and Initialization |

No prior knowledge of RSTS/E is required to understand this document, and in general, each part can stand alone without reference to the rest of the manual. However, Part I *(Overview of RSTS/E Functionality)* should be considered prerequisite reading for the other five parts. In addition, the reader should have some basic knowledge of assembly language programming concepts as well as of the general architecture of the PDP-11 processor.

# Part I
# OVERVIEW OF RSTS/E FUNCTIONALITY

This part of the *RSTS/E V8.0 Internals Manual* presents a general overview of the total RSTS/E operating system.

Some facets of RSTS/E that are discussed here (most notably the system utility programs) are not dealt with in later parts. While these utilities are an integral piece of the RSTS/E environment, they are not true system components. Utilities run under the RSTS/E monitor as *applications programs* and, as such, are not actually part of the system executive. They are discussed here, however, to complete the picture of the RSTS/E system as a whole.

# CHAPTER 1
# Overview of RSTS/E Functionality [V8.0]

RSTS/E is a multiuser, general-purpose timesharing system. Its uses include interactive timesharing, batch processing, program development, and special purpose applications. Using multiterminal services, up to 127 concurrent terminal users (in both local and remote locations) can interact with multiple application tasks. Without multiterminal services there can be a maximum of 63 users. Tasks can share computational, storage, and I/O services provided by the RSTS/E system.

Programs can be written in any of several languages. BASIC-PLUS and the MACRO-11 assembler language are included as standard programming languages, while several other languages are optionally available.

The RSTS/E system consists of the RSTS/E monitor, standard device drivers, system utilities (known as Commonly Used System Programs or CUSPs), and a standard set of additional software components. Some of the major features of RSTS/E include the following:

- Interactive timesharing
- Dynamic allocation of system resources
- User and job privileges
- Interjob communications
- Extensive file processing including file sharing and protection mechanisms
- A software-maintained cache of frequently accessed disk data
- Disk file and device backup and restore utilities
- Terminal handling designed for an interactive environment
- Magnetic tape processing
- Shared common code
- Line printer spooling
- User command languages
- System manager definition of user commands
- System operational control utilities
- System reliability features
- System maintenance tools

The remainder of this chapter presents a general overview of RSTS/E concepts and functionality. Section 1.1 discusses job management, Section 1.2 discusses device control, and Section 1.3 discusses operational control of the system.

## 1.1 Job Management

The basic unit of work under RSTS/E is a "job." The RSTS/E monitor balances all work done on the system using this job concept.

Each user of a RSTS/E system is associated with a job and normally interacts with that job by using a terminal. Thus, from the user's point of view, a job is all the work done at the terminal from the time he logs onto the system until the time he logs off. However, jobs can also run detached (that is, not associated with a terminal) or they can be associated with a pseudo-keyboard, running under control of another job.

Although a RSTS/E system is limited to 63(10) jobs, some systems can support as many as 127(10) terminal users through use of the (optional) multiterminal feature. These limits are theoretical, however, and cannot always be achieved. The actual number of user jobs can be diminished by many factors such as memory capabilities, swap storage on disk, and the number of system processes running as detached jobs.

### 1.1.1 Timesharing and Allocation of System Resources

Under RSTS/E, jobs run one at a time. A job runs until it either enters an I/O wait state or exhausts the time quantum (or "time slice") assigned to it. When the currently running job ceases to run, the scheduler runs the next ready job. At the same time, interrupt-driven I/O device handlers process requested data transfers. After a transfer completes, the requesting job is marked as ready to run.

RSTS/E attempts to keep as many jobs in memory as possible. When more memory is required to run a job than is available, the system temporarily moves some jobs out of memory and stores them on the public disk structure in one of four areas known as "swap" files. This operation is known as "swapping." When a swapped out job is eligible to run again, it is swapped back into memory and started at the point at which it previously ceased execution. Jobs waiting for keyboard input and jobs waiting for device I/O completion can be stored in the swap files, while jobs involved in data transfers with DMA (direct memory access) devices, such as disks or magnetic tapes, must remain in memory.

The scheduler allocates central processor (CPU) time and memory residency among jobs based on their priority and processing requirements. A round-robin algorithm is used to select among eligible runnable jobs with the same priority. The system manager, any privileged user, or a privileged program can alter job priorities.

RSTS/E uses the memory management hardware capabilities to map a user's job area and any shared code into the user job's virtual address space. The size of a job can be expanded dynamically, subject to limits imposed by the system manager or other privileged users. The absolute limit for a job is 32 K-words,* but this limit is obtainable only if the job uses no resident libraries and runs under control of the "disappearing" RSX run-time system (see Section 15.2.2). The actual limit is usually 28 K-words. BASIC-PLUS programs cannot be larger than 16 K-words.

* One K-word equals 1024(10) words or 4000(8) bytes.

### 1.1.2 Privilege

As a resource sharing system, RSTS/E gives every user access to all the system peripherals and resources, as well as a wide range of functional capabilities, unless restricted by the system manager or other privileged users.

A RSTS/E user is either privileged or nonprivileged. A privileged user has full access to all the capabilities allowed by the system, and can control operations of the system such as starting up or shutting down the total system. A privileged user can add or delete user accounts, including those of other privileged users, and can designate certain programs as privileged, thereby granting program-selected privileged capabilities to normally nonprivileged users. Privileged users also have access to system utilities that control the use of various features and parameters to make the system operate more efficiently.

### 1.1.3 Interjob Communications

RSTS/E jobs can communicate with each other by sending and/or receiving interjob messages under program control. With the optional DECnet/E feature, they can also exchange messages with other user jobs on remote computer systems tied into the same network.

### 1.1.4 Shared Common Code

Under RSTS/E, user jobs can share data and program code in two ways: run-time systems and resident libraries of software routines. Run-time systems extend the functionality of the monitor. Every user job must run under the control of a run-time system, which acts as an interface between the RSTS/E monitor and the job. Resident libraries, on the other hand, are typically used to extend the functionality of a program; they run under control of the user job, from which they are accessed only as needed.

RSTS/E provides several standard run-time systems, each with its own set of features and characteristics. For example, the BASIC-PLUS run-time system supports and controls all programs written in that language, while the RSX run-time system can emulate a subset of the system directives of the RSX-11M operating system. User-written run-time systems can also be implemented.

The software routines that comprise a resident library must be written in the MACRO-11 assembler language and must run under the RSX run-time system or one of its derivatives. RSTS/E provides no resident library support for programs written in BASIC-PLUS.

It is also possible, through use of the task builder utility (TKB), to create cluster libraries — that is, resident libraries that share the same address window in the user job's virtual address space. For example, the FMS-11/RSTS, COBOL-81 and RMS-11 resident libraries can be clustered in such a way that they use only eight K-words of virtual address space. Unclustered, they would occupy twenty K-words.

### 1.1.5 Batch Processing

Using the system utility BATCH, RSTS/E users can submit batch jobs to perform tasks that require no terminal interaction. Users can also submmit batch jobs to run programs at a later time, such as outside normal working hours. For each batch job, the user can set a limit on the amount of CPU time or elapsed time used and can request error checking.

The system manager starts up and controls the batch processors. There can be a maximum of eight batch processors per system. Each batch processor requires a pseudo-keyboard and two jobs — one for itself and one for the task being executed.

## 1.2 Device Control

RSTS/E supports several types of peripheral devices — disks, magnetic tapes, terminals, line printers, and so on.

### 1.2.1 Disks and File Processing

RSTS/E is a disk-based system requiring at least ten megabytes of file-structured disk space.

#### 1.2.1.1 Disk Types -

RSTS/E supports two logical types of disk structure: public and private. The public disk structure consists of a system disk and (optionally) additional public disk packs or cartridges. These additional disks are considered a logical extension of the system disk.

The system disk contains monitor code, the system initialization code, the primary run-time system, and the control files for system startup and crash recovery. Some installations also use the system disk for storage of active user jobs that are temporarily swapped out of memory. Remaining space on the system disk is used for auxiliary run-time systems, optional system files and CUSPs. This space is also available for general storage of user programs and data files.

The system disk must be physically online and logically mounted whenever the system is running. All other public disks that users may need to access should also be physically online and logically mounted.

Any disk drives not devoted to the public structure can be used for private disk packs or cartridges. Unlike public disks, private disks can be physically and logically dismounted and moved to other drives during timesharing. Users have access to mounted private disks only if they have an account on those disks, or if the protection codes of the files in other accounts permit access. This makes it possible to restrict disk storage to a defined set of users, a distinct advantage when numerous projects are in progress.

The file structure of a private disk is the same as that of a public disk.

#### 1.2.1.2 File System -

RSTS/E disk files can be created, deleted and renamed using commands issued from a user program or commands typed at the user's terminal. They can also be updated or extended under program control. The file processor permits files to be created and extended dynamically by allocating space on the specified disk wherever it is available. Alternatively, a file can be created at a specific location on the disk and/or preallocated to use only physically contiguous space.

The RSTS/E on-disk file structure is arranged as a hierarchy consisting of a master file directory (MFD), group file directories (GFDs), user file directories (UFDs), and files. Each disk initialized for use on a RSTS/E system contains a master file directory. The MFD catalogs GFDs which, in turn, catalog UFDs. UFDs catalog files. The MFD/GFD structure on the system disk also catalogs those accounts under which a user can log onto the system.

Up to 255(10) user accounts can be created in each of up to 254(10) groups. Thus, each disk can theoretically contain up to 64,770(10) user accounts. However, since each account directory requires disk space, the actual limit of user accounts is much lower, depending on disk size and usage.

A user file directory catalogs all files within a user account and contains all retrieval information for the files. Files are pure data and contain no linkage or structural information. Each file in a UFD is distinguished from other files in the same UFD by a unique file name and type. Since a single file cannot span multiple disk packs or cartridges, a disk file's size is limited only by the storage capacity of the volume on which it resides. Thus, the number of files a UFD can contain depends on the size of the individual files and the size of the disk.

Access to a disk file is governed by its protection code which specifies the read and write access rights for the file's owner, for other users within the owner's group, and for all other users. In addition, a single user can own an entire private disk, thereby denying access to files on that disk, regardless of protection code.

Files can be accessed simultaneously by multiple users. When opened for shared update, a single file can be updated by several users at the same time. The file processor uses a block interlock mechanism that to prevent different users from updating the same part of a file at the same time.

#### 1.2.1.3 Data Caching -

Data caching is an optional RSTS/E feature that minimizes disk accesses for frequently used data by keeping such data in a software-maintained "cache" — a specially designated area of system memory. The data retained in this cache can be restricted to disk directory blocks only, or it can include data from disk files. In the latter case, a privileged user can allow all disk files to be cached or only certain eligible files.

Data from disk files should be cached only on systems with greater than 256 K-words of memory.

#### 1.2.1.4 Backup -

RSTS/E provides the ability to back up disk data to an auxiliary backup medium — either disk or magnetic tape. A privileged user can specify an entire disk to be backed up, or he can selectively specify accounts and/or individual files. Selective backup can be done online at any time. Backing up an entire disk, however, requires either the disk to be logically dismounted or the backup process to be done offline. Since the system disk must always be logically mounted, it can only be totally backed up offline.

### 1.2.2 Magnetic Tape Processing

RSTS/E magnetic tape support permits processing of 9-track magtapes with densities of either 800 or 1600 BPI (bits per inch). One of two different labeling formats can be used: DOS-11 or ANSI.

DOS-11 format is used for data exchange between PDP-11 systems: RSTS/E, RT-11, RSX-11M, RSX-11M-PLUS, IAS, and VAX/VMS.

ANSI format (defined by the American National Standards Institute) is used for data exchange with the above systems, as well as with other computer systems that support the format. RSTS/E implements a subset of the ANSI format, processing only volume-header, file-header, and end-of-file labels. RSTS/E performs no access checking. Files can be processed with either fixed- or variable-length record formats.

With the exception of PIP, RSTS/E utilities process single-volume tape file sets. PIP, however, can process volume sets consisting of more than one tape reel.

A tape volume is considered private to the job that has access to it.

### 1.2.3 Terminal Handling

Every RSTS/E system must include at least one terminal — the system console terminal. Potentially, a system can have as many as 126(10) additional terminals and pseudo-keyboards.

The RSTS/E terminal handler is designed for interactive environments. Its major features include the following:

- Full-duplex communications
- Modem control
- Type-ahead with immediate echo
- Programmable echo control
- Multiterminal I/O for individual jobs
- Pseudo-keyboard capability

The optional echo control feature allows programs to handle terminal input one field at a time and to retain control of the screen display. This feature gives application programs the capability of simulating block mode input on terminals. RSTS/E does not support actual hardware block mode data transfers.

The optional multiterminal service feature allows one job to control multiple terminals on one logical channel. This permits several terminals, all performing the same function, to be controlled by one user program without tying up several jobs or I/O channels.

Pseudo-keyboards are logical, rather than physical, devices. They have input and output buffers to and from which a program can send and receive data, but they have no real terminal associated with them. Using a pseudo-keyboard as a communications device, a program can start up and control another job. The controlled job is not aware that it is not dealing with a real terminal. Each RSTS/E system includes at least one pseudo-keyboard. The maximum number of terminals and pseudo-keyboards a system can have is a combined total of 128(10).

### 1.2.4 Line Printer Spooling

RSTS/E supports two line printer spooling packages. Both packages permit users to submit files to be printed either by typing a keyboard command or by issuing a system directive under program control. Both packages permit a user to specify multiple copies of a single file with one command and to use a variety of forms control.

The larger spooling package supports up to eight spoolers per system. Each spooler, along with the queue manager, requires one job slot. Each spooler can be dedicated to a single line printer, or it can be set up to handle general printing. Spooled files cannot contain variable length records, must be of sequential file organization, and must be less that 32,767(10) blocks long.

The smaller spooling package is designed primarily for use on small RSTS/E systems (particularly micro-RSTS) and supports only one spooler per system. The spooler and the queue manager share one job slot. The spooler can control up to two line printers and can handle any RMS file organization.

While both spooling packages can be used on the same system, only one package can be addressed directly from DCL. The system manager determines which is used. Both packages can be accessed by the Concise Command Language (CCL) interface.

Both printer spooling packages support keyboard spooling to hardcopy terminals, such as receive-only printers like the serial LA180 and the LA120-RA. However, data integrity is not checked on the serial line.

## 1.3 Operational Control

RSTS/E provides several levels of operational control. System users generally interact with their jobs through commands typed at the terminal keyboard. The system manager (or any privileged user) can access several system utility programs to start and stop the system and to optimize system efficiency.

### 1.3.1 Keyboard Monitors and Command Languages

Immediately after a user logs onto the system, his terminal is under control of the system's default keyboard monitor. At that point, the terminal is said to be at the system command level because the user can type a system command and the keyboard monitor will process it accordingly. The keyboard monitor examines each ASCII text line entered and determines whether the line is a system command. If it is, the command is executed immediately. The terminal returns to the system command level when a program terminates execution or when a Control-C is typed at the terminal.

There are four standard keyboard monitors supported by RSTS/E: DCL, RSX, RT-11, and BASIC-PLUS. Each of these keyboard monitors interprets a standard set of system commands, permitting users to perform all the fundamental functions of the RSTS/E system — logging on and off, running programs, and so on. Each keyboard monitor also has additional commands that allow the user to perform functions appropriate to that keyboard monitor's environment.

The DCL keyboard monitor is based on the DCL (DIGITAL Command Language) environment available on most DIGITAL operating systems. It is implemented under RSTS/E as a subset of the DCL available under VAX/VMS.

The RSTS/E Concise Command Language (CCL) facility is not a real keyboard monitor, but it allows the system manager to define additional commands to run system utilities, as well as other user programs. A CCL command is an abbreviation for the command to run a particular program. Each CCL command definition specifies the full form of the RUN command, the disk file containing the program to invoke when the CCL command is typed, and any entry parameters for the program.

### 1.3.2 System Management

The system manager and other privileged users have access to many system utilities or CUSPs (Commonly Used System Programs). These CUSPs control the day to day operation of the overall system.

Using the offline system initialization program (INIT.SYS), the system manager can tailor the RSTS/E monitor by allocating memory usage, specifying system default parameters and software components, and enabling and disabling devices. INIT.SYS also permits the system manager to initialize new disk packs for system use and to verify the integrity of existing disks.

Various other online CUSPs provide access to functions such as the following:

- Defining system startup and auto restart control files
- Adding or removing swap files, run-time systems and resident libraries
- Logically mounting and dismounting disk packs
- Adding or removing CCL commands
- Defining terminal characteristics
- Changing job priorities and other job-related parameters
- Adding or deleting user accounts
- Changing account passwords
- Processing system statistics
- Controlling usage of the disk data cache

### 1.3.3 System Reliability

RSTS/E provides an error logging mechanism to record certain classes of hardware errors: memory parity errors, disk and magnetic tape errors, and so on. This error logging facility processes system traps that occur in the monitor, along with other events such as system startup, shutdown, and auto restarts. These events are recorded in a disk file that can be printed and analyzed online.

If the hardware detects a power failure, systems equipped with the necessary hardware backup options will execute a controlled shutdown. When power is restored, the system automatically reloads itself from the system disk and reinitiates system operations by doing a "cold" (or "auto") restart. The system resumes operation without operator intervention, but any users previously logged in must log in again.

If nonrecoverable hardware or system software errors result in a system crash, RSTS/E will attempt to dump the read/write portion of the monitor's address space to a disk file. It then attempts to reload itself from disk and execute an auto restart.

The system manager can disable either of these capabilities.

# Part II
# THE MONITOR

The monitor makes the RSTS/E operating system what it is — a resource sharing, timesharing system. It performs the following functions:

- Management of the timesharing of system resources
- Management of the use of available memory
- Memory mapping functions as required for transfer of control and data communications
- Manipulation of all on-disk directory structures
- Support of user trap processing and other service functions
- Support of interjob communications
- Support of all generic device functions
- Handling of all device-dependent operations

This part of the *RSTS/E V8.0 Internals Manual* discusses the general structure and operation of the RSTS/E monitor. Chapter 2 lists the major monitor components and discusses how they are laid out in memory and how they interact. Details of the job control data structures and the operation of the scheduler are discussed in Chapter 3. Chapter 4 covers memory control and the operation of the memory manager and the swap manager. The user interface to general monitor services is discussed in Chapter 5, and Chapter 6 covers general device control. Other monitor services are covered in later parts.

# CHAPTER 2
# Monitor Structure [V8.0]

This chapter discusses the major components of the monitor (Section 2.1), how they are mapped into memory (Section 2.2), and how they interact (Section 2.3).

## 2.1 Major Components

There are basically seven major software components within the RSTS/E monitor.

- *Monitor Dispatch Routines.* These routines act as the interface between the user jobs and the various monitor services. They handle the initial processing of all EMT calls from the user.
- *Scheduler.* This module determines the order in which user jobs will run, based on job status and priority.
- *Memory Manager.* This module handles memory allocation, determining where in memory a user job or run-time system will reside, based on available space and the "swappability" of currently resident user modules.
- *Swap Manager.* This module handles all swapping of user modules, both into and out of memory, determining where within the swap files the user modules will reside and initiating the required disk transfers through the disk I/O subsystem.
- *File Processor.* This module handles all requests to manipulate the on-disk directory structure. It also processes requests for a variety of additional non-file-related services such as sending and receiving messages, logging on and off the system, setting terminal characteristics, and so on.

  NOTE

  > A commonly held misconception is that the file processor handles file I/O. This is wrong. The file processor deals only with the on-disk directory structure. Reading and writing of file data is done by the disk I/O subsystem.

- *I/O Routines.* These routines handle all requests to perform I/O to the various peripheral devices on the system, acting as an interface between user jobs or other monitor modules and the actual physical device drivers. These routines also perform functions common to all devices. Of particular interest here is the disk I/O subsystem that handles read and write requests to all disks.
- *Device Drivers.* These modules handle the device-dependent operations for each device on the system. They process all device interrupts and perform the actual input or output functions for all devices. Of particular interest here is the terminal driver, which also handles job creation in response to user input at a terminal keyboard or a system directive to spawn a job.

Figure 2-1 shows the *conceptual* relationship between these components. Note, however, that the hierarchy and interfacing between components are not as simple and 'clean' as the lines on the diagram suggest. The software modules that make up these components do not necessarily exist as self-contained contiguous pieces within memory, and the flow of data and processing control between the components can be very complex.

In addition to the above processing modules, there are several data structures required by, and considered part of, the monitor.

- *Interrupt Vectors.* The interrupt vectors are located in the very low end of memory, as dictated by the PDP-11 hardware architecture. Every hardware device capable of interrupting the processor has at least one unique pair of locations (two words) reserved for its interrupt vector. The first word contains the location of the device's service routine; the second word contains the processor status word (PS) to be used by the service routine.
- *Low Core.* This group of memory locations contains a variety of fixed system-wide parameters, such as the maximum number of jobs allowed during timesharing, directory and data caching parameters, and so on.
- *Monitor and File Processor Stacks.* These two memory areas are used to store intermediate values. The file processor maintains an additional stack, apart from the general monitor stack, for storage of intermediate values during the processing of asynchronous functions.
- *Device Data Blocks (DDBs).* Every nondisk peripheral device has its own permanently assigned DDB, containing such information as its driver index, device characteristics, and current owner job. Device data blocks are discussed in detail in Section 6.3.
- *Monitor Tables.* These tables are permanently allocated at system generation time and contain information such as job status, device names, and device driver addresses.
- *Buffers.* The monitor uses several types of buffer structures.
    - *General Small Buffer Pool.* Data structures that hold transient information (such as job control blocks, queue entries, and run-time system and resident library control blocks) are dynamically created using small (32(10)-byte) buffers from this pool.
    - *File Processor Small Buffer Pool (FIP Pool).* Small (32(10)-byte) buffers from this pool are used (primarily by the file processor) to hold Concise Command Language (CCL) control blocks, file control blocks (FCBs), and window control blocks (WCBs).
    - *Extended Buffer Pool (XBUF).* XBUF consists of two separate buffer pools containing large (64(10)-byte) buffers. It is primarily used to retain directory information (directory caching) or blocks of data from disk files (data caching). It is also used for data buffers by DECnet, the 2780 emulator, and the local send/receive monitor function.

In addition to these data structures, the monitor "owns" the I/O page.

This four K-word area is always located at the very top of every PDP-11's addressable memory space. As the name implies, the I/O page is used to access the peripheral I/O devices which are (or can be) attached to the PDP-11. Device data and status registers are located here, as well as certain other registers relating to general CPU operation.

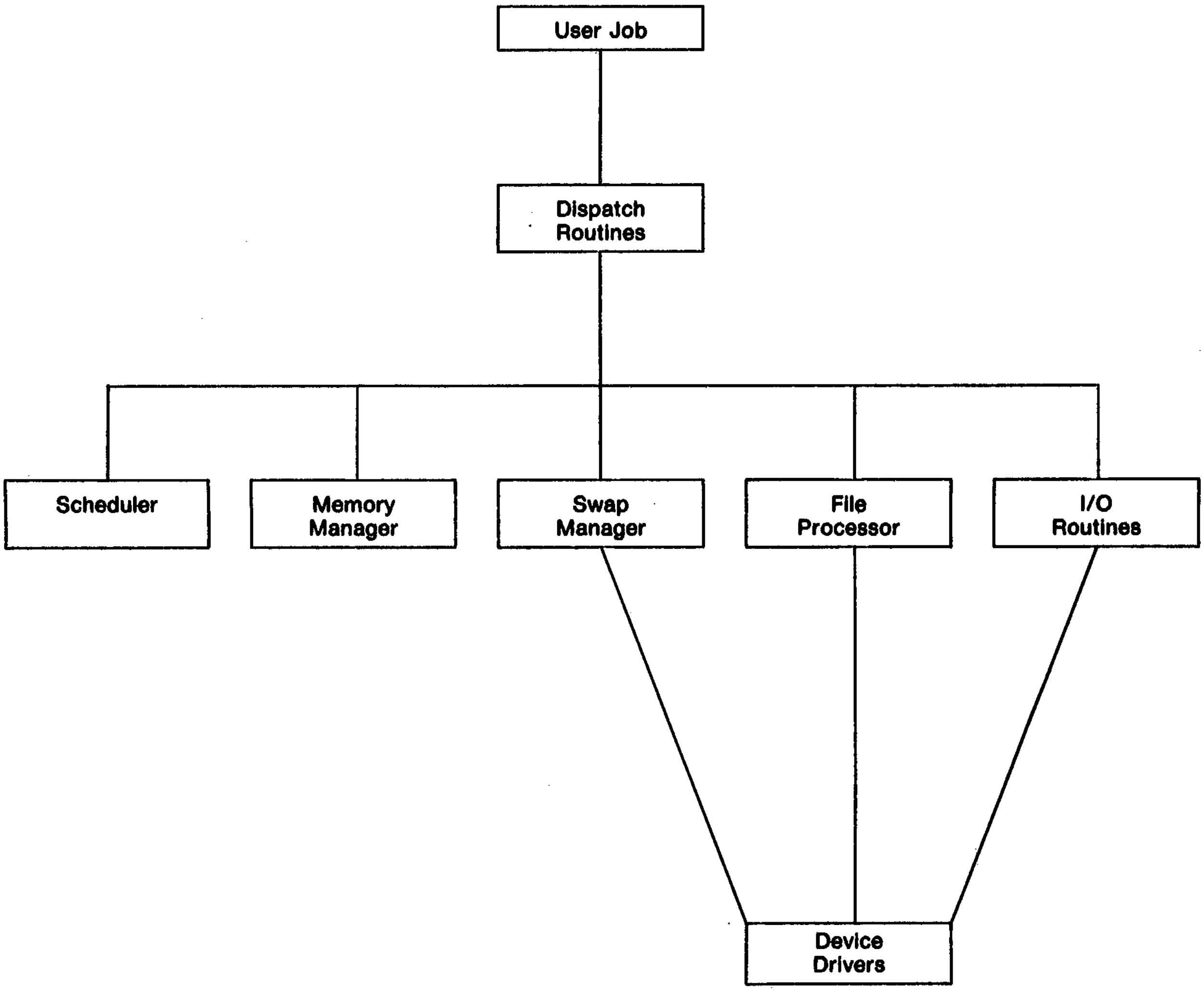


**Figure 2-1:** Conceptual Overview of RSTS/E Monitor

## 2.2 Memory Usage

All PDP-11 computers have sixteen bits in a word. Therefore, the maximum addressable memory space of all models, regardless of size or feature level, is 32 K-words. However, the RSTS/E monitor itself can be over 40 K-words long, depending on overall system configuration, with the I/O page adding still another four K-words. In addition, the monitor must be able to access portions of the user job space in order to transfer data during device I/O as well as during the execution of other monitor services.

The solution to this problem of limited address space lies in the use of the memory management hardware and the dynamic segmentation, or "mapping," of various pieces of the monitor.

### 2.2.1 Memory Management Hardware

RSTS/E uses the memory management hardware feature of the PDP-11 processor. With this feature, the central processor can operate in either of two modes: kernel mode or user mode. When the CPU is operating in kernel mode, the software can execute any valid PDP-11 instruction. In user mode, however, certain instructions, such as the HALT instruction, are prohibited.

NOTE

> The larger PDP-11 processor models provide a further mode of operation known as supervisor mode. This mode is generally not used by RSTS/E.

For each CPU mode there exists a set of eight active page registers (APRs), numbered 0 through 7. Each APR is capable of addressing (or "mapping") up to four K-words of contiguous memory. Thus, for each mode, a "virtual machine" is available of up to 32 K-words of addressable memory (8 APRs x 4 K-words).

Each APR consists of a page address register (PAR), defining the physical position in memory of the start of the page, and a page descriptor register (PDR), defining the length of the page. The PDR also specifies the page access code (PAC) for that page — either read-only access or read-write access.

It is important to realize that the mapping invoked when using the memory management hardware does not involve any actual movement of data. A mapping simply describes the physical pieces of memory in use.

APRs are located within the I/O page, and their contents can be changed by any program that has read-write access to that address space. In RSTS/E, only the monitor has such access.

The memory management hardware provides three important benefits: protection, relocation, and segmentation.

- *Protection.* The RSTS/E monitor operates in kernel mode, while user jobs operate in user mode. In this way, the overall system is protected from a user program that might attempt to execute a HALT or some other prohibited instruction. In addition, the monitor uses the page access code in the page descriptor register to selectively protect certain portions of memory with read-only access. In this way, system-wide shareable code (such as run-time systems, resident libraries, and parts of the monitor itself) is protected from inadvertent corruption.
- *Relocation.* The use of APRs permits each job to be treated as if it occupied the lowest locations in memory. Each page address register acts as a relocation register containing an offset. When a virtual address is referenced, the offset is automatically added to calculate the actual physical address. Through this use of relocation, up to 2 million words of memory can be accessed.
- *Segmentation.* The use of APRs also permits a user job's address space to appear contiguous when, in fact, its various pieces can be widely spread across physical memory.

In addition to these basic features, certain PDP-11 processor models support an expanded memory management capability known as "instruction and data (I&D) space," or simply "data space." Processors that support data space have two sets of page registers for each processor mode (kernel, user, and supervisor). One set, the instruction mapping registers, is used for memory references using the program counter (PC). The other set, the data mapping registers, is used for other memory references. This permits machine instructions to be in one virtual address space and data to be in another.

[In general, to qualify for such treatment, a software module must access no local data and must not execute subroutine calls using in-line arguments. Note, however, that these are not the only criteria. The use of data space requires adherence to strict coding conventions. A discussion of these conventions is outside the scope of this document.]

The use of data space can be enabled separately for each processor mode. If it is not enabled, the instruction mapping registers are used for both instructions and data.

When data space is used by RSTS/E, the monitor can make double use of part of its virtual address space. Many of the common subroutines in the permanently mapped root are mapped in instruction space, but do not need mapping in data space. Thus, the corresponding data register can be used to map extra small buffers.

### 2.2.2 Memory Mapping

The use of the memory management hardware is standardized in the RSTS/E system. The monitor routines are mapped using the kernel mode APRs, while user jobs are mapped with the user mode APRs. In addition, within each mode, the use of individual APRs is standardized.

#### 2.2.2.1 Monitor Mapping -

To operate within the limitation of a 32 K-word address space, the RSTS/E monitor uses a form of "overlay" structure consisting of a permanently mapped "root" and memory-resident "phases" that are dynamically mapped as needed. The root consists of certain system data structures that must be permanently accessible to all portions of the monitor — the monitor stacks, device control blocks, job status tables, the general small buffer pool, and so on — as well as the monitor's central control code. The dynamically mapped phases, on the other hand, consist of those modules and data structures that are needed only for particular functions.

Figure 2-2 shows a conceptual diagram of monitor memory mapping. You may want to refer to this figure frequently throughout the remainder of this section.

A phase is basically a collection of code modules that are linked together at SYSGEN time, using the LINK utility, to form a single save-image file. (Optimally, the collection of modules within a phase are related — the DSK phase contains disk driver code, the TER phase contains terminal service code, and so on.) The resulting save-image files are then combined into one save-image library (SIL) using the SILUS utility. Due to the structure of the SIL header, the maximum number of phases is restricted to fifteen.

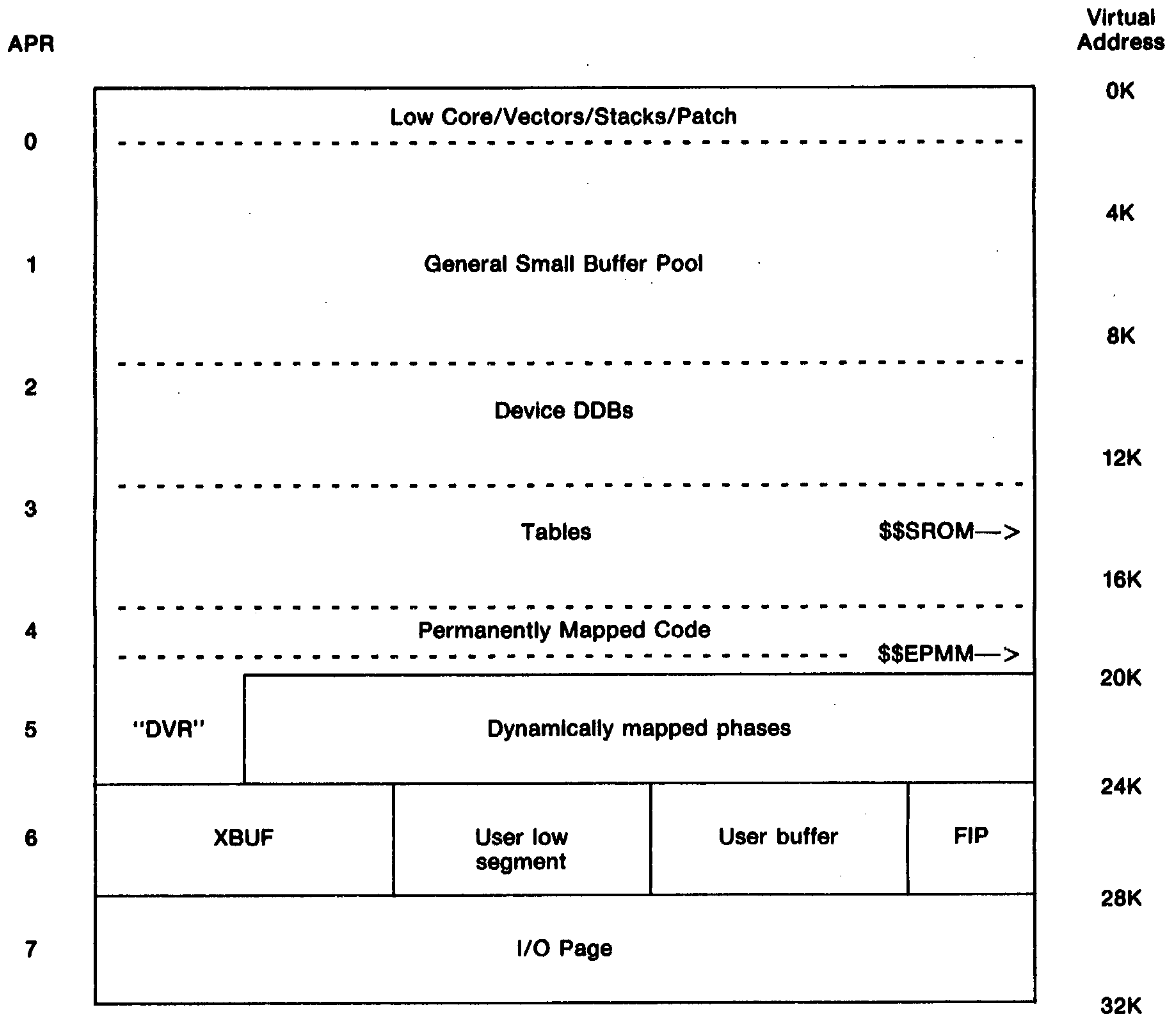
In general, a phase is less than or equal to four K-words in size — that is, the amount that can be mapped with one APR. One major exception to this is the RSTS phase, which can be up to 24 K-words long. The first twenty K-words of the RSTS phase contain the permanently mapped root, while the remaining four K-words contain device drivers, portions of the disk I/O subsystem, and other random modules that do not need to be permanently accessible. This four K-word segment is dynamically mapped along with the other phases and is sometimes referred to as the "driver phase" ("DVR"). This term is a misnomer, however, because the modules in this section do not exist as a separate module within the monitor SIL, but are linked together with the root as part of the RSTS phase.

The first five kernel page registers (APR 0 to APR 4) are always used to map the monitor root. APR 5 is dynamically mapped among the various phases (except for the FIP phase, discussed in Section 2.2.2.2), as well as the upper four K-words of the RSTS phase. When APR 5 is used to map a non-FIP phase, APR 6 is generally used to dynamically access information in user programs or the extended buffer pool, XBUF. APR 7 is normally mapped to the I/O page.

When the data space feature is enabled on those processors supporting it, sections of the permanently mapped root within the RSTS phase are mapped with only one of the corresponding I&D mapping registers. Since the low core area of memory contains the monitor patch space, along with the "one-shot" startup code, it requires both instruction and data mapping. However, the remaining data areas — small buffers, DDB's, and monitor tables — require only data space mapping. Similarly, while some of the root code needs both instruction and data mapping within the same virtual address space, a portion of the root code does not. Thus, these portions of code are consolidated and mapped using only instruction space. The corresponding data space is available for mapping additional small buffers.

The RSTS/E monitor uses the page access code of the page descriptor register to selectively protect certain portions of kernel mode memory. The lower section of the RSTS phase is designated read-write since it contains many locations and buffer areas that change dynamically during timesharing. Likewise, APR 6 and APR 7 are designated read/write. The remainder of the monitor root (locations above symbol $$SROM) is designated read-only to protect the static tables and processing code from being accidently corrupted.

Note that even though symbol $$SROM is depicted in Figure 2-2 as being somewhere between 12 and 16 K-words, the start of read-only memory will always, in practice, fall on a four K-word boundary. To ensure that no virtual address space is wasted, kernel mode pages are always exactly four K-words long, beginning on a four K-word boundary within physical memory. Thus, in the example shown in Figure 2-2, memory locations between $$SROM and sixteen K-words are, in fact, read-write.



$$SROM = Start of read-only memory
$$EPMM = End of permanently mapped memory

**Figure 2-2:** RSTS/E Monitor Mapping

### 2.2.2.2 File Processor Mapping -

Since the file processor handles so many of the monitor service functions, it is much larger than four K-words. Thus, mapping of this portion of the monitor is different from and more complicated than that of other phases. In fact, the file processor is mapped in a fashion very similar to that of the overall monitor itself, with a static root portion and dynamically mapped "overlays." Unlike the monitor, however, all of these overlays (or service routines) are not necessarily memory-resident. When physical memory is limited, portions of the file processor code frequently reside on disk and are brought into memory only when they are needed for execution.*

Figure 2-3 shows the memory management mapping during execution of the file processor. APR 6 is used to map the file processor root code (CTL) and small buffer pool (FIPOOL). These two segments are known as the FIP phase. CTL is the main FIP control module,

* In fact, the file processor is the only portion of the monitor capable of using disk-resident overlays. This is the major reason that the file processor is responsible for so many non-file-related functions; other monitor components do not have the ability to store less frequently executed routines on disk.

responsible for mapping and dispatching to the proper service routine for a particular function. FIPOOL is basically an extension of the general small buffer pool and is used (primarily by the file processor) to supply buffers for Concise Command Language (CCL) control blocks, file control blocks (FCBs), and window control blocks (WCBs).

APR 5 is used to dynamically map the file processor service routines — both memory- and disk-resident. The system initialization code (INIT.SYS) allocates physical memory for the memory-resident routines, along with a disk overlay buffer (OVRBUF), during system startup. OVRBUF is used to hold any disk-resident routines that are required in memory for execution. Since this buffer is 1000(8) bytes long, all disk-resident routines must be less than 1000 bytes. In fact, since any service routine can optionally be made disk-resident, all service routines must be less than this length.

When data space is enabled on those processors that support it, the FIP phase is mapped so that CTL is mapped only in instruction space, leaving a full four K-words of data space for FIPOOL. The service routines are mapped in both instruction and data space.

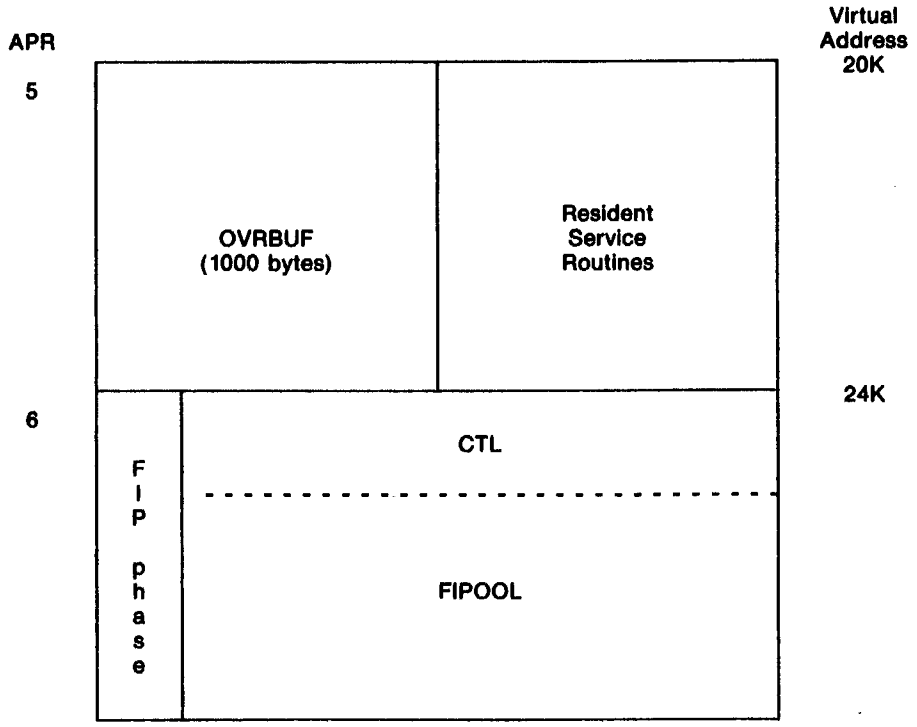


**Figure 2-3:** FIP Mapping

### 2.2.2.3 User Mapping -

A user job consists of three distinct portions:

- The low segment (or job image), containing the user program code and dynamic data areas necessary for passing information between the program, the run-time system, and the monitor
- The middle segment, containing zero to five resident libraries
- The high segment, containing the run-time system and static data areas used by the monitor in controlling run-time system execution

These three portions of user job space are discussed in detail in later chapters.

When assigning the user mode active page registers, the monitor always maps the lowest locations of the user job image with APR 0 and the highest locations of the run-time system with APR 7. One or more of APR 1 to APR 6 can be used to map the remaining user space, depending on the size of the user program, the run-time system, and the number of resident libraries used, if any. (If the user job is running under the "disappearing" RSX run-time system (see Section 15.2.2), APR 7 is also available to map user space or resident libraries.)

Since the first 1000(8) bytes of the user job image always contain standard dynamic data areas, and the program running in the job's low segment can have its own dynamic data, the APRs used to map the job's low segment are always designated read/write.

Normally, run-time systems and resident libraries are shareable by many user jobs and, therefore, should be written in reentrant code. Any data or intermediate values necessary during processing should be stored in the user job image or on the user stack, thereby keeping the shareable code "pure." (For this reason, the user job image is frequently referred to as the "impure" segment and the run-time system and resident libraries together are referred to as the "pure" segment.) One way the monitor ensures that such shareable code remains uncorrupted is to designate the appropriate APRs as read-only. If necessary, however, it is possible to specify a run-time system or resident library as read-write. Normally, this is done only with a single-user run-time system or during debugging.

## 2.3 Component Interaction

There are two basic ways in which the various monitor components can interact with one another:

- Processing code in one phase can access subroutines in other phases using a special form of subroutine linkage.
- One monitor function can "schedule" another using what is known as the "level three queue" mechanism.

### 2.3.1 Subroutine Linkage (CALLMI)

When processing code in one phase needs to access a routine in another (unmapped) phase, it does so by calling routine CALLMI (CALL Mapped Indirect) in the root. This routine remaps APR 5 to the appropriate phase, does a table lookup for the proper address, and then transfers control to the desired routine. Any data required is passed to the routine, either through the permanent structures or in space mapped by another APR. Each phase must know where to jump in the root, and the root must know where to jump in the phases. Both these cases are handled during system generation when the monitor SIL is built.

The first case (knowing where to jump in the root) is handled through the LINK utility. By linking the monitor root first, the SYSGEN process can supply the root phase symbol table to LINK for use in building subsequent phases.

The second case (knowing where to jump in the phases) is handled through the merge feature of SILUS. Using this feature, any SIL module can pass information back to another (earlier) module in the same SIL. Thus, any phase that contains locations that must be recorded in the root can specify the proper values to the SILUS utility which then merges those values back into the root phase.

### 2.3.2 Component Scheduling (L3QUE)

Scheduling of the various monitor components is done through the "level three queue" mechanism. This mechanism takes advantage of the central processor hardware priority structure.

The PDP-11 central processor can operate with eight different priority levels, numbered 0 to 7. When the CPU is operating at a particular priority level, only those interrupts with a higher designated priority are honored. For example, if the current priority level is five, only interrupts with a designated priority of six or seven will be processed. The CPU priority is incorporated into the processor status word (PS).

Within the RSTS/E system, user programs operate at a priority level of zero. The null job runs with a priority level of one, and normal monitor processing is at level three. Hardware interrupts and special critical functions operate with priority levels of four to six. (RSTS/E does not use priority level two or support any priority seven devices.) The level three queue mechanism uses this hierarchy.

All monitor processing and interrupt servicing routines exit through the level three queue processing code — a module in the monitor root called RTI3. (Note that even user requests to the monitor, using the EMT call, are interrupts.) On entry to this code, the previous hardware priority — that is, the operating priority level before the interrupt — is checked. If that level was greater than two, some monitor processing was interrupted and control returns to the interrupted function.

However, if the previous priority was less than or equal to two, either a user program or the null job was interrupted. Before returning to the user, RTI3 tests two flag words (L3QUE and L3QUE2) containing bit masks of monitor functions that need to run. These two flag words are structured so that bits set in L3QUE are of higher priority than those set in L3QUE2 and less significant bits are of higher priority than more significant bits (see Figure 2-4).

Using the highest priority bit that is set as an index, RTI3 retrieves the address of the necessary module from table L3QTBL. Control is transferred to the module. The module called will, in turn, exit through RTI3. When there are no more monitor functions that need to run, control is returned to the user.

This scheme works quite nicely. If the service routine just completed was entered at a level higher than two, then another monitor process must have been interrupted, and that process must be allowed to finish. Control will be returned to the user job eventually, but all monitor services must be completed first.

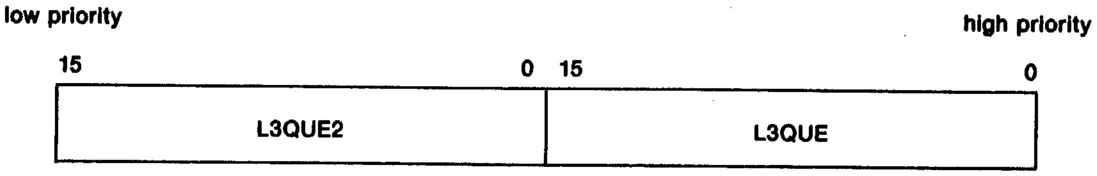


Figure 2-4: Level Three Queue Flag Words

# CHAPTER 3
# Job Control [V8.0]

RSTS/E is designed to run user jobs — the basic unit of work within the system. Thus, job control is one of the most important functions of the RSTS/E monitor. This chapter discusses the data structures used in job control (Section 3.1) and the general details of job scheduling (Section 3.2).

## 3.1 Data Structures

The job control structures are the heart of the RSTS/E system. They hold the information necessary for the monitor to timeshare system resources (such as CPU time, memory, etc.) properly among all users. In addition, they provide access to information necessary for almost every other monitor service, including device and file handling.

The job control structures consist of a combination of tables and blocks. Tables contain one word for each possible job on the system, and thus, their size is determined by the number of jobs configured at SYSGEN time. As jobs are created or removed, information within the tables is changed, but the size of the tables remains constant.

Blocks, on the other hand, exist on a per-job basis. With the exception of the secondary job data block (see Section 3.1.3), they are typically 16(10) words long and generally exist only as long as a particular job exists. Job blocks are generated when a job is first created and deleted when the job is removed.

A job is created when a delimiter is typed at a logged off terminal or when spawned by another job using the "create job" system directive. At that time, a job slot is found in the main job table, small buffers are allocated from the general pool to hold the job blocks, and the basic job structure is built. Note that a job does not have to be logged in to exist.

A job is removed from the system when it is killed or when it is terminated after logging off. A logged off job is terminated when its keyboard monitor issues a read for a system level command from the terminal keyboard. (Job creation and termination are handled by the terminal service module. See Chapter 13.)

### 3.1.1 Job Table (JOBTBL)

The job table is the root of the job control structures. It is permanently assigned at system generation time and contains one 1-word entry for each possible job on the system. Each entry corresponds to a possible job "slot."

Control information for a particular job slot can be accessed by indexing into the job table using the job number, multiplied by two. The value at that location, for an existing job, is the address of the job data block (JDB). If there is currently no existing job associated with that job number, the value in the job table is zero.

The first word of the job table corresponds to the null job (job 0) and always contains a value of zero. The last word in the table always contains a -1 to signify the end of the table.

Thus, the total length (in words) of the job table is the maximum number of jobs, as specified at SYSGEN time, plus two.

### 3.1.2 Job Data Block (JDB)

The job data block (JDB) is pointed to by the job table and is the most easily accessible of the job control structures. It contains the most frequently accessed, or most system critical, information about the job. Specifically, the JDB contains size, memory control and scheduling information. It also contains pointers to other job control blocks for the job.

The JDB is allocated from the general small buffer pool when the job is first created. It is deleted when the job is removed from the system.

Figure 3-1 shows the contents of the job data block. Each entry is discussed below.

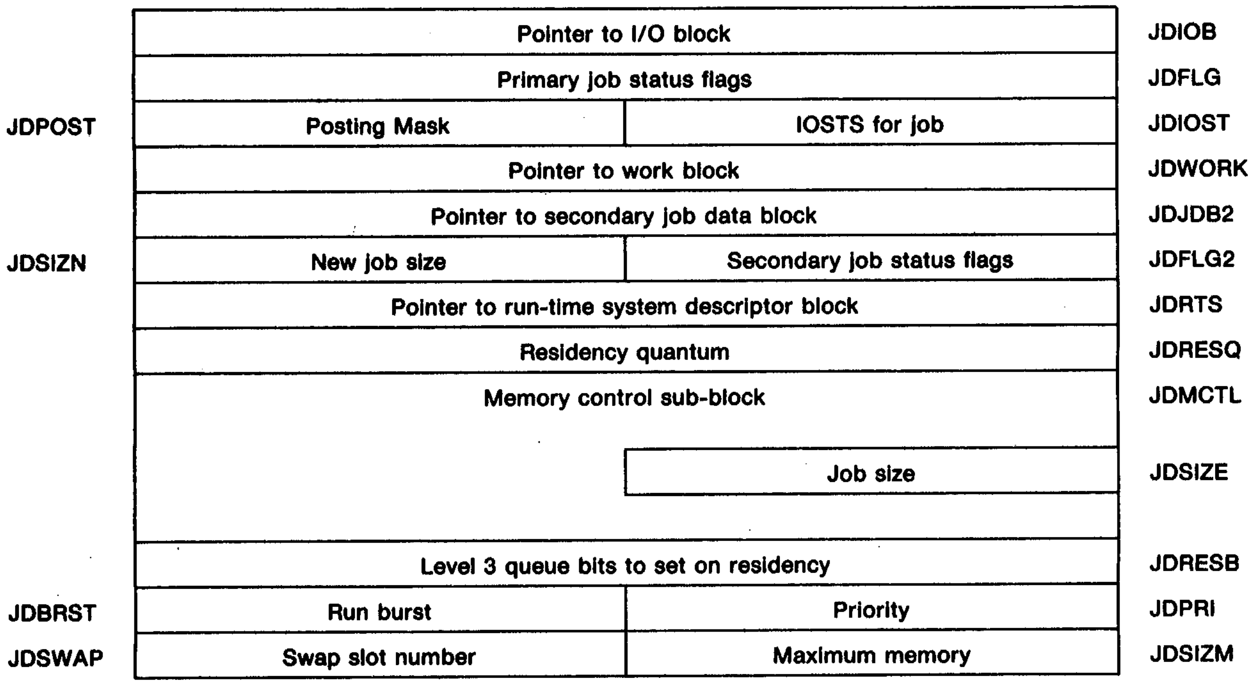
*Offset* *Contents*

JDIOB This word contains the address of the I/O data block (IOB) for the job (see Section 3.1.4). It is filled in when the job is created and remains constant for the life of the job.

JDFLG This word contains the primary status flags for the job (see Section 3.1.2.1).

JDIOST This byte contains the error code to be returned to the job upon completion of the current monitor call. If the JFIOST bit is set in the primary job status word (JDFLG), the monitor moves the contents of this byte (as a word value) to the location FIRQB+0 in the user program area before returning control to the job. Error codes range between 0 and 177(8) inclusive. A value of zero means no error occurred. (See the *System Directives Manual* for a complete list of error codes.)

JDPOST This byte indicates what data should be moved from the job's work block to the FIRQB or XRB in the job image area before the monitor returns control to the job. This process is called "posting" and is discussed in detail in Section 5.1.3. This byte is meaningful only if the status bit JFPOST is set in the primary job status word (JDFLG), indicating that posting should be done.



| | | | |
|---|---|---|---|
| | Pointer to I/O block | | JDIOB |
| | Primary job status flags | | JDFLG |
| JDPOST | Posting Mask | IOSTS for job | JDIOST |
| | Pointer to work block | | JDWORK |
| | Pointer to secondary job data block | | JDJDB2 |
| JDSIZN | New job size | Secondary job status flags | JDFLG2 |
| | Pointer to run-time system descriptor block | | JDRTS |
| | Residency quantum | | JDRESQ |
| | Memory control sub-block | | JDMCTL |
| | | Job size | JDSIZE |
| | Level 3 queue bits to set on residency | | JDRESB |
| JDBRST | Run burst | Priority | JDPRI |
| JDSWAP | Swap slot number | Maximum memory | JDSIZM |

**Figure 3-1:** Job Data Block

JDWORK This word contains the address of the work block (WRK) for the job (see Section 3.1.5). It is filled in when the job is created and remains constant for the life of the job.

JDJDB2 This word contains the address of the secondary job data block (JDB2) for the job (see Section 3.1.3). It is filled in when the job is created and remains constant for the life of the job.

JDFLG2 This byte contains the secondary job status flags (see Section 3.1.2.2).

JDSIZN This byte contains the size (in K-words) that the job should be the next time it is made resident. It is used when a job requests a memory expansion that cannot be done in place. JDSIZN is set to the desired size and the job is swapped out. The memory manager then allocates the additional space when the job is returned to memory.

JDRTS This word contains the address of the run-time system descriptor block (RTS) currently associated with the job. Every job has a run-time system associated with it at all times. If the job is using the disappearing RSX run-time system, JDRTS will contain the address of the null run-time system descriptor block (NULRTS). This word is filled in when the job is created but can change during the life of the job. (See Chapter 15 for a complete discussion of run-time systems.)

JDRESQ This word contains the current residency quantum for the job. Every time the job is swapped into memory, this word is set to some nonzero value. As long as it remains nonzero, the job is not eligible to be swapped out. (See Section 4.2.1.1.)

JDMCTL These five words are the memory control sub-block (MCB) for the job. (Memory control sub-blocks are discussed in detail in Section 4.1.1.)

JDSIZE This word (within the memory control sub-block) contains the current size of the job in K-words.

JDRESB This word contains the bit mask to be posted to the level three queue (L3QUE) when the job is made resident. The bits posted are used by the monitor to notify itself when the job is finally resident after being swapped in. (See Section 2.3.2 for a discussion of the level three queue mechanism.)

JDPRI This byte contains the job's priority. Values can range from -128 to +127(10). The scheduler uses this byte to determine which job to run next. It is set to a system default of -8 at job creation, but can be modified during the LOGIN sequence or with the UTILTY program. (Job priority is discussed in detail in Section 3.2.2.)

JDBRST This byte contains the job's run burst. The run burst is the maximum amount of time the job will be allowed to execute compute-bound before the scheduler is invoked. The value is in units of 1/60 or 1/50 of a second, depending on the type of clock being used. It is set to a system default of six at job creation, but can be modified during the LOGIN sequence or with the UTILTY program.

JDSIZM This byte contains the private memory size maximum for the job. Values can range from 1 to 255 K-words, with a value of 255 indicating that the job can use up to the system job size maximum. It is set to a system default of 255, but can be modified during the LOGIN sequence or with the UTILTY program.

JDSWAP This byte contains the swapped out location of the job. If the value is nonzero, it represents the slot number within the swapping file where the job is located (see Section 4.2.2). If the value is zero, the job is either resident in memory or has no job image. (A job has no job image immediately after it is created.)

### 3.1.2.1 Primary Job Status Flags (JDFLG) -

The job status flags, contained in word JDFLG of the job data block, are used by the monitor to determine what actions it must take before returning control to the job. They are also used to refresh the job's "keyword."

The keyword is located in the job image area and defines the job's status in the timesharing environment (whether it is logged in under a privileged account, locked in memory, and so on). Seven bits are controlled by the monitor; nine are controlled by the job or the run-time system. Copies of six of the seven monitor-controlled bits are maintained here in JDFLG; the remaining bit is incorporated into the job priority byte, JDPRI. (See Section 5.1.2 for more information on the keyword.)

The primary job status bits are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | JFPOST | If this bit is set, the information in word JDPOST is used as a mask for updating the job's FIRQB or XRB (see Section 5.1.3). In addition, the information in J2PPTR and J2PCNT in the secondary job data block (JDB2) can be used to post large amounts of data to a user-defined buffer. (See Section 3.1.3.) |

1 JFIOKY If this bit is set, the monitor-controlled bits of the job's keyword are copied into location 400(8) of the job image area and word JDIOST in the job data block is posted to the job's FIRQB.

2 JFCEMT If this bit is set, the RSX support within the monitor will post the job information indicated by JFIOKY rather than the standard monitor routines.

3 JFCC If this bit is set, a Control-C was typed at the job's terminal. When the job becomes runnable, the pseudo-vector P.CC will be taken, unless JF2CC is also set. (See Section 15.3.)

4 JF2CC If this bit is set, Control-C was typed at the job's terminal at least twice since the job was last run. When the job becomes runnable, the pseudo-vector P.2CC will be taken. (See Section 15.3.)

5 JFPPT If this bit is set, the hardware floating point unit has taken an exception trap. When the job is made runnable, the pseudo-vector P.FPP will be taken. (See Section 15.3.)

6 JFGO If this bit is set, the I/O "redo" condition specified by JFREDO is ignored when the job is made runnable. This bit is set if a Control-C was typed at the job's terminal during an interruptable I/O operation. This is done, rather than clearing JFREDO, in order to avoid a possible race condition.

7 JFREDO If this bit is set when a job appears runnable (and JFGO is not set) then the job is stalled waiting for an I/O completion and is not really runnable. That is, the job has been made resident simply to permit the I/O to continue.

8 JFSYST If this bit is set, the job can use temporary privileges.

9 JFFPP If this bit is set, the contents of the floating point hardware (if any) are saved and restored along with the job image. It is one of the keyword bits.

10 JFPRIV If this bit is set, the job is logged into a privileged account and has permanent privilege. It is one of the keyword bits.

11 JFSYS If this bit is set, the job is currently running with temporary privileges. It is one of the keyword bits.

12 JFNOPR If this bit is set, the job is not logged in. It 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). It is one of the keyword bits.

14 JFLOCK If this bit is set, the job is locked in memory and is not eligible to be swapped. It is one of the keyword bits.

15 JFSPCL If this bit is set, some special or fatal error condition exists for the job and special processing is required before the job can be made runnable. The flag bits in JDFLG2 specify the action to be performed.

### 3.1.2.2 Secondary Job Status Flags (JDFLG2) -

If bit JFSPCL is set in the primary job status word (JDFLG) of the job data block, some special or fatal error condition exists for the job. The monitor then checks the secondary status flags in byte JDFLG2 to determine what action to take.

Error conditions require the monitor to either return an error to the job or kill the job altogether. Killing a job is a two step process. First, the job must be logged out. At that point, the job can be deleted and its data structures can be returned to the general small buffer pool.

If JFSPCL is set in JDFLG but no bits are set here in JDFLG2, the monitor determines whether the system is just starting up. If so, the primary run-time system is entered at either its P.STRT entry point (normal startups) or its P.CRAS entry point (crash recovery startups). If the system is not starting up, the *job's* run-time system is entered at its P.NEW entry point on the assumption that "new user" processing must be done. (See Section 15.5 for a discussion of run-time system entry points.)

The bits are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | JFCTXT | If this bit is set, the job's context is to be saved. |
| 1 | JFPRTY | If this bit is set, the special condition is a memory parity error. |
| 2 | JFRUN | If this bit is set, the job's run-time system is entered at its P.RUN entry point to run a user program. |
| 3 | JFSWPR | If this bit is set, the special condition is a run-time system or resident library load failure. |
| 4 | JFSTAK | If this bit is set, the special condition is a stack overflow. |
| 5 | JFSWPE | If this bit is set, the special condition is a swap error. |
| 6 | JFKIL2 | If this bit is set, the job is being killed and the LOGOUT phase has completed. The job can now be deleted from the system. |
| 7 | JFKILL If this bit is set and JFKIL2 is also set, the job control information can be deleted. If JFKIL2 is not set, set it and begin the LOGOUT sequence for the job. | |

### 3.1.3 Secondary Job Data Block (JDB2)

The secondary job data block (JDB2) is an extension of the primary job data block. It contains information about the job that is used less frequently or is less time critical than information in the JDB. It is primarily used to hold accounting and directory information, along with other miscellaneous pieces of job-related data.

Unlike the other job control blocks, the secondary job data block is not allocated dynamically but is permanently assigned during system generation. There is one block for each possible job, as defined by the system parameter, JOBMAX. In addition, the format of the JDB2 is not the same on all systems but varies according to the requirements of each system's software configuration.

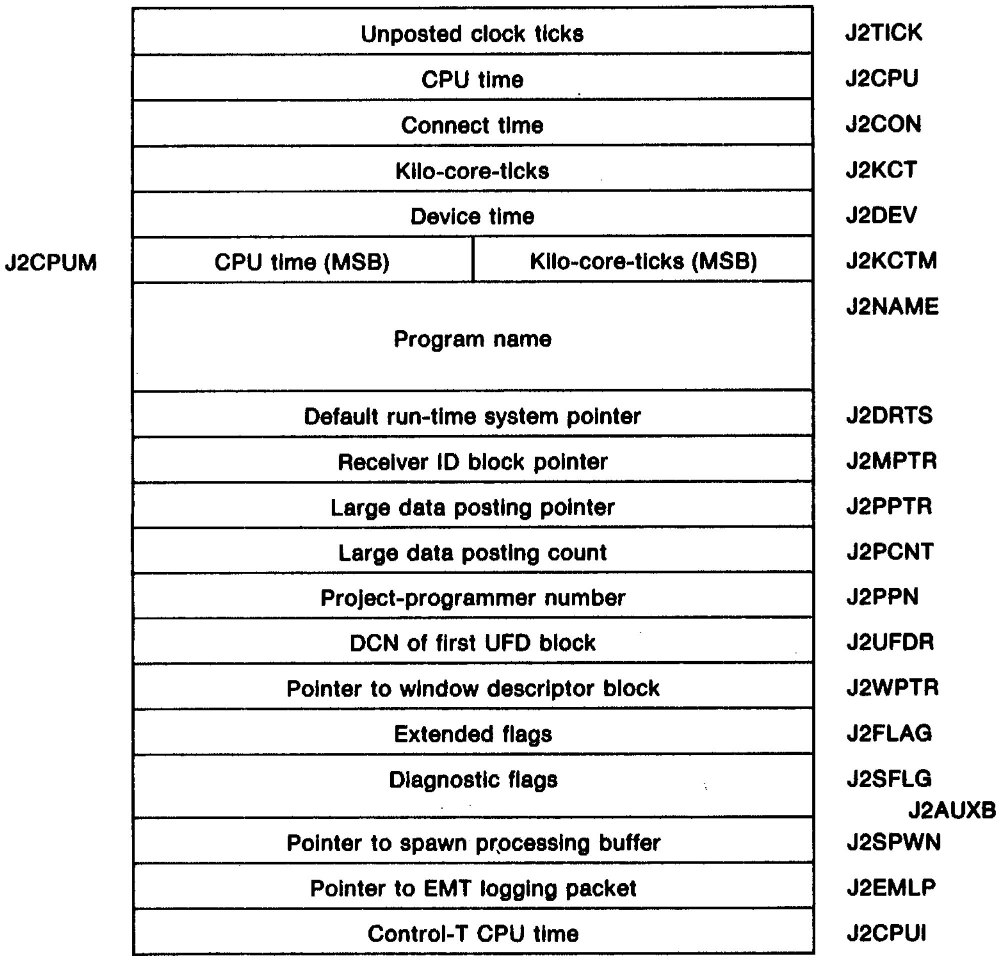
Originally, the JDB2 was dynamically allocated from the small buffer pool, but as more features were added to the monitor, more data was required in the job control structures.

Eventually, in version 8.0, the existing space was filled. Rather than create a JDB3, which would have been both unwieldy and inefficient (using an entire small buffer for the sake of only one or two words), the JDB2 was changed to a permanently assembled part of the monitor table structure. As an added benefit of this new configuration, the secondary job data blocks are now contiguous, permitting the data to be accessed sequentially with a simple scan rather than by working through a linked list mechanism.

Figure 3-2 shows the format of the secondary job data block. Following the figure is a description of each entry.

Not all systems need exactly the same fields in the JDB2. Therefore, the length and format of this block vary from system to system. The first portion (as defined, in bytes, by symbol J2FXSZ) is fixed, while the remaining data items are optional and may not be present on all systems.

The symbol J2AUXB is a special offset within the JDB2. It is used to access those data items that contain large buffer contorted addresses of various dynamic structures that can be created during the life of the job. These buffer addresses are always grouped together in the JDB2 and, for each system, the total number of such addresses is defined by symbol J2ABCT. When the job is removed from the system, the monitor uses these two symbols — J2AUXB and J2ABCT — to access any existing large buffers and return them to XBUF.



| | | | |
|---|---|---|---|
| | Unposted clock ticks | | J2TICK |
| | CPU time | | J2CPU |
| | Connect time | | J2CON |
| | Kilo-core-ticks | | J2KCT |
| | Device time | | J2DEV |
| J2CPUM | CPU time (MSB) | Kilo-core-ticks (MSB) | J2KCTM |
| | Program name | | J2NAME |
| | Default run-time system pointer | | J2DRTS |
| | Receiver ID block pointer | | J2MPTR |
| | Large data posting pointer | | J2PPTR |
| | Large data posting count | | J2PCNT |
| | Project-programmer number | | J2PPN |
| | DCN of first UFD block | | J2UFDR |
| | Pointer to window descriptor block | | J2WPTR |
| | Extended flags | | J2FLAG |
| | Diagnostic flags | | J2SFLG<br>J2AUXB |
| | Pointer to spawn processing buffer | | J2SPWN |
| | Pointer to EMT logging packet | | J2EMLP |
| | Control-T CPU time | | J2CPUI |

J2ABCT = (J2CPUI-J2AUXB)/2
J2FXSZ = J2SPWN + 2

**Figure 3-2:** Secondary Job Data Block

| *Offset* | *Contents* |
|---|---|
| J2TICK | This word contains the number of clock interrupts counted against this job's run time. It is incremented at each clock interrupt while the job is executing. When the job's remaining timing information is collected into the data items below (either upon request or when the job is unscheduled), this value is converted to tenths of a second and added to J2CPU. Any remainder is returned to this word to avoid round-off problems. The units of this word depend on how fast the clock is interrupting. For a clock running at 60 hertz, the units are 1/60 of a second. |
| J2CPU | This word contains the low-order sixteen bits of the total CPU time used by this job through the last time J2TICK was posted. The units of this word are tenths of a second. |
| J2CON | This word contains the total connect time, in minutes for this job. Connect time is only computed while the job is logged in. |

J2KCT This word contains the low-order sixteen bits of the job's kilo-core-ticks. One kilo-core-tick is the use of one K-word of memory while executing for one-tenth of a second. Using two K-words for one-tenth of a second is two kilo-core-ticks, and so on. This unit of measure gives a more accurate picture of the use of system resources.

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, and so on.

J2KCTM This byte contains the high-order eight bits of the job's kilo-core-ticks (see J2KCT above).

J2CPUM This byte contains the high-order eight bits of the job's CPU time (see J2CPU above).

J2NAME These two words contain the program name in RAD50. The program name is specified using the .NAME system call. All the standard run-time systems issue this call to post the program name each time a program is run. The contents of this word are for informational purposes only and are ignored by the monitor.

J2DRTS This word contains the address of the run-time system descriptor block (RTS) of the default run-time system for this job. If the job's default run-time system is not installed when the user program exits, the system default run-time system is used. (See Chapter 15 for a detailed discussion of run-time systems.)

J2MPTR This word contains the address of the job's receiver ID block (RIB). If the job is not a message receiver, this word contains a zero. (See Section 5.2 for a discussion of the system message facility.)

J2PPTR This word is used as a pointer to a monitor buffer to be used to transfer information to or from a user program buffer. It is normally used for large message send/receive buffer transfers. If the lower five bits of the pointer are zero, the pointer is the address of a small buffer from the general pool. If the lower five bits are nonzero, the pointer is the "contorted" address of a large buffer that has been rotated left seven bits (see Section 5.3.3).

J2PCNT This word specifies the number of bytes to transfer to or from the buffer specified by J2PPTR.

J2PPN This word contains the project-programmer number under which this job is running. The high byte contains the project (or group) number and the low byte contains the programmer (or user) number. If the job is not logged into an account, this word is zero.

J2UFDR This word contains the device cluster number (DCN) of the first cluster of the UFD specified in J2PPN. It is filled in during the LOGIN sequence and provides a short cut method for starting a directory search when a job that is logged in requests one of its own files. This word has no meaning if the job is not logged in. (See Chapter 8 for a detailed discussion of the disk directory structure.)

J2WPTR This word contains the address of the job's first window descriptor block (WDB). If the job is not attached to any resident libraries, this word contains a zero. (See Chapter 16 for a discussion of resident libraries and window descriptor blocks.)

J2FLAG This word contains the extended job flags. These flags are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | J2FSPW | If this bit is set, the job was spawned as a logged-in job. |
| 1-15 | | (Unused) |

J2SFLG This word contains bit flags used for internal diagnostics.

J2SPWN This word contains the contorted address of the large buffer (from XBUF) used during the job spawning process. (See Section 5.3.3 for a discussion of contorted addresses.)

J2EMLP This word contains the contorted address of the large buffer (from XBUF) containing the EMT logging packet for this job. It is an optional data item and is only included in systems that support EMT logging.

J2CPUI This word contains the CPU time used by this job as of the last time a Control-T "mini-SYSTAT" was taken. It is an optional data item and is only included in systems that support the Control-T feature.

### 3.1.4 I/O Block (IOB)

The job's I/O block (IOB) is used to access information about each channel that the job has open. It is 16(10) words long, containing one entry for each possible channel.

NOTE

> BASIC-PLUS only allows twelve channels. Channel 0 is the job's terminal and channels 13 to 15 are used internally by the BASIC-PLUS run-time system.

If an entry in the IOB is zero, the corresponding channel is closed. If the entry is nonzero, the channel is open and the word contains the address of the device data block (DDB) for nondisk devices or the file control block (FCB) for disk files. (See Section 6.3 for a discussion of DDBs and Section 9.1.6 for a discussion of FCBs.)

Channel 0, assigned to the job's terminal, is a special case. It is permanently open and the IOB entry contains the pointer to the DDB of the terminal on which the job was created. If the job detaches from the terminal, the IOB entry is not cleared, but the monitor removes the job number from the device data block, disassociating the terminal and the job. It also clears the device status bit in the DDB, indicating that the device is no longer a console for a job. (See Section 13.2.3 for a discussion of detached jobs.)

The I/O block is allocated from the general small buffer pool when the job is first created. It is deleted when the job is removed from the system.

### 3.1.5 Work Block (WRK)

The job's work block (WRK) is essentially a scratch area used to hold information that is normally contained only in the user program area. This allows the job to be swapped out during long monitor calls, especially during file processor and I/O stalls. Depending on the type of service the job is requesting, the work block contains a copy of either the FIRQB (for file processor operations) or the XRB (for I/O operations). (See Section 5.1.2 for a discussion of the FIRQB and XRB.)

The work block is allocated from the general small buffer pool when the job is first created. It is deleted when the job is removed from the system.

### 3.1.6 Job Status Tables (JBSTAT and JBWAIT)

Two tables, JBSTAT and JBWAIT, are used to determine whether or not a job is runnable. Each of these tables has one 1-word entry for every possible job on the system, including the null job. The entry for a particular job is accessed by indexing into the tables by a value equal to the job number, multiplied by two.

These two tables contain bit masks for device I/O and various other events. Whenever a user job requests a monitor service that cannot be satisfied immediately, the job's entry in JBSTAT is set to zero and the appropriate bit is set in the job's entry in JBWAIT. This indicates that the job is stalled pending completion of some function. When the function completes, the corresponding bit is set in JBSTAT to indicate that the request has been satisfied.

When the scheduler wants to determine whether a job is runnable, it performs a logical AND of the job's JBSTAT entry with its JBWAIT entry. If the result is nonzero, one or more bits match and the job is runnable. If the result is zero, either no bits match or the JBWAIT word is zero; the job is stalled and is not runnable.

It is possible for a job's JBWAIT entry to have more than one bit set, indicating that the job is stalled pending completion of multiple operations. Completion of any one operation will make the job runnable.

It is also possible for a job's JBWAIT entry to contain a value of zero. This condition is known as "hibernation" and occurs if a detached job attempts to do I/O to its console terminal. In this case, both the JBSTAT and JBWAIT entries are set to zero. When the job is later reattached to a terminal, JBWAIT is updated to some nonzero value and processing continues as normal.

When a Control-C is typed at the job's terminal, the monitor sets the appropriate bits for all interruptable devices in the job's JBSTAT entry. In this way, if the job was stalled pending an interruptable I/O operation, it will be made runnable immediately so that it can process the Control-C.

All bit assignments in the JBWAIT and JBSTAT tables have the general form "JS.xx", for device I/O (where xx is a physical device name), or "JSxxx", for other events. The permanently assigned bits are as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | JS.SY | This job status bit is for all direct memory access (DMA) devices. Since disk devices do synchronous I/O, many devices can share one bit without confusion. In addition, all other synchronous I/O devices use this bit. These devices are given unique symbolic names, but all these names are equated to this bit. |
| 1 | JS.KB | This job status bit is for terminal input. |

11 JSTEL This job status bit is for terminal output.

12 JSFIP This job status bit is for the file processor.

13 JSTIM This job status bit is for various timeout waits such as a wait in response to a .SLEEP monitor call, a wait for an incoming message, or a wait for terminal input from a multiterminal set.

14 JSBUF This job status bit is for stalls incurred when no buffer space is available for I/O buffers.

All other bits are assigned at system generation time for devices that can stall and destall a job while still processing I/O.

### 3.1.7 Fixed Memory Locations

Several fixed locations in low memory within the monitor are used to store information about the currently executing job. This provides a shortcut for accessing the most frequently used job control information. The information maintained is as follows:

*Symbol* *Content*

JOB This byte contains the job number (multiplied by two) of the currently executing job. If the value in this byte is zero, the null job is running. In that case, the data items below have special meanings, as noted.

NEXT If this byte is nonzero, it contains the job number (multiplied by two) of the job that was scheduled to be the current job but has not yet been swapped into memory. This job will begin execution immediately upon becoming resident. In this circumstance, the job shown in JOB is said to be "subscheduled" to make use of the available CPU time while waiting for the next job to swap in.

JOBDA This word contains the address of the job data block (JDB) for the currently executing job. If JOB contains a zero, this word will also contain a zero.

JOBF This word contains the address of the flag word (JDFLG) in the current job's JDB. If JOB contains a zero, this word will contain the address of a dummy value of zero.

IOSTS This word contains the address of the JDIOST and JDPOST bytes in the current job's JDB. If JOB contains a zero, this word will contain the address of a dummy value of zero.

JOBWRK This word contains the address of the work block (WRK) for the currently executing job. If JOB contains a zero, this word will also contain a zero.

JOBJD2 This word contains a pointer to the word at offset J2TICK in the secondary job data block (JDB2) for the currently executing job. If JOB contains a zero, this word contains the address of a scratch location to be used for J2TICK. Since the J2TICK data item is at offset 0 in the JDB2, this word also serves as a general pointer to the secondary job data block of the current job.

JOBRTS This word contains the address of the run-time system block (RTS) for the currently executing job. If the current job is using the disappearing RSX run-time system, this word will contain the address of the null RTS descriptor, NULRTS. If JOB contains a zero, this word will also contain a zero. (See Chapter 15 for a discussion of the run-time system descriptor block.)

CPUTIM This word contains a pointer to the word at offset J2CPU in the secondary job data block (JDB2) for the currently executing job.

JOBWDB This word contains the address of the first window descriptor block (WDB) used by the currently executing job. If the current job is not attached to any resident libraries, or if JOB contains a zero, this word will contain a zero. (See Chapter 16 for a complete discussion of resident libraries and window descriptor blocks.)

## 3.2 Job Scheduling

RSTS/E uses a priority-driven scheme for scheduling user jobs. When deciding what job to run next, the scheduler uses three criteria: runnability, priority, and residency. If several jobs are runnable, the resident job with the highest priority is run first.

### 3.2.1 Runnable Jobs

To determine if a job is runnable, the scheduler uses two tables — JBWAIT and JBSTAT. As discussed in Section 3.1.6, JBWAIT and JBSTAT are identical in format and contain bit masks for each existing job. A bit set in JBWAIT represents an event for which the job is waiting, and a bit set in JBSTAT represents an event that has completed.

When looking for a job to run, the scheduler scans these tables, doing a logical AND of each job's JBWAIT entry with its JBSTAT entry. If the result is nonzero, one or more of the bits set in JBWAIT is matched by the corresponding bit set in JBSTAT. That is, one or more of the events for which the job was waiting has completed, and the job is runnable. If the result is zero, no bits match, no events have completed, and the job is not runnable.

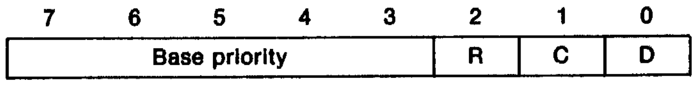
The scheduler does not run the first runnable job it finds, however. It continues to scan the tables, looking for the *highest priority* runnable job.

### 3.2.2 Job Priority

Job priority is a signed 8-bit value. It is divided into two parts. (See Figure 3-3.) The high-order five bits are generally static and, when combined with zeros in the low-order bits, represent the job's base priority — a signed multiple of eight in the range of -128 to +120(10). The base priority is set to a default of -8 when the job is created, but can be changed by the system manager (using the UTILTY program) or by the user job itself (by issuing the "change priority" system function call).

The low-order three bits of the job priority are dynamic and are used under certain circumstances to increase the job's base priority temporarily.

Bit 0 is controlled by the monitor and is set to raise the job's priority by one temporarily. This is done if the terminal operator enters a line delimiter while the job is waiting for keyboard input. Bit 0 may also be set (about 50% of the time, depending on certain system parameters) upon completion of a disk access.



R = Special run priority
C = Control-C typed
D = Line delimiter typed

**Figure 3-3:** Job Priority

Bit 1 is also controlled by the monitor and is set to raise the job's priority by two temporarily when the terminal operator types a Control-C.

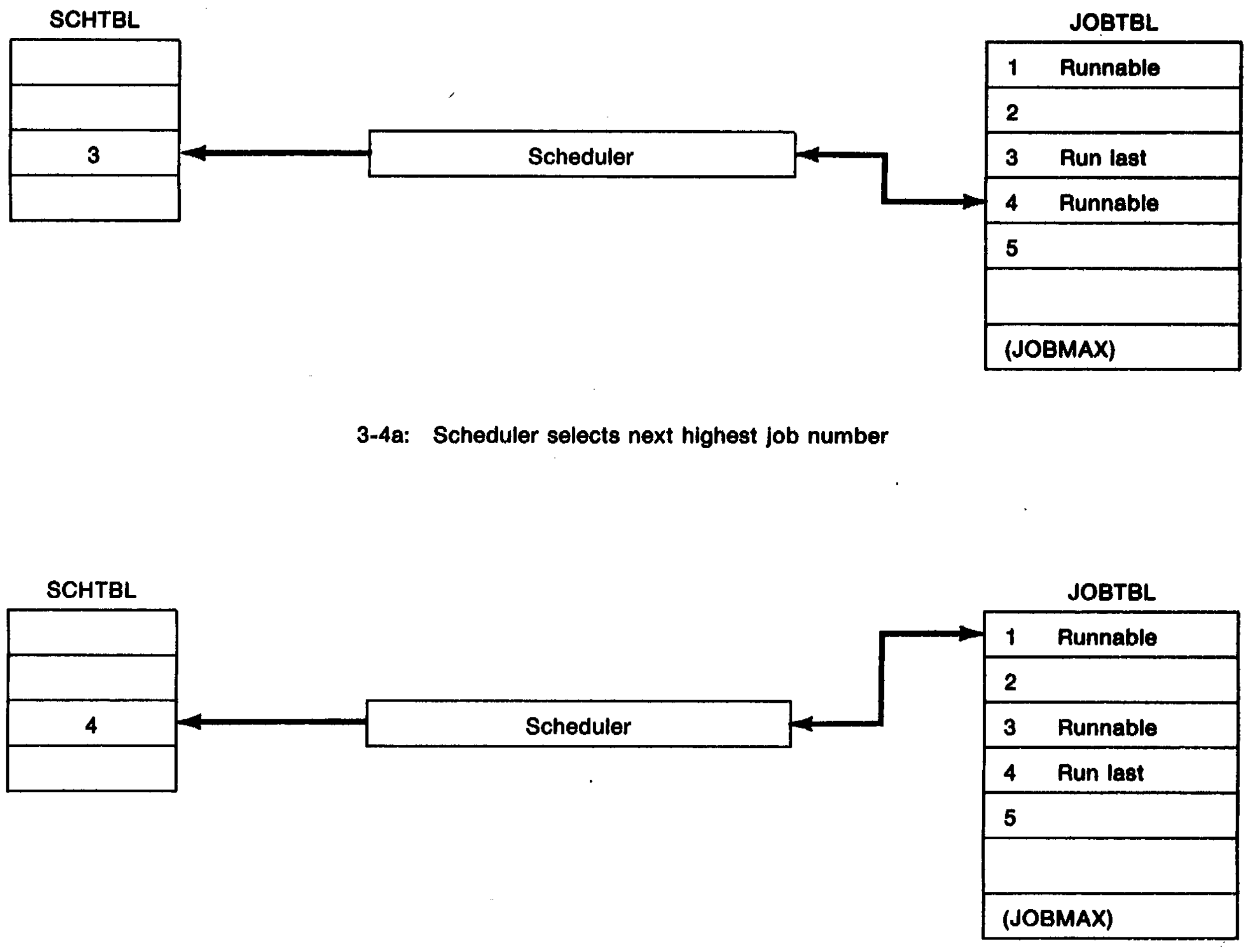
Bit 2 is controlled by the user program and is set by issuing the "set special run priority" system function call. Setting this bit increases the job's priority by four. This increase remains in effect until the job returns to the keyboard monitor or chains to another program.

### 3.2.3 Round-Robin Scheduling

If the scheduler finds several jobs that are runnable, the job with the highest priority is run first. However, if all the jobs found runnable have the same priority, they are run sequentially, by job number, until a higher priority job becomes runnable. This scheme is referred to as "round-robin" scheduling.

To prevent giving more runtime to one job than others of the same priority, the scheduler varies the starting point of the round-robin algorithm each time the algorithm is executed. This is done using information maintained in SCHTBL, a 256(10)-byte table consisting of one 1-byte entry for each priority level. Each entry contains the job number of the last job run at that priority.

When the scheduler must choose between two or more jobs of equal priority, it uses the job number in the appropriate SCHTBL entry as the starting point from which to select the job to run next. This is done by selecting the job with the next highest job number from the number in SCHTBL (see Figure 3-4a). If none of the currently runnable jobs has a job number higher than the number in SCHTBL, the scheduler selects the job with the *lowest* job number (see Figure 3-4b). The number of the job selected is then recorded in SCHTBL to be used as the starting point the next time the round-robin algorithm is executed.



3-4a: Scheduler selects next highest job number

3-4b: Scheduler selects lowest job number

**Figure 3-4:** Round-Robin Scheduling

### 3.2.4 Run Burst

Once the scheduler selects a job to run, the job is allowed to execute for a time defined by its "run burst." This is the amount of time, measured in clock ticks, that the job can run compute-bound before the scheduler will look for another runnable job. This is also called a "time slice." The run burst is doubled if any of the low-order three bits are set in the job's priority.

The run burst is set to a system default of one-tenth of a second when the job is first created, but can be changed by the system manager (with the UTILTY program) or by the user job itself (by issuing the "change priority" system function call).

### 3.2.5 Subscheduling

If the job chosen by the scheduler is not currently resident, the scheduler invokes the memory manager (through the level three queue mechanism) to allocate the necessary space and have the job brought into memory. The scheduler then rescans the job table to find the highest priority resident runnable job and permits it to run while the preferred job is being swapped in. This process is known as "subscheduling."

The number of the preferred job is saved in fixed memory location NEXT, and the subscheduled job is allowed to run until the preferred job becomes resident or for the period defined by its run burst, whichever is shorter. If the subscheduled job stops execution before the swap is complete (because it stalls for I/O, its run burst is exhausted, whatever), another resident runnable job is subscheduled.

If there is no resident runnable job, the scheduler runs the null task. This task runs until the next device interrupt and simply rotates the CPU panel lights.

# CHAPTER 4
# Memory Control [V8.0]

In RSTS/E, as in any operating system, memory is used for many different purposes. The monitor code takes a portion, along with run-time systems and resident libraries. Memory-resident storage requirements, such as directory and data caching, claim a large chunk. And, of course, there must be space for the user's applications programs. With all these demands on available memory — an obviously limited resource — memory control becomes a significant aspect of the total RSTS/E system.

Memory control is not the same as memory mapping. Memory mapping, as discussed in Chapter 2, involves determining where within the *virtual* address space certain pieces of the system are to exist. Memory control, on the other hand, involves determining where things will reside within *physical* memory. In a timesharing system such as RSTS/E, there is typically more demand for memory space than can be satisfied at any one time, and the actual contents of physical memory are constantly changing. Memory control, therefore, involves coordinating the contents of physical memory.

This chapter discusses the data structures used for memory control (Section 4.1) and the general allocation and scheduling of memory usage (Section 4.2).

## 4.1 Data Structures

RSTS/E does not use one central data structure to describe and control memory. Instead, specific portions of other structures are combined into two doubly-linked (forward and back) lists. The first list (MEMLST) describes the current memory users, while the second list (RESLST) keeps track of those items requiring memory.

### 4.1.1 Memory Control Sub-Blocks (MCB)

The basic unit of the memory control data structure is the memory control sub-block (MCB). The MCB is not actually a data structure in itself, but is rather a substructure contained within other structures that describe pieces of the system that use memory. The major system entities that require memory are the following:

- Jobs. An MCB is contained in the main job data block (JDB).
- Run-time systems. An MCB is contained in the run-time system descriptor block (RTS).
- Resident libraries. An MCB is contained in the library descriptor block (LIB).
- The monitor. MCBs are found in various places throughout the central monitor describing such things as XBUF, locked out memory, nonexistent memory, and so on.

The MCB within each of these structures contains an indication of the amount of memory required, as well as information concerning the location and status of each particular segment of memory.

Figure 4-1 shows the format of a memory control sub-block. Following the figure is a description of each entry.

| | | | |
|---|---|---|---|
| | Pointer to previous MCB | | M.PPRV |
| | Pointer to next MCB | | M.PNXT |
| | Total size of memory segment | | M.TSIZ |
| M.CTRL | Status/control | Actual size | M.SIZE |
| | Physical starting address | | M.PHYA |

**Figure 4-1:** Memory Control Sub-Block

*Offset* *Contents*

M.PPRV This word contains a pointer to the previous MCB (plus offset M.PNXT) in the memory control list (either MEMLST or RESLST). Note that this word contains a zero in the first entry in either memory list.

M.PNXT This word contains a pointer to the next MCB in the memory control list (either MEMLST or RESLST). Note that this word contains a zero in the last entry in either memory list.

NOTE

> If both M.PPRV and M.PNXT contain a zero in a particular MCB, the MCB is not in either memory control list — the memory segment described is not currently resident, nor is residency required.

M.TSIZ This word contains the size (in K-words) of the memory segment described by this MCB. This size includes the amount of memory used by the piece of the system to which the MCB applies, plus any available memory that follows it. Note that the entity may not be using all the memory specified by this value.

M.SIZE This byte contains the size (in K-words) of the actual memory in use by the piece of the system to which this MCB applies.

M.CTRL This byte contains status and control information about the memory segment described by this MCB (see Section 4.1.1.1).

M.PHYA This word contains the starting address of the memory segment described by this MCB, in memory management units (MMU units) — that is, the physical starting address divided by 64(10).

Note that memory control sub-blocks exist only for those pieces of the system where memory is actually in use; segments of available memory are not specifically described. However, such "holes" are implicitly described by the information contained within each MCB. By subtracting the size of the in-use portion of each memory segment (M.SIZE) from the segment's total size (M.TSIZ), the size (in K-words) of each corresponding hole (if any) can be calculated. Similarly, the starting address of the hole can be calculated by adding the actual in-use size of the explicitly described memory segment (M.SIZE) to the segment's physical starting address (M.PHYA).

#### 4.1.1.1 Memory Status Information (M.CTRL) -

Status and control information concerning the memory segment is contained in the byte at offset M.CTRL in the memory control sub-block. The bits in this byte are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | REQ | Residency is requested but the MCB is not linked into RESLST because the memory segment is currently being swapped out. |
| 1 | OUT | The memory segment should be (or is already) swapped out of memory, or is in the process of being swapped out. |
| 2 | IN | The memory segment should be swapped into memory, or is in the process of being swapped in. |
| 3-5 | | (unused) |
| 6 | SWP | A swap should be done. IN and OUT determine the direction. |
| 7 | LCK | The memory segment is not availabe for allocation or for other uses. Either the segment is locked in memory or is in the process of being swapped in or out. |

Some typical combinations of these bits are as follows:

| | |
|---|---|
| LCK!SWP!OUT | The memory segment is currently resident but should be swapped out. |
| LCK!OUT | The memory segment is in the process of being swapped out. |
| LCK!SWP!IN | The memory segment has been allocated memory and is ready to be swapped in. |
| LCK!IN | The memory segment is in the process of being swapped in. |
| LCK | The memory segment is locked in memory and is not available to be swapped out. |
| OUT | The memory segment is not currently resident and residency is not required. This is the state of a nonrunnable user job that is swapped out. In this case, the MCB is not linked into either MEMLST or RESLST. |

### 4.1.2 Resident Memory List (MEMLST)

The resident memory control list (MEMLST) describes all segments of physical memory. As memory is divided among several different pieces of the system, the memory control sub-blocks for each piece are linked into MEMLST, in ascending order. Thus, scanning the list from its root to its tail gives a complete description of all physical memory in the order in which it is allocated.

The resident memory control list is based at fixed location MEMLST. This address is the actual location of the first entry in the list; it is a list root, not a head pointer.

MEMLST always contains at least three entries: the root MCB, the MCB describing the system default run-time system (which is always resident), and the tail.MCB.

The root MCB is the monitor's memory control sub-block and describes the memory used by the monitor, plus any free memory following it. The status of the monitor memory segment is always locked (LCK) to indicate that the memory is not available for other uses, and the starting physical address (M.PHYA) is always zero.

The tail MCB is a dummy entry used to terminate the list and define the highest addressable memory location on the system. The sizes shown in the tail MCB (M.TSIZ and M.SIZE) are always set to one (although these values are not used by the monitor) and the status is always locked (LCK). The starting physical address (M.PHYA) contains the first nonexistent memory address on the system (divided by 64(10)), and thus corresponds to the total system memory size. Note that this does *not* include the address range used by the I/O page.

### 4.1.3 Desired Residency List (RESLST)

When a user job, run-time system or resident library is not currently resident in memory but needs to be, the scheduler links the entity's memory control sub-block into the desired residency list (RESLST). Usually the MCB is added to the end of the list, but under certain (higher priority) circumstances it is linked at the beginning. The memory manager then uses this list to keep track of and process requests for memory.

The first entry in the desired residency list is pointed to by fixed location RESLST. If the list is empty, the location RESLST contains a zero.

When a memory control sub-block is linked into RESLST, the byte at offset M.SIZE is set to the requested memory size (in K-words). The control and status byte (M.CTRL) is set to LCK!SWP!IN to show that a swap in from disk is required. The physical starting address can be either zero (to show that no specific memory address is necessary), or the actual desired memory address, in MMU units. (Note that run-time systems can request specific placement within memory but user jobs cannot. On the other hand, resident libraries *must* be loaded at a specific address. This has nothing to do with any feature of libraries themselves, but is a function of the memory manager algorithm.)

## 4.2 Memory Scheduling

When a memory control sub-block is added to RESLST, the memory manager is scheduled (through the level three queue mechanism) to fulfill the desired residency request. The memory manager allocates the space required and then schedules the swap manager. The swap manager handles the actual transfer from the disk to main memory.

### 4.2.1 Memory Manager

The memory manager processes entries in the desired residency control list (RESLST) on a first in, first out basis. Reviewing the first entry, the memory manager searches all of memory for the smallest space that will satisfy the request. This is known as "best fit." It is possible, to patch the system so that the search is terminated once *any* usable space is found. This is known as "first fit." With the system patched in this way, the memory manager runs faster, but memory tends to become fragmented.

If space is found, the memory manager allocates it to the requesting entity, storing the pertinent information in the memory control sub-block. LCK!SWP!IN are set in the status and control byte, and the MCB is then removed from RESLST and linked into MEMLST. At this point, the swap manager is scheduled (through the level three queue mechanism) to handle the actual transfer from disk.

If space is not available, the memory manager looks for one or more entities that can be removed from memory to make the necessary room. In selecting what pieces to swap out, the memory manager looks first for a "swappable" job. It then looks for a job without a resident run-time system. Finally, it looks for a run-time system that was not loaded at a specific address. Control bits LCK!SWP!OUT are set in the MCB of the entities selected and the swap manager is scheduled.

Note that job and memory scheduling are independent on RSTS/E. The memory manager ignores job priority and run status when selecting a job to swap out and may even swap out a job that is currently runnable. The only job that is absolutely exempt from swapping is the job whose number is in location JOB (see Section 3.1.7).

The memory control sub-block of the entity requesting memory is left in RESLST until any necessary swaps out are completed. At that time, when space is available, the memory manager proceeds as above, updating the MCB and linking it into MEMLST. The swap manager is then scheduled to bring the required entity into memory.

#### 4.2.1.1 Residency Quantum -

A job's "swappability" is based on what is known as its residency quantum. This value is assigned when the job first enters memory and determines how long the job is to remain resident. The residency quantum is decreased by a value that depends on the job's priority (higher priority jobs remain resident longer) for every tenth of a second that the job is resident and running. Only the amount of time a job runs is counted against its residency quantum. When the residency quantum reaches zero, the job is considered swappable.

Because there is system overhead associated with moving things in and out of memory, too much swapping can result in a condition known as thrashing — more time is spent swapping jobs than running them. By guaranteeing a minimum amount of time that each job is allowed to stay in core without being swapped out, this condition can be avoided.

A job's residency quantum depends on its size and is essentially a measure of the amount of work needed to swap the job in from the disk. The formula used is as follows:

(job size in K-words) * ..QMUL + ..QADD

The constant ..QMUL is called the residency quantum multiplier and can be considered a measure of the time needed to transfer a one K-word block of data from the disk. The constant ..QADD is the residency quantum additive factor and can be considered a measure of the average swapping disk seek time.

The residency quantum is immediately set to zero if the job stalls for any non-DMA I/O. It is also decreased by certain "punishment factors" if the job requests disk I/O or file processing.

- If the job enters a disk I/O wait state, its residency quantum is decreased by the number of blocks to read or write multiplied by constant ..BQNT. A value of zero will prevent punishment for I/O.

- If the job queues a request to the file processor, its residency quantum is decreased by the depth in the FIP queue multiplied by the constant ..FQNT. A value of zero will prevent punishment for file processor requests.

### 4.2.1.2 Locking -

By issuing the "lock job in memory" system call, a job can lock itself in memory, making it unswappable. Note, however, that this system call merely sets bit JFLOCK in the primary job status flags (JDFLG) of the job data block. It does not set the LCK bit in the job's MCB; the job *can* still be swapped out under certain circumstances.

- If the job increases in size, but cannot grow in place, it will be swapped out and immediately swapped back in at its new size.
- The job will be swapped out if a portion of its job image is required to load a run-time system for another (runnable) job.
- The job will become eligible to be swapped out if its run-time system becomes nonresident.
- The job will become eligible to be swapped out if the program exits, chains or aborts on an error.

Locking a job in memory is discouraged. Since a job cannot specify where in memory it is to be located, locking may fragment memory.

## 4.2.2 Swap Manager

The swap manager processes entries in the resident memory control list (MEMLST) that need to be swapped either in or out. Scanning MEMLST for entries that have the SWP bit set in the status and control byte, it allocates and sets up a disk request queue entry block (DSQ) for any memory segment that must be swapped in or out. The DSQ is then passed to the disk I/O subsystem, initiating the actual transfer. (See Chapter 10 for a discussion of the DSQ and the disk I/O subsystem.)

If the memory segment to be swapped is larger than 31 K-words, the swap manager uses two DSQs — one to maintain the swap context and the other to request the transfer of each 31 K-word segment.

NOTE

> Since run-time systems contain pure code, there is no need to save the core image on disk. Thus, when "swapping out" a run-time system, no actual transfer takes place. In fact, the memory manager simply unlinks the run-time system's MCB from MEMLST, thereby freeing up the space, and no action is required by the swap manager.

When the swap is complete, the swap manager updates the status byte of the MCB. If the memory segment was swapped out, the entry is removed from MEMLST.

When the swap manager has completed its processing, the memory manager is scheduled again to continue whatever processing it must do.

### 4.2.2.1 Swap Files -

When a user job is to be swapped out, the swap manager selects one of the system "swap files" in which to store the job. RSTS/E allows specification of up to four distinct swap files, numbered 0 to 3. Swap file 2 is permanently assigned to file SWAP.SYS on the system disk. The other files can be assigned on any logically mounted disk, or they can be allocated to a logically unmounted disk, using the entire disk volume as a single file. They can also be added or removed during timesharing.

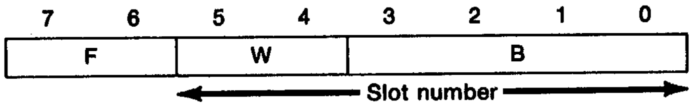
Each swap file is divided into "slots." The size of each slot is equal to SWPMAX, a system startup parameter equal to the maximum size to which any job can expand in memory. Thus, each slot is capable of holding one user job. Each swap file can have up to 63(10) slots, so each file has the potential for handling the system maximum of 63 jobs. Since all jobs must be swappable, the total number of jobs that all swap files can accommodate limits the number of jobs that can log in. As swapping space is added or removed during timesharing, the monitor can modify the actual number of permissible jobs accordingly. If swapping space is increased, the monitor modifies the number of jobs only if it is requested to do so. However, it will always modify the number of jobs if swapping space is decreased.

Swap file 2, SWAP.SYS, must have at least two slots since system startup requires two jobs.

Each swap file has an associated bit map — four words of sixteen bits each, corresponding to the 63 possible slots (plus one unused bit). Allocated (in use) slots are represented by zeroes; ones denote free slots. Bit 0 in each map represents a dummy slot and is always zero.

These four bit maps are stored contiguously. The swap manager selects file space for a job to be swapped by scanning them sequentially, looking for a free slot. The order of the scan depends on the current job status. If the job is runnable or stalled for I/O (including waiting for terminal input), the scan is done from low to high (file 0/slot 0 to file 3/slot 63). If it is hibernating or in a sleep-wait state (including message-receive wait), the scan is done from high to low (file 3/slot 63 to file 0/slot 0). Normally, the lower numbered files are allocated to high-speed disks, while the higher numbered files are allocated to slower disks.

When a job is swapped out, the swap manager marks its swapped out location at offset JDSWAP in the job data block. Figure 4-2 shows the format of this byte. Bits 6 and 7 indicate which of the four swap files the job is swapped to, and bits 0 to 5 indicate the slot within the file.



F = swap file number
W = word within swap file bit map
B = bit within bit map word

**Figure 4-2:** Job Swap Parameter

# CHAPTER 5
# Monitor Services [V8.0]

In addition to job and memory control, the RSTS/E monitor is responsible for providing several other services within the system. These services include interfacing to the peripheral devices, dynamic allocation and deallocation of general buffer space, interjob communications, and file management. These additional services are used by the monitor itself as well as by user jobs.

NOTE

> The use of monitor buffer space, however, is available only to monitor components. User jobs cannot access this feature.

This chapter discusses various aspects of the service functions provided by the RSTS/E monitor. Section 5.1 covers the general format and processing of the user/monitor interface, Section 5.2 discusses the interjob communication facility, and Section 5.3 covers the structure and usage of the monitor's general buffer space. Device control is discussed separately in Chapter 6. File management is covered in Part III, "Disk I/O and the File Processor".

## 5.1 User/Monitor Interface

With the exception of the use of monitor buffer space, the additional service functions provided by the monitor are available to internal monitor components as well as to user jobs. Modules within the monitor access these functions either through the level three queue mechanism or by direct subroutine call (see Section 2.3). User jobs, however, cannot use either of these methods. This section, therefore, discusses the interface used by user jobs to request monitor services.

### 5.1.1 Privilege

Not all jobs have equal access to the services provided by RSTS/E. In this regard, there are two types of job: privileged and nonprivileged. Privileged jobs have certain capabilities that nonprivileged jobs do not have. In particular, a privileged job has the following special capabilities:

- A privileged job has unlimited read and write access to all files on the system, regardless of their protection codes. It can create and delete files under any account, and can access files on locked disks.
- A privileged job has the ability to designate user programs as privileged.
- A privileged job can use certain system functions not available to nonprivileged jobs — various system programs, privileged system function SYS calls, and the PEEK function.

A job has privilege under one of the following conditions:

- When it is logged out
- When it is logged in under a privileged account

- When it is running a privileged program

A logged out job has privilege because the system must perform certain privileged operations to log the job onto the system. The privilege remains in effect as long as the job remains logged out.

A job running under a privileged account has permanent privilege until it logs off the system or changes to a nonprivileged account.

A job running a privileged program has temporary privilege. Such a program has most of the capabilities of a job with permanent privilege except that it cannot designate another program as privileged. Temporary privilege remains in effect until the program exits, aborts or chains to another program. A program can also drop its temporary privilege by issuing the "drop temporary privilege" system call.

### 5.1.2 User Low Core Area

RSTS/E uses a standardized method for transferring data and status information between a user job and the monitor. This method requires that the lowest addresses of the job's virtual address space have a specific format.

Within the user job image area, virtual addresses 0 to 777(8) have special meaning attached to them and are used for communication with the monitor. The format of this low core area is as follows:

| *Addr* | *Size** | *Symbol* | *Contents* |
|---|---|---|---|
| 0 | 60 | | This area is controlled solely by the job or the run-time system. |
| 60 | 30 | CONTXT | This area is used by the monitor for storing job context information in order to make the job swappable. Any data read from this area must be considered random and any data written into this area by the job may be destroyed. |
| 110 | 60 | FPPTXT | This area is used by the monitor for storing the floating point hardware context information (if any) in order to make the job swappable. This area is not used by the monitor unless specified in the user's keyword (KEY). |
| 170 | 210 | | This area is the default stack area for the job. When the job is started, its stack pointer (SP, hardware register 6) is set to 400(8). The job can use another area for its stack but must be aware that certain catastrophic errors (such as disk swap failures) cause the stack pointer to be reset to this area. Thus, this area should not be used as storage for general data. |
| 400 | | USRSP | This is the address to which the user job's stack pointer is initially set. |
| 400 | 2 | KEY | This is the job's keyword (see Section 5.1.2.1). |

* in octal bytes

| | | | |
|---|---|---|---|
| 402 | | IOSTS | This is the address to which the monitor returns error codes when a monitor service request fails for any reason. If the request completes without an error, this location is returned with a value of zero. While error codes are all unsigned nonzero byte values (7 bits), this location is returned as a word value for the convenience of the user job.<br>NOTE<br>The mnemonic symbol IOSTS is not defined in the prefix file COMMON.MAC. However, it is used, by convention, to refer to this location. By equating this symbol to the symbol FIRQB, which is defined in COMMON.MAC, the correct equivalence is made. |
| 402 | 40 | FIRQB | This area is the job's file request queue block (FIRQB). See Section 5.1.2.2. |
| 442 | 16 | XRB | This area is the job's transfer request block (XRB). See Section 5.1.2.3. |
| 460 | 200 | CORCMN | This area is used as a common data exchange area when it is necessary to exchange lengthy data between the monitor and the job or between programs running under the same job number. |
| 660 | 54 | | This area is controlled solely by the user job or the run-time system. |
| 734 | 2 | USRPPN | This word contains the job's assigned project-programmer number (PPN). The high byte contains the project (or group) number and the low byte contains the programmer (or user) number. If the job has no assigned logical PPN, this word will contain a value of zero. |
| 736 | 2 | USRPRT | This word contains the job's default output file protection code. The value of the code itself (ranging from 0 to 255(10)) is in the high byte. The low byte contains a nonzero flag to indicate that the value is an explicit protection code (particularly for a value of zero). If the low byte is zero, the job has no explicit default protection code and the system default will be used whenever necessary. |
| 740 | 40 | USRLOG | This area contains the job's private logical device name table. Logical device names are discussed in detail in Section 6.2.2. |
| 1000 | | NSTORG | This is the start of the nonspecific user job image. |

#### 5.1.2.1 Job Keyword (KEY) -

The keyword holds certain status information for the job. Of the sixteen bits in the keyword, seven are controlled by the monitor. The other nine bits can be used by the job.

The keyword is "refreshed" by the monitor upon certain entry points into the run-time system. During this refresh operation, the monitor clears the entire word and then updates the monitor-controlled bits from information stored in the user's job data block. Six of the seven monitor bits are maintained in the primary flags word (JDFLG), and the remaining bit is incorporated into the job priority byte (JDPRI). See Sections 3.1.2 and 3.2.2.

The monitor-controlled bits are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-7 | | (unused) |
| 8 | JFSPRI | This bit indicates that the job is running with the special run priority (see Section 3.2.2). |
| 9 | JFFPP | This bit indicates that the contents of the floating point hardware unit (if any) should be part of the context of this job. |
| 10 | JFPRIV | This bit indicates that the job is logged in under a privileged account and is therefore permanently privileged. |
| 11 | JFSYS | This bit indicates that the job is currently running with temporary privileges. This bit is never set if the JFPRIV bit is set. |
| 12 | JFNOPR | This bit indicates that the job is not yet logged in. |
| 13 | JFBIG | This bit indicates that the job can exceed its private memory size maximum. |
| 14 | JFLOCK | This bit indicates that the job should not be swapped out. Note, however, that there are certain conditions under which the job can still be swapped (see Section 4.2.1). |
| 15 | | (unused) |

### 5.1.2.2 File Request Queue Block (FIRQB) -

The file request queue block (FIRQB) is the main communications area between the monitor and the job during file or device operations that do not involve reading or writing.

The FIRQB has a fixed format for all monitor requests. That is, with a few exceptions (noted below), each word or byte in the block is reserved for a particular item (device name, file size, and so on) even though that item may not be pertinent to all monitor requests. The general procedure for setting up the FIRQB is to first zero the entire block. Then, only those parameters that apply to the specific call are loaded.

Note that the first word of the FIRQB is not used to pass data to the monitor but is used to return error codes to the user. This is the status word, IOSTS, described in Section 5.1.2.

The byte at offset FQJOB is a "read-only" location; the monitor ignores any data provided in this word. Instead, upon initiation of each request, the monitor loads this word with the user's job number (multiplied by two). This is essentially a convenience to the user job for obtaining its job number.

Figure 5-1 shows the format of the file request queue block. Each entry is discussed below.

| *Offset* | *Contents* |
|---|---|
| FIRQB | This word is used to return error codes to the user. By convention, the symbol FIRQB is equated to symbol IOSTS, the status word, as described in Section 5.1.2. |

FQJOB This byte contains the job number (multiplied by two). It is refreshed by the monitor on every monitor call.

FQFUN This byte contains the subfunction code for calls to the file processor.

| | | | |
|---|---|---|---|
| | Returned status | | FIRQB |
| FQFUN | Subfunction code | Job number * 2 | FQJOB |
| FQSIZM | MSB of file size | Channel number * 2 | FQFIL/FQERNO |
| | Project number | Programmer number | FQPPN |
| | File name | | FQNAM1 |
| | File type | | FQEXT |
| | File size | | FQSIZ |
| | Buffer length | | FQBUFL/FQNAM2 |
| | Open mode | | FQMODE |
| | Status flags | | FQFLAG |
| FQPROT | Protection code | "Code is real" | FQPFLG |
| | Device name | | FQDEV |
| | "Number is real" | Device unit number | FQDEVN |
| | Cluster size | | FQCLUS |
| | Number of entries in directory lookup | | FQNENT |

**Figure 5-1:** File Request Queue Block

FQFIL This byte contains the channel number (multiplied by two) for file operations that require an open channel. RSTS/E permits up to 16(10) channels to be open at once, numbered 0 to 15.

FQERNO When the monitor request is to return the text of an error message, this byte contains the corresponding error code. (Note that this offset is equal to FQFIL.)

FQSIZM This byte contains the high-order eight bits of the file size (see FQSIZ below).

FQPPN This word contains the project-programmer number (PPN) for file operations. The low byte contains the programmer (or user) number and the high byte contains the project (or group) number.

FQNAM1 This two word area contains a file name, in RAD50 format.

FQEXT This word contains a file type, in RAD50 format.

FQSIZ During file creation operations, this word contains the low-order sixteen bits of the desired file size (in blocks). When opening an existing file, the monitor loads this word with the actual size of the file (in blocks). The high-order (most significant) bits of this value are found at offset FQSIZM.

FQBUFL When opening a file or device, the monitor loads this word with the device default buffer size (in bytes).

FQNAM2 This three word area contains a file name and file type (in RAD50 format) for file renaming operations. (Note that this offset is equal to FQBUFL.)

FQMODE This word contains the device dependent mode settings for file and device open operations.

FQFLAG When opening a file or device, the monitor loads this word with a set of status flags and a device handler index.

FQPFLG This byte is used as a nonzero flag to indicate that the value in the following byte (particularly a value of zero) is an explicit protection code. If the value in this byte is zero, no protection code has been specified and the job default protection code (found in location USRPRT of the job's low core area) will be used. (See Section 5.1.2.)

FQPROT If the value in byte FQPFLG is nonzero, this byte contains the desired file protection code for file creation operations.

FQDEV This word contains a device name (as two ASCII characters).

FQDEVN This word contains a device number in the low byte. The high byte contains a nonzero flag to indicate an explicit unit number (particularly a unit number of zero). If the high byte is zero, no unit number has been specified.

FQCLUS This word contains the desired cluster size for file create operations.

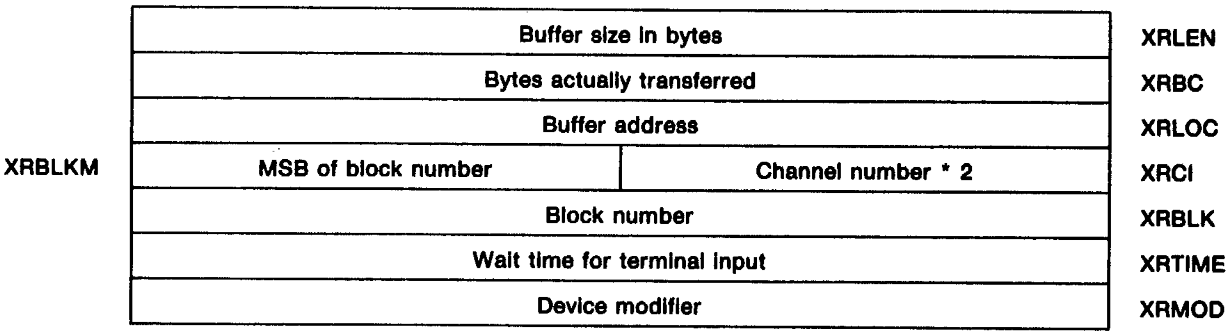
FQNENT This word contains the user-supplied parameter for the "run new program" system call.

### 5.1.2.3 Transfer Request Block (XRB) -

The transfer request block (XRB) is the main communication area between the monitor and the user job during monitor handling of file or device I/O. It is also the area used during all simple informational monitor calls.

As with the FIRQB, the XRB has a fixed format, with each byte or word being reserved for a particular data item.

Figure 5-2 shows the format of the transfer request block. Following the figure is a description of each entry.



| | | | |
|---|---|---|---|
| | Buffer size in bytes | | XRLEN |
| | Bytes actually transferred | | XRBC |
| | Buffer address | | XRLOC |
| XRBLKM | MSB of block number | Channel number * 2 | XRCI |
| | Block number | | XRBLK |
| | Wait time for terminal input | | XRTIME |
| | Device modifier | | XRMOD |

**Figure 5-2:** Transfer Request Block

| *Offset* | *Contents* |
|---|---|
| XRLEN | This word contains the size (in bytes) of the buffer being used for the transfer. The value must not be zero. |
| XRBC | For a read operation, the monitor loads this word with the number of bytes actually read. For a write operation, this word contains the number of bytes to write. |
| XRLOC | This word contains the address of the user buffer being used the transfer. |
| XRCI | This byte contains the channel number (multiplied by two) on which the I/O operation is to take place. |
| XRBLKM | This byte contains the high-order eight bits of the block number (see XRBLK below). |
| XRBLK | For random-access I/O operations, this word contains the low-order sixteen bits of the virtual block number (VBN) to be read or written. If the 3-byte VBN is zero, the next sequential block is accessed. This value is ignored for non-random-access devices. The high-order (most significant) bits of this value are found at offset XRBLKM. |
| XRTIME | This word contains the time limit that the user program wishes to wait for terminal input. It is ignored for all other operations. |
| XRMOD | This word contains a device-dependent modifier. |

### 5.1.3 User Request Processing

All monitor requests from a user job take the form of emulator traps (EMTs) — a form of software-generated interrupt to the central processor. When a user job issues an EMT, the central processor is forced into kernel mode and control transfers to the monitor EMT handler.

This routine maps the lowest four K-words of the user job image into kernel virtual address space, using kernel APR 6. It copies the appropriate portion of the user's low core area (either the FIRQB or the XRB) into the job's work block (WRK) in the monitor's virtual address space. Kernel APR 6 is then made available for other uses.

The user job is made nonrunnable and may be made eligible to be swapped out while the request is being processed.

The EMT handler extracts the numeric code provided with the EMT, thus identifying the function required. This code is used as an index into the EMT address table, EMTTBL, and the corresponding entry in that table is used to dispatch to the appropriate service routine.

Upon completion of the required function, any information that must be passed back to the user job is handled through a process known as "posting." The lowest four K-words of the user job image are again mapped into the monitor's address space, using kernel APR 6. The value in byte JDPOST of the user's job data block is then used to determine what data should be moved from the job's work block to its FIRQB or XRB. If the value in JDPOST is positive, it is used as an index into a table of bit masks used for posting to the FIRQB. If it is negative, the low-order seven bits are used as a bit mask for posting to the XRB. In either case, each bit corresponds to a word in the job's work block that is to be copied (if the bit is set) or not copied (if it is clear) into the job image low core area.

User request EMT 0, a call to the file processor (FIP), is a special case. FIP requests are made for various forms of file management, as well as certain general system functions. The basic EMT from the user is handled in the same fashion as for all other (general) EMTs. After dispatching through EMTTBL, however, control transfers to an intermediate service routine, FIPEMT. This routine enters the job's work block in the file processor's queue (FIQUE). When the file processor is ready to honor the user's request, it removes the work block from the queue, extracts the function code at FIRQB+FQFUN, and uses the code as an index into table FIPTBL. The corresponding entry in that table is then used to dispatch to the final service module. (See Chapter 9 for a full discussion of the file processor.)

When the user job is running under control of the RT-11 or RSX run-time systems, certain "monitor" calls are processed by the run-time system itself. In these instances, the monitor EMT trap handler reroutes the call back to the run-time system. Thus, the run-time system acts as a run-time translator for certain service requests, asking the RSTS/E monitor to perform basic subfunctions. (See Chapter 15 for more details on run-time system emulators.)

## 5.2 Interjob Communication

The RSTS/E interjob communication facility allows a user job to exchange messages with other jobs running either locally or (with the DECnet option) on a distant (remote) system.

### 5.2.1 Receiver ID Block (RIB)

To receive messages, a job must first declare itself a message receiver by issuing the "declare message receiver" monitor call. The monitor allocates a small buffer from the general pool and builds a receiver identification block (RIB) for the job. The RIB is pointed to by the word at offset J2MPTR in the job's secondary job data block (JDB2).

All receiver ID blocks on the system are linked together in a list of message receivers. This list is based at fixed location SNDLST, with the word at that location containing the address of the first receiver ID block in the list. Each RIB, in turn, points to the next block in the list. The system utility ERRCPY is always the first message receiver in the list. Therefore, SNDLST never contains a zero.

A single job can declare itself as a multiple receiver by issuing more than one "declare message receiver" monitor call. For each call issued, the monitor sets up a separate receiver ID block, each with a separate queue of pending messages and with its own set of characteristics.

Figure 5-3 shows the format of the receiver ID block. Each entry is discussed below.

| *Offset* | *Contents* |
|---|---|
| S.LINK | This word contains the address of the next receiver ID block in the receiver list. If this is the last RIB in the list, this word will contain a zero. |

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 is filled with trailing spaces. This is the main identification of the receiver and must be unique within the system.

| | | | |
|---|---|---|---|
| | Link to next receiver ID block | | S.LINK |
| | Receiver ID<br>(6 bytes of ASCII) | | S.RCID |
| S.OBJT | Object type | Job number * 2 | S.JBNO |
| S.SRBN | RIB number | Access flags | S.ACCS |
| | Buffer maximum | | S.BMAX |
| S.MCNT | Pending message count | Message maximum | S.MMAX |
| | Pointer to first pending message | | S.MLST |
| | Pointer to last pending message | | S.MEND |
| S.LCNT | Logical link count | Maximum link count | S.LMAX |
| | Pointer to first logical link block | | S.LLST |
| | Pointer to last logical link block | | |
| | | Outgoing link maximum | S.OMAX |
| | Reserved for network use | | |

**Figure 5-3:** Receiver ID Block

S.JBNO This byte contains the job number (multiplied by two) of the job to which this RIB belongs.

S.OBJT This byte contains the object type for use in DECnet network messages.

S.ACCS This byte contains the receiver's access control bits, describing the characteristics of the associated receiver and the type of senders that are permitted to send messages to the receiver. The majority of the restrictions indicated here are specified by the job when it declares itself a receiver. The bits are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | SA.LCL | This bit indicates that local senders can send messages to this receiver. The type of local sender permitted can be further modified by the SA.PRV bit. |
| 1 | SA.PRV | This bit indicates that local senders must be privileged to send messages to this receiver. This bit cannot be set if the SA.LCL bit is clear. If the SA.LCL bit is set and this bit is clear, then any local sender can send messages to this receiver. |

| | | |
|---|---|---|
| 2 | SA.NET | This bit indicates that network senders can send messages to this receiver. (A network sender is any job, either local or remote, that uses the DECnet facility to establish and use logical links.) |
| 3 | SA.1SH | This bit indicates that only one logical link can be established during the time this job is a message receiver. This is different from a link maximum of one, as specified at offset S.LMAX of the RIB. A link maximum of one means the receiver can have only one link at a time, whereas the SA.1SH bit means the receiver can have one and only one logical link. If this bit is set when a logical link is established, the link maximum is automatically set to zero, no matter what its original value. |
| 4 | SA.NCS | This bit is used to modify the function of the "conditional sleep" monitor call. Normally, any unreceived messages pending for the receiver will block execution of a conditional sleep. If this bit is set, however, the monitor will not check the job's message queue when determining whether or not it should suspend execution of the job. Several other conditions (such as a delimiter received on an open terminal) can still block the sleep, but a pending message will not. |
| 5 | | (unused) |
| 6 | SA.EVT | This bit indicates that the receiver ID block belongs to the DECnet event logger. |
| 7 | SA.XOF | This bit indicates that messages from local senders are temporarily not permitted. |

S.SRBN This byte contains the RIB number for this receiver if the job has declared itself as more than one receiver. The job assigns this number when it issues the call which establishes this receiver ID block.

S.BMAX This word contains the maximum amount of buffer space (in bytes) that can be used to hold pending message information. When this limit is exceeded, messages are no longer queued from local senders. Network messages are always queued, regardless of this declared maximum, but any buffer space used is counted against the maximum. This limit is specified by the job when it declares itself a receiver.

S.MMAX This byte contains the maximum number of messages that can be pending for this receiver at any one time. This value can range from 1 to 255(10) and is specified by the job when it declares itself a receiver. The actual number of pending messages can be restricted even further by the limit imposed on buffer usage.

S.MCNT This byte contains a count of the number of messages (if any) currently pending for this receiver.

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 currently pending, this word will contain a zero.

S.MEND This word is a pointer to the pending message block (PMB) for the last message pending for this receiver. If no messages are currently pending, this word points to the word at offset S.MLST.

S.LMAX This byte contains the maximum number of incoming requests for logical links that will be queued for this receiver.* This limit is specified by the job when it declares itself a receiver.

S.LCNT This byte contains a count of the current number of logical links for this receiver.

S.LLST This word points to the first logical link descriptor block for this receiver. If no such logical links exist, this word will contain a zero.

S.LLST+2 This word points to the last logical link descriptor block for this receiver. If no such logical links exist, this word will point to the word at offset S.LLST.

S.OMAX This byte contains the maximum number of outgoing logical links permitted for this receiver. This limit is specified by the job when it declares itself a receiver.

### 5.2.2 Pending Message Block (PMB)

Each message waiting to be processed by its intended receiver has an associated 16(10)-word pending message block (PMB). All PMBs for a particular receiver are kept in a linked list pointed to by the word at offset S.MLST in the receiver ID block. The list is maintained and processed on a "first in, first out" basis.

There are two sections in a pending message block. A 6-word header region describes the message type and any additional control information. The content and meaning of the remaining ten words depend on the type of the message. Local senders can use this area for whatever purpose the sender and receiver decide. For network messages, this area is used for logical link connection information.

Messages can be either large or small. A small message is completely contained within the pending message block. A large message, on the other hand, can be up to 512(10) bytes long. In this case, the header region contains the address of an additional buffer (or block of buffers) containing the message data. The actual message data is preceded by a 2-word header, as follows:

- One word containing the buffer size (in bytes)
- One word containing the offset from the beginning of the buffer (including the 2-word header) to the first byte of actual data

When the job issues a "receive message" system call, it must specify a buffer to receive the message data. This buffer can be smaller than the message sent. When this happens, the monitor transfers as much of the message as it can, leaving the remainder in the large message buffer and updating the second word of the buffer header to point to the first

* Logical links are a feature of the DECnet message facility. Refer to any of the DECnet/E network programming manuals for details.

untransferred byte of data. The number of bytes remaining is also updated in word P$BREM of the PMB. When the job issues its next "receive message" call, data is transferred from the point where the previous transfer left off. When all of the large message data has been moved to the user buffer, the pending message block and the large message buffer are returned to their respective buffer pools. The user job can alternately request that any excess data that does not fit in the specified user buffer be discarded. In this case, the PMB and large message buffer are deallocated immediately.

Figure 5-4 shows the format of the pending message block. Each entry is discussed below.

| | | | |
|---|---|---|---|
| | Link to next pending message block | | P$LINK |
| | "Contorted" pointer to message buffer | | P$BUFA |
| P$SNDR | Sender job number * 2 | Message type | P$TYPE |
| | Sender PPN | | P$SPPN |
| | (Unused) | Sender's keyboard number | P$SKBN |
| | Number of bytes remaining | | P$BREM |
| | Small message data | | P$PARM |

**Figure 5-4:** Pending Message Block

| *Offset* | *Contents* |
|---|---|
| P$LINK | This word contains the address of the next PMB in the list of pending messages for this receiver. If this is the last PMB in the list, this word will contain a zero. |
| P$BUFA | This word contains the "contorted" address of the buffer containing the large message data, if any (see Section 5.3.3.2). If this word is zero, no large message data was sent. |
| P$TYPE | This byte specifies the type of message described by the PMB. The type definitions are as follows: |

-1 = local sender message
-2 = connect initiate message (DECnet)
-3 = connect confirm message (DECnet)
-4 = connect reject message (DECnet)
-5 = network sender message (DECnet)
-6 = interrupt message (DECnet)
-7 = link service message (DECnet)
-8 = disconnect message (DECnet)

-9 = link abort message (DECnet)
-10 = event logger message (DECnet)

| | |
|---|---|
| P$SNDR | This byte contains the job number (multiplied by two) of the local message sender. |
| P$SPPN | This word contains the project-programmer number (PPN) of the local message sender. The high byte contains the project (or group) number and the low byte contains the programmer (or user) number. |
| P$SKBN | This byte contains the keyboard number of the sending job. If the sender is a network sender or a local detached job, this byte contains a value of -1. |
| P$BREM | This word specifies the number of bytes remaining in the large message buffer that have not yet been transferred to the user. |
| P$PARM | These ten words contain the user specified parameters for small messages and the network parameters for DECnet messages. |

## 5.3 Monitor Buffers

Because the requirement for certain in-core monitor data structures changes during the course of timesharing, many of the structures used by various monitor components are not permanently configured. They are allocated instead as needed from free space maintained for that purpose. Structures such as job data blocks, file control blocks, queue entries, I/O data buffers, disk caching buffers, and so on are all stored in dynamically allocated monitor controlled buffers.

NOTE

> The use of the buffer features described in this section are available only to monitor components, not to user jobs.

### 5.3.1 Buffer Types

There are basically two types of monitor controlled buffers under RSTS/E: small buffers, consisting of 32(10) bytes of memory, and large buffers, consisting of 64(10) bytes.

#### 5.3.1.1 Small Buffers -

Small buffers are used for data structures such as job data blocks, queue entries, resident library and run-time system control blocks, and asynchronous I/O data storage.

Since many of these structures must be accessible to several monitor components, the majority of small buffers are allocated from a general pool in the monitor's permanently mapped root. Originally, all small buffers came from this general pool. However, increased demand for buffers due to additional monitor features and more powerful processors, coupled with the limited size of the root (and, therefore, of the small buffer pool itself), eventually led to what was known as the "small buffer problem" — a shortage of buffers resulting in decreased system response and throughput.

To relieve this shortage, version 7.1 implemented two new optional features: a file processor buffer pool and the use of instruction and data (I&D) space APRs.

The FIP pool is basically an extension of the general small buffer pool. It is used primarily by the file processor and is mapped with the FIP phase using kernel APR 6. It is used to supply buffers for Concise Command Language (CCL) control blocks, file control blocks (FCBs) and window control blocks (WCBs) for open files. This lessens the demand for buffers from the general pool.

Use of the FIP pool is enabled and disabled through INIT and is available on all RSTS/E systems. The use of I&D space, however, is only available on those PDP-11 processor models that support the special memory management hardware feature known as 'data space' (see Section 2.2.1).

When data space is used by RSTS/E, the monitor makes double use of a portion of its virtual address space. Many of the common subroutines in the permanently mapped root are mapped in instruction space but do not need mapping in data space. Thus, the corresponding portion of the data space is used to map extra small buffers.

#### 5.3.1.2 Large Buffers -

Large buffers are used for caching disk data, buffering output to the line printer, local and DECnet messages, and for direct memory access (DMA) device data transfers. Large buffers are maintained the extended buffer pool (XBUF). XBUF does not reside in the monitor's permanently mapped root but is dynamically mapped as needed using kernel APR 6.

On processors that support UNIBUS mapping for DMA devices, a portion of XBUF is permanently mapped by some number of UNIBUS mapping registers (UMRs).* This portion, which can be up to 32 K-words long, is used by the DMC-11 driver, primarily for transmitting and receiving DECnet messages. The remainder of the extended buffer pool, up to 400 K-words, is unmapped (but can be dynamically mapped with UMRs as needed) and is available for use by those monitor components that do not require UMRs.

### 5.3.2 Arrangement of Free Space

Since there are two types of small buffers — general buffers and FIP buffers — and two types of large buffers — mapped and unmapped — the monitor maintains four distinct buffer pools, named as follows:

- MONPOL - general small buffer pool
- FIPPOL - file processor small buffer pool
- LRGPOL - unmapped large buffer pool
- EXTPOL - mapped large buffer pool

Within each of these pools, free unallocated buffer space is arranged as a linked list of available memory. This is illustrated in Figure 5-5.

Each buffer pool has a 2-word header whose address is globally equated to the appropriate pool name, listed above. The first word of the header contains the pool's allocation increment, minus one: 31(10) for small buffers and 63(10) for large buffers. The second word contains a link to the first word of available memory within the pool.

Each segment of available memory also contains a 2-word header consisting of a link to the next available segment, followed by the size of the segment (including the 2-word header).

* UMRs are a hardware feature, functioning in much the same way as memory management APRs. They permit 22-bit memory addresses to be processed by the 18-bit UNIBUS. See Chapter 6.

Since small buffers all reside in the low end of memory, the links used for MONPOL and FIPPOL are all standard physical memory addresses and segment sizes are in bytes. However, since large buffer addresses frequently cannot be expressed in sixteen bits, the links used for LRGPOL and EXTPOL are memory management addresses — that is, the physical memory address divided by 64. Likewise, segment sizes for these pools are expressed in memory management allocation units, or "slivers," where a sliver is equal to 64(10) bytes.

Unallocated segments for each buffer pool are linked together in ascending order of physical memory address. The last available segment contains a value of zero in its link word.

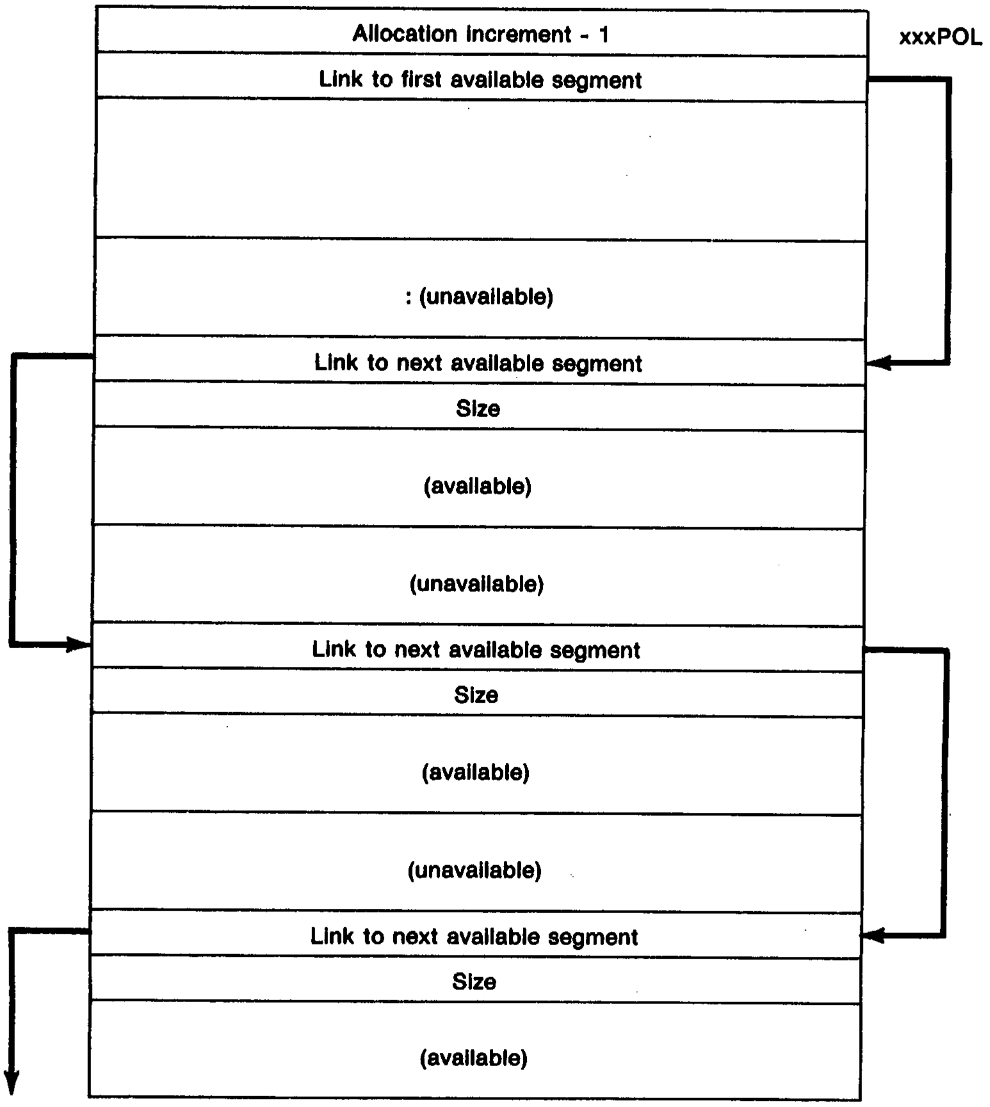

Allocation increment - 1
xxxPOL
Link to first available segment
: (unavailable)
Link to next available segment
Size
(available)
(unavailable)
Link to next available segment
Size
(available)
(unavailable)
Link to next available segment
Size
(available)

**Figure 5-5:** Unallocated Monitor Buffer Space

### 5.3.3 Buffer Usage

Within RSTS/E, buffer space is allocated using subroutine BUFFER. Input parameters to this subroutine include the following:

- Size (in bytes) of required space

- Pool in which to start the search for space
- Number of buffers to leave in the general small buffer pool (MONPOL)

Note that a request for buffer space specifies the amount of memory required, not the actual number of buffers. If the amount of space requested is not a multiple of 32(10) bytes (for FIPPOL or MONPOL) or 64(10) bytes (for LRGPOL or EXTPOL), the buffer allocator will round up the requested size and the caller will receive an integral number of buffers. The concern, however, is to provide the required amount of contiguous memory rather than a particular number of buffers.

### 5.3.3.1 Fall Back Allocation -

Note also that the pool specified is not always the pool from which space is ultimately allocated. RSTS/E uses a "fall back" procedure whereby an unsuccessful search for space in one pool can fall back to another, alternate pool: Requests for space in LRGPOL (the unmapped portion of XBUF) can and will be filled from mapped EXTPOL if no space is available in LRGPOL; requests for EXTPOL space can be filled from MONPOL; and requests for file processor small buffers can easily be filled with space from the general pool (MONPOL).

Because requests for space from any pool can ultimately be filled from the general small buffer pool, requests for buffer space must specify the minimum number of small buffers that must be left in MONPOL after the request is satisfied. This is a safeguard against exhausting the monitor pool. Since so many critical monitor data structures come from this pool, having no buffers available is likely to cause a system crash.

### 5.3.3.2 Contorted Addresses -

When a request for buffer space is granted, the monitor returns to the caller the actual number of bytes allocated along with a pointer to the beginning of the allocated memory. Since a request for large buffer space can be filled from the general small buffer pool, a mechanism is necessary to permit the requestor to determine whether the space returned is actually a large or a small buffer. The buffer pointer itself provides this mechanism. If the least significant five bits are zero, the pointer is the actual address of a small buffer. (Note that small buffers always begin on 40(8)-byte boundaries.) If these bits are not zero, the pointer is what is known as the "contorted" address of a large buffer: Kernel APR 6 has been mapped to the beginning of the buffer and the virtual address has been rotated left seven bits.

### 5.3.3.3 Large Buffer Headers -

When a request is made for a large buffer, the buffer allocator sets up a 4-word header at the beginning of the buffer. This header has the following format:

| *Offset* | *Symbol* | *Content* |
|---|---|---|
| 0 | BF.SIZ | This word contains the total size of the buffer (in bytes). |
| 2 | BF.OFF | This word contains the byte offset (from the beginning of the buffer) to the start of the data. The allocator sets this to an initial value of eight. |

| | | |
|---|---|---|
| 4 | BF.LNK | If the buffer is in a linked list, this word contains the link to the next buffer header. This pointer can be a standard address or a contorted address of another large buffer. The allocator sets this to an initial value of zero. |
| 6 | BF.CNT | This word contains the number of bytes of data contained in the buffer. The allocator sets this to an initial value of zero. |

This header can be used at the requestors discretion, but the information at BF.SIZ and BF.LNK must be retained for proper deallocation of the buffer when it is returned to free space.

### 5.3.3.4 Small Buffer Chain Blocks -

Under RSTS/E, devices that do not do transfers directly to and from user buffers can alternately store data in monitor buffer space while a transfer is taking place. Either large or small buffers can be used. If large buffers are used, the device driver must include its own data processing. In this case, the header mentioned in Section 5.3.3.3 can be used to keep track of the data.

When using small buffers, however, several common monitor subroutines are provided for the storing and retrieving of character-oriented data in a "chain" of small buffers — a series of small buffers linked together using the first word of each buffer as a pointer to the next. These common subroutines add data to the end of a chain and remove it from the beginning. When the last buffer is filled, another is allocated and linked onto the chain. When the first buffer is emptied, it is deallocated and returned to free space.

Any number of small buffer chains can be used by a single process, but each chain requires what is known as a small buffer control block or buffer chain block (BCB). The format of this block is as follows:

| *Offset* | *Symbol* | *Content* |
|---|---|---|
| 0 | EP | This word is the "empty pointer." It contains a pointer to the next byte of data to be removed from the chain. |
| 2 | FP | This word is the "fill pointer." It contains a pointer to the next available location in the chain where the next incoming byte of data is to be stored. |
| 4 | BC | This word is the buffer count. It contains the remaining number of small buffers that can be added to the chain. It is initially set to the small buffer quota specified for the process and is decremented each time a buffer is added to the chain and incremented each time a buffer is removed. If the value becomes negative, an attempt to add a buffer to the chain can fail due to an overall lack of buffers in the monitor pool (MONPOL). |

# CHAPTER 6
# Device Control [V8.0]

The RSTS/E operating system recognizes three basic types of devices: disks, terminals, and "others." Disk control is centralized, with all disk types sharing a common set of routines for processing functions above the actual device level. Terminal control is also centralized. Other devices, however, are considered separately, and each device type must have its own independent set of control routines, or device driver.

This chapter discusses control of "other" devices. (Disks and terminals are covered separately in later chapters.) Section 6.1 gives an overview of basic concepts, device characteristics, and processing flow. Section 6.2 discusses device identification, including the details of tables used by the monitor to parse a device identifier. Section 6.3 discusses device data blocks (DDBs), the primary control structure for individual devices. The general structure of a RSTS/E device driver is covered in Section 6.4. Section 6.5 gives the details of additional tables used by the monitor in individual device control.

## 6.1 General Overview

To a large extent, the control of an individual device depends on the specific operational characteristics of that particular device type. Some of these characteristics are inherent in the device itself, while others are imposed on the device by the driver software or by the setting of external parameters.

This section presents some of the basic characteristics of devices, followed by a discussion of the general processing involved in servicing a user request for a device function.

### 6.1.1 Device Characteristics

Devices operate either asynchronously or synchronously. In RSTS/E, an asynchronous device transfers data to and from monitor buffers, permitting the user job to continue execution while the actual physical transfer occurs. The job need only be stalled for I/O when a requested input is not totally contained in monitor buffers or when there are not enough buffers to hold a requested output. Synchronous devices, on the other hand, transfer data directly to and from the user's buffer and require that the job be stalled and locked in core while data is being transferred.

A device is either synchronous or asynchronous depending upon how the driver code is written. Character-oriented devices (transferring one byte at a time) typically operate asynchronously, while block-oriented devices (transferring more than one byte at a time) typically operate synchronously.

A block-oriented device that transfers data to and from the user's buffer without intermediate processing by the driver is known as a direct memory access (DMA) or nonprocessor request (NPR) device. Initiating a data transfer on such a DMA device entails supplying the device hardware controller with a physical memory address and a byte count. Once started, the transfer proceeds automatically without interference from the CPU. When the transfer is complete, the device generates an interrupt and the driver completes any necessary processing.

Since the PDP-11 UNIBUS has only eighteen address lines, device controllers that do DMA transfers can directly access only 128 K-words of memory. However, some PDP-11 processor models support 22-bit addressing and, therefore, can have up to 1920 K-words of physical memory. To permit full memory addressing capabilities for DMA devices on these processor models, a hardware mapping feature is available that functions much the same way as the hardware memory management unit discussed in Section 2.2.1. This feature uses 32(10) UNIBUS mapping registers (UMRs), numbered from 0 to 31, each capable of mapping up to four K-words each. UNIBUS mapping register 31 is hard-wired to the I/O page; seven UMRs are permanently assigned to the monitor's address space; and the remainder are available to permanently or dynamically map portions of the extended buffer pool or of user space (see Section 5.3.1).

All device drivers that perform DMA transfers must use UMRs when running on those processors that support them. A set of common monitor subroutines are provided for the allocation, initialization and deallocation of UMRs.

Because data transfers on DMA devices are automatic, once they are started they cannot be prematurely terminated. Asynchronous devices, however, can be interrupted. In RSTS/E, this is accomplished by typing a Control-C character at the terminal keyboard. At that point, the monitor aborts any asynchronous I/O in progress by clearing the "I/O in progress" bit (bit JFREDO in the JDFLG word of the job data block) and unstalling the job. The driver is not informed of the abort.

During timesharing, certain devices can be assigned to a job for its exclusive (temporary) use. In general, RSTS/E file-structured disks cannot be assigned but all other devices can be, depending on their physical availability. If a device is already assigned to a job, it cannot be assigned to another job until the first job releases it.

RSTS/E normally permits any job to use any device. However, the system manager can choose to restrict the use of certain devices to privileged users only. (Note that file-structured disks and the null device cannot be restricted in this fashion.) A restricted device can be accessed only by a permanently privileged user (that is, a user running under a privileged account) or by a nonprivileged user that has gained ownership of the device either by having a privileged user reassign the device to it or by running a privileged program that uses the device.

### 6.1.2 Processing Flow

Communication between a user job and a device driver is accomplished with either the transfer request block (XRB) or the file request queue block (FIRQB). The structure used depends on the requested function. When a device is opened or assigned, the FIRQB is used. All other functions use the XRB. (See Section 5.1.2 for descriptions of both these data strucutres.)

When a user request is made, the monitor EMT handler first copies the FIRQB or the XRB to the user job's work block (WRK). If the request is for an I/O service function, the appropriate bit is set in the job's JBWAIT entry, stalling the job. In most cases, this permits the job to be made eligible for swapping during processing of the request. However, if the function is a read or a write to a DMA device that transfers data directly between the user buffer and the device, the driver will lock the job in core so the user buffer remains available throughout the transfer.

Any common processing that can be handled by the monitor EMT handler or the file processor is done before control is actually passed to the driver. For example, most of the processing required to open or assign a device is done by FIP: checking user privilege, verifying that the device is not already assigned to someone else, updating the user's I/O block (IOB) pointers, and so on.

Any device control functions that require initial common processing by FIP use the special file processor function call, EMT 0. The monitor EMT handler dispatches control to the file processor EMT handler (FIPEMT), and FIP then retrieves the subfunction code from the FIRQB at offset FQFUN. Once all the common processing is complete, control is transferred to the driver.

As part of the monitor, all driver routines exit to the level three queue code. Normally this is accomplished by a simple return, transferring control back to the portion of the central monitor or the file processor that originally dispatched to the driver. These modules ultimately exit to the level three processing code at RTI3.

During the execution of most device functions, the requesting job is not stalled and remains eligible to run again when all monitor level processing is complete. However, if the requested function is an I/O service call (a read or a write), the EMT handler stalls the job before entering the driver. This is done by setting the appropriate bit in the job's JBWAIT entry and loading a zero into the job's JBSTAT entry (see Section 3.1.6). This makes the job unrunnable. The job remains stalled until the driver indicates that the I/O function is complete and sets the appropriate bit in the job's JBSTAT entry. This is done by calling routine IOEXIT before the driver returns from the I/O service routine.

Frequently the driver is unable to complete the requested I/O function immediately. This can happen for any number of reasons:

- There is not enough buffered data to complete a read.
- There is not enough available buffer space to store data for a write.
- A DMA transfer has not completed.
- Some system resource (such as a UMR) is required but not available.
- The device is not in the proper state.

If this happens, the driver initiates any necessary device operations and then exits to wait for the interrupt service routine to flag that processing can continue. To do this, the driver exits through routine IOREDO. This routine sets the "I/O in progress" bit (bit JFREDO in the JDFLG word of the job data block) and then exits to the level three code as usual. Later, when the interrupt handler detects the condition for which the main driver is waiting, the I/O service routine is reentered and processing of the requested user function continues.

Upon completion of the user request, any information that must be returned to the calling job is handled through the posting process, discussed in Section 5.1.3.

## 6.2 Device Identification

To access one of the peripheral devices configured on the system, the user provides a device specification, or "designator." This designator can be either the device's actual physical device name or a logical device name assigned to the device.

A physical device name consists of two alphabetic characters specifying the name of the device type, followed by a decimal unit number. A logical device name consists of one to six alphanumeric characters and can be assigned to a device alone or, in the case of file-structured devices, to a particular directory account on the device.

The system manager (or any other privileged user, for that matter) can associate certain system-wide logical names to devices and accounts. Once assigned, these names can be used by any job on the system. Alternately, each job can assign its own, private logical names to the devices it is using.

The monitor processes a device designator in the following order:

- First, the monitor determines whether the designator is one of the user job's private logical names.
- If it is not, the monitor next determines whether the designator is a system-wide logical name.
- If the designator is neither a private or a system-wide logical name, the monitor assumes it is a physical device name.

### 6.2.1 Physical Device Names

When a physical device name is specified, the monitor uses a series of related tables to validate the device designator and then map it to the proper control structures for the individual device.

Figure 6-1 shows the relationship of these physical device specification tables. You may wish to refer to this figure frequently throughout the remainder of this section.

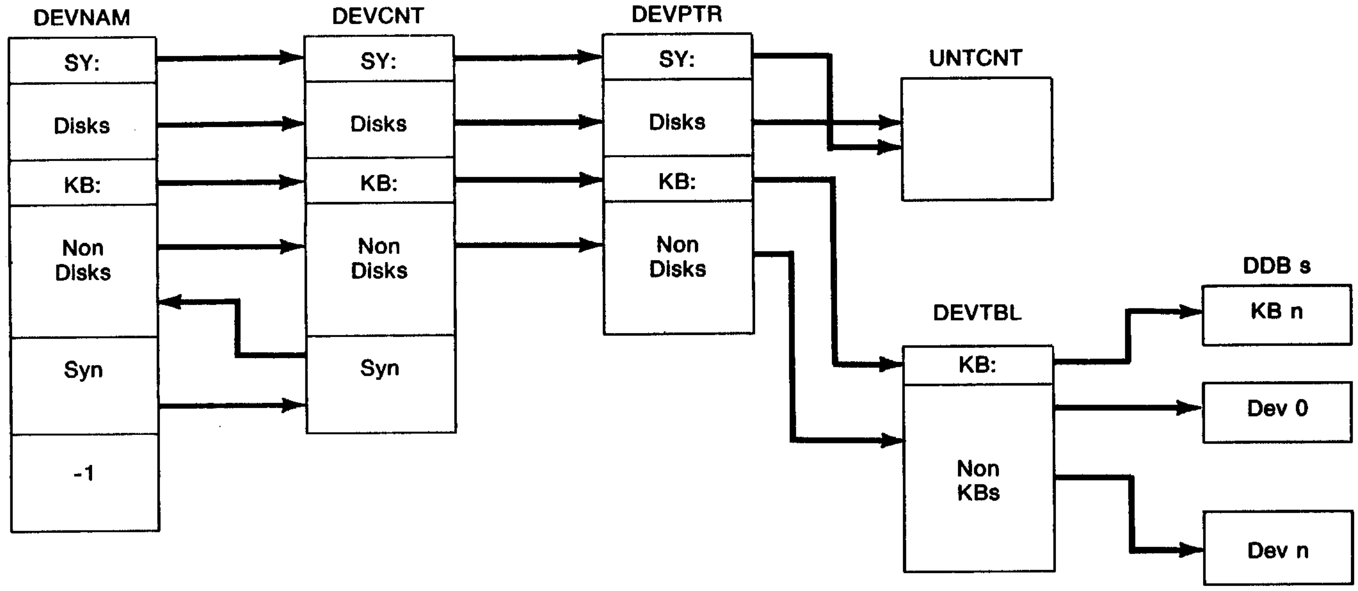


**Figure 6-1:** Device Specification Tables

#### 6.2.1.1 Device Type Names (DEVNAM) -

Given a physical device specification, the first table the monitor checks is the device name table (DEVNAM). This table contains a list of all valid device type names for the system.

Each 1-word entry in DEVNAM contains the 2-character ASCII name of the associated device type. Entries are included for every supported disk type (whether configured or not), as well as one entry for each nondisk device type configured on the system. There are also entries for all valid synonyms of physical device names (such as "TT" for "KB" or "MT" for "MM"). The end of the device name table is always signaled by a full word containing the value -1.

The length of the table is variable, depending on the number of different devices configured, but the general structure (as shown in Figure 6-2) is constant. Each section of the device name table is discussed below.

| *Address* | *Contents* |
|---|---|
| DEVNAM-2 | This entry always contains "SY", the standard name of the system disk. It is at offset -2 to indicate that it is not really a normal device type name. |
| DEVNAM | The entries that start at this offset contain the names of all disk types that are supported by RSTS/E, whether or not they are configured into the particular system. The number of entries varies from release to release. |
| DEVNKB | This entry always contains "KB", the standard name for all keyboard type devices, including pseudo-keyboards. |
| DEVNKB+2 | This entry always contains "NL", the name of the null device. |

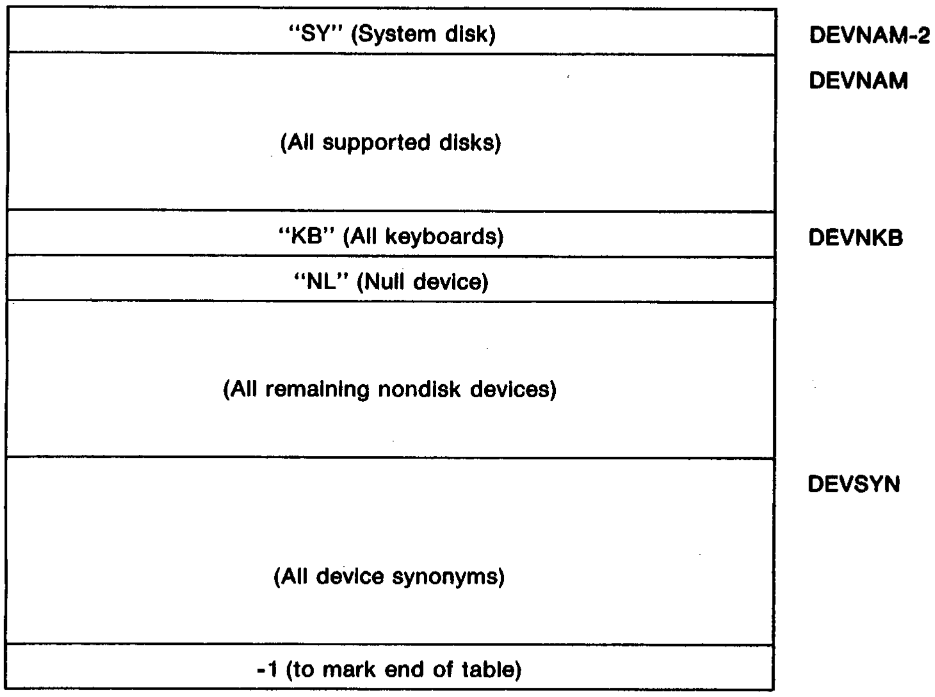


**Figure 6-2:** Device Name Table

DEVNKB+4 The entries that begin at this offset contain the names of all remaining nondisk devices configured on the system.

DEVSYN The entries that begin at this offset contain any alternate physical device names (synonyms) recognized by the system. These are different from logical device names since they are alternate names for device *types* rather than individual devices. Some standard synonyms are as follows:

TT for KB (keyboards)
CR for CD (card readers)
MT for MM (magnetic tapes)

In addition to the device names and synonyms specified in DEVNAM, there is one other valid 2-character device name always recognized by the monitor — "TI". "TI" is not actually the name of a particular physical device but can be used by a job to specify its console terminal, in much the same way that "SY" can be used to specify the system disk. Since it is essentially a logical device name (insofar as it specifies an individual device for each individual user), "TI" cannot be used with an explicit unit number. The form "TIn:" (including "TI0:") is therefore illegal.

### 6.2.1.2 Unit Numbers (DEVCNT) -

Once the monitor has verified that a given device type is valid, it validates the unit number. Note that the unit number is optional. When it is missing from the device designator, the following rules apply:

- If the device is a disk, the specification is considered valid, even if the particular disk named is not configured on the system. In addition, the specification is assumed to refer to the public disk structure as a whole. That is, it is considered a "generic" disk specification.
- If the device name is "KB" (or some appropriate synonym), the specification refers to the job's console terminal — the terminal that is open on channel 0. If the job is detached, it has no console terminal and the designator is in error.
- If the device is a nondisk, nonterminal device, unit 0 of the device type is assumed.

If the unit number is explicitly specified, or if unit 0 is assumed as in the last rule above, the monitor uses the device unit count table (DEVCNT) to check that the specified decimal unit number is within range.

The device unit count table has a format identical to that of the device name table (except that there is no terminating word of -1), and the same offset is used to index into both tables. The content of each 1-word entry in DEVCNT varies, depending on whether the corresponding entry in DEVNAM is an actual physical device name or a synonym.

For actual physical devices, the entry in DEVCNT contains the maximum valid unit number of the device. (For example, an entry of zero indicates that exactly one unit — unit 0 — exists for that device type.) A value of -1 indicates that no units of the corresponding device type have been configured in the system. (Note that the value corresponding to "SY" is always zero, indicating that there is only one system disk. Likewise, the value corresponding to "NL" is always zero.)

DEVCNT entries corresponding to device name synonyms contain a pointer back to the appropriate entry in the device name table of the actual physical device name. For example, the DEVCNT entry for the synonym "TT" contains the address DEVNKB — the DEVNAM entry for keyboards.

### 6.2.1.3 Mapping to Individual Device Structures (DEVPTR and DEVTBL) -

When the device type name and unit number have been validated, the monitor then uses a table of pointers (DEVPTR) to map the specification to the individual control structures for the particular device. The format of this table is basically the same as that of DEVNAM and DEVCNT, and therefore, the same offset is used. (Note, however, that there are no entries for device name synonyms. When this stage of processing has been reached, a synonym has already been translated into an actual physical device name.)

The content of each 1-word entry in DEVPTR varies according to whether or not the device in question is a disk. If the device is a disk, the DEVPTR entry (including the entry for "SY") contains a pointer into the disk unit count and status table (UNTCNT). (See Section 10.2.1 for details of the disk unit count and status table.)

If a particular disk type is not configured on the system, the DEVPTR entry points to the UNTCNT entry of the next configured disk type. If none of the remaining supported disks are configured, the entry points to a dummy entry at the end of UNTCNT. The DEVPTR entry for the system disk (SY), contains a pointer to the UNTCNT entry of the device type and unit from which the system was bootstrapped. Note that this is not necessarily unit 0 of the first supported disk type.

If the device in question is not a disk, the DEVPTR entry for the device points into the device retrieval table (DEVTBL). This table contains an entry for every unit of every nondisk device configured on the system. Each entry consists of a pointer to the device's private device data block (DDB). (See Section 6.3 for a discussion the the DDB.)

For both disks and nondisks, the address pointed to by the DEVPTR entry contains information for unit 0 of the specified device type. Information for additional units of the same device type immediately follow the unit 0 entry. See Figure 6-3.

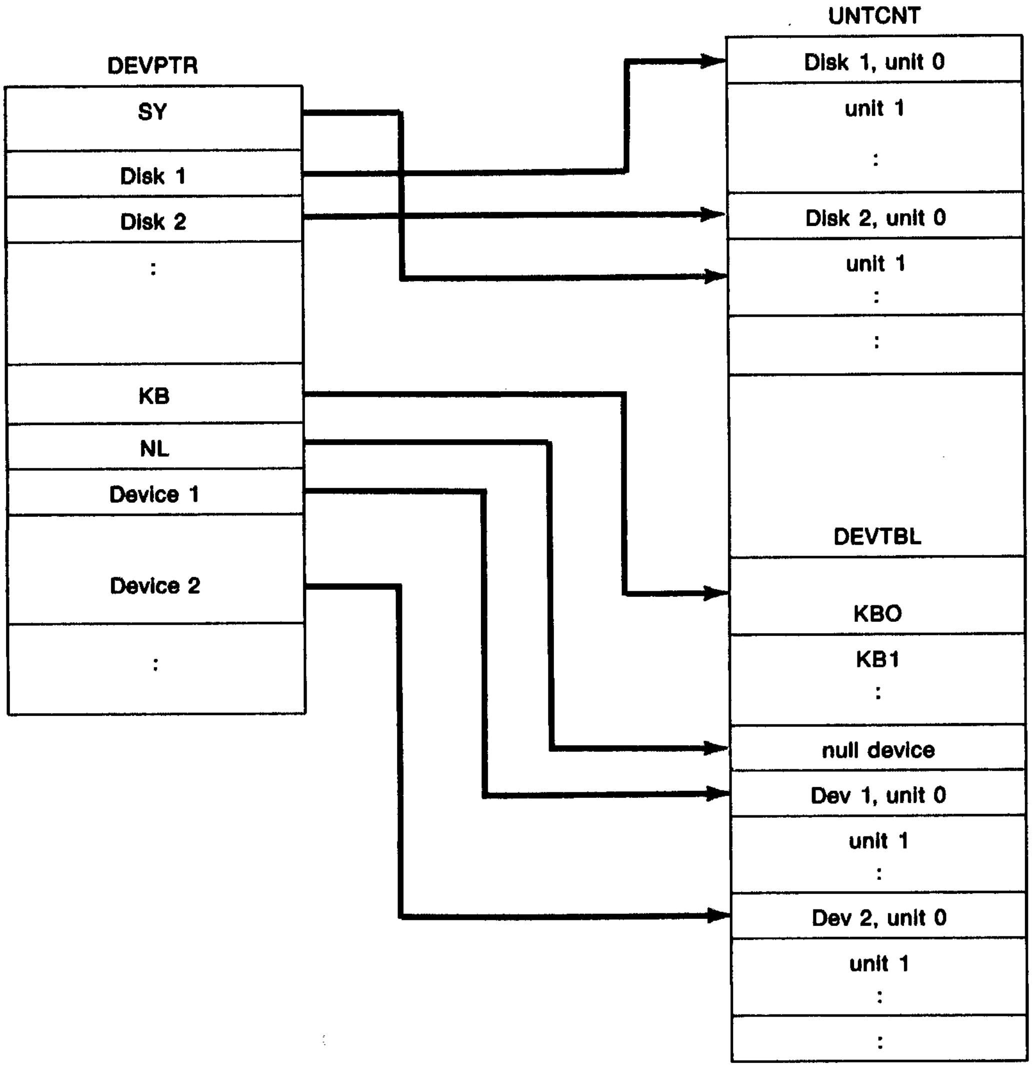


**Figure 6-3:** Mapping DEVPTR into UNTCNT and DEVTBL

### 6.2.2 Logical Device Names

As mentioned before, logical device names can be either system-wide or local to a particular job.

#### 6.2.2.1 System-Wide Logicals (LOGNAM) -

System-wide logical names include the pack ID for all mounted disk packs, the system library ("LB"), and any additional assignments defined by the system manager (or any other privileged user) with the UTILTY program.

The monitor uses the system logical name table (LOGNAM) to translate system-wide logical device assignments to physical device names and account numbers. This table contains one 5-word entry for every disk configured on the system, plus one entry for every possible system-wide logical name. The maximum number of system-wide logicals is declared during system generation, but the actual assignments can be added or deleted during timesharing. Figure 6-4 shows the format of each entry. Following the figure is a description of each entry.

| | | |
|---|---|---|
| Logical name (in RAD50) | | 0<br>2 |
| Device name (in ASCII) | | 4 |
| "Unit number is real" | Unit number | 6 |
| Project-programmer number | | 10 |

**Figure 6-4:** Logical Device Name Entry

*Offset* *Contents*

0 These two words contain the logical name, in RAD50 format. If the first word contains a zero, the entry is unused.

4 This word contains the physical device type name (in ASCII) of the device associated with this logical name.

6 These two bytes contain the device unit number and a flag that indicates that the unit number is, in fact, real (particularly a unit number of zero). Note that it is possible that the name at offset 4 does not have an explicit unit number (e.g., "SY" or "TI"). In such a case, *both* bytes will be zero.

10 If the device indicated at offset 4 is a file-structured device, this word contains the account number (if any) associated with the logical name assignment. The programmer (or user) number is in the low byte, and the project (or group) number is in the high byte.

The first entries in the logical name table are reserved for the pack ID of each disk unit on the system (ordered as they are in the device name table, DEVNAM). In these entries, the logical name field contains the ID of the pack currently mounted on the disk unit. If no pack is mounted, the name field contains zeros. In addition, the account number field of these entries always contains a zero.

The remaining entries in LOGNAM correspond to user-defined system-wide logical device names. (Note that these logical names can also include assignments for disk devices.) The first of these entries is initialized at system generation to the system library — SY:[1,1]. However, this entry can be modified or removed during timesharing.

The final entry in the table is followed by a full word containing a value of -1, terminating the table.

### 6.2.2.2 User Logicals (USRLOG) -

A user job's private logical device names are cataloged in the low core area of the job image, beginning at symbolic address, USRLOG. This 16(10)-word area can contain either three or four logical device name entries. Each entry is four words long and is identical in format to the first four words of entries in the system logical name table, LOGNAM. (See Section 6.2.2.1.)

If there are no account numbers associated with any of the user's logical assignments, all sixteen words in the USRLOG area can be used to catalog logical names, permitting a total of four logical assignments. However, if one or more of the user's logical assignments includes an account number, only three assignments can be made. The last four words of the USRLOG area are used to store accounting information (as shown in Figure 6-5), and thus, only three individual entries are possible.

| | |
|---|---|
| -1 (flag indicating this is account data) | USRLOG+30 |
| Project-programmer number for 1st logical | USRLOG+32 |
| Project-programmer number for 2nd logical | USRLOG+34 |
| Project-programmer number for 3rd logical | USRLOG+36 |

**Figure 6-5:** Account Numbers for User Logicals

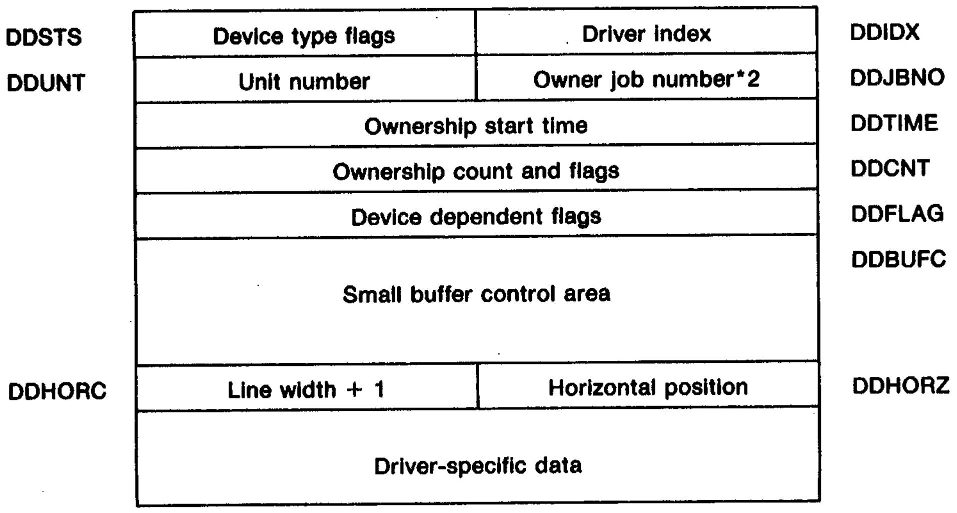
## 6.3 Device Data Blocks (DDBs)

The primary control structure for (nondisk) devices is the device data block (DDB). (Note that disk control structures are unique and are described in detail in Section 10.2.) There is one DDB for every individual nondisk device unit configured on the system. DDBs are allocated during system generation and are located in the permanently mapped root of the RSTS monitor phase.

Device data blocks are used to control the actual use of a device by its driver and to communicate between the driver and common monitor support routines. They contain information concerning ownership status and small buffer usage, as well as other information specific to the particular device.

The address of a particular device data block can be accessed by the monitor in two ways. Given a standard device designator, the DDB is accessed using the procedure and multilayered data structures discussed in Section 6.2. However, if the device has been previously opened by a job, associating it with an I/O channel, the address of the device data block is stored in the job's IOB (see Section 3.1.4). In this case, the job and channel numbers are sufficient to reference the DDB.

Since the information required for device control varies for different types of devices, the size and content of individual DDBs also varies. However, a certain minimal set of information must be present. Figure 6-6 shows the format and content of the device data block. The first four words are required for all devices. The next five words are standardized for devices that use them. The remaining space is totally device-specific. Following the figure is a description of each entry.



| | | | |
|---|---|---|---|
| DDSTS | Device type flags | Driver index | DDIDX |
| DDUNT | Unit number | Owner job number*2 | DDJBNO |
| | Ownership start time | | DDTIME |
| | Ownership count and flags | | DDCNT |
| | Device dependent flags | | DDFLAG |
| | Small buffer control area | | DDBUFC |
| DDHORC | Line width + 1 | Horizontal position | DDHORZ |
| | Driver-specific data | | |

**Figure 6-6:** Device Data Block

*Offset* *Contents*

DDIDX This byte contains the driver "index," used to access the required device driver through the driver dispatch and data tables. Every driver is assigned a unique, non-negative even index during system generation. (See Section 6.4.1.)

DDSTS This byte contains a set of flags describing the characteristics of the device. The setting of these bits is defined as follows:

*Bit* *Symbol* *Meaning*

8 DDPRVO If this bit is set, ownership of the device is reserved to privileged users. Most devices are shared on a first come, first served basis and do not require privilege. Some devices, however, may be considered critical enough to be reserved for privileged ownership only. Any nondisk, nonnull device can run in either state.

9 DDRLO This bit is used by the monitor general purpose I/O routines to determine whether a requested read function is valid. If set, it indicates that read requests are not allowed for this device unit; it is a write-only device (such as a line printer).

10 DDWLO This bit is used by the monitor general purpose I/O routines to determine whether a requested write function is valid. If set, it indicates that write requests are not allowed for this device unit; it is a read-only device (such as a card reader).

NOTE

A device unit that is normally read/write can be opened in such a fashion as to render it read- or write-only. In this case, it is up to the device driver to set or clear either DDRLO or

| | | |
|---|---|---|
| | | DDWLO when the device is opened and to restore it to its general state when the device is closed. |
| 11 | DDNET | This bit indicates that the device is not owned by a user job but is instead owned by the DECnet/E software. |
| 12 | (Unused) | |
| 13 | DDAUX | This bit indicates that the device can use KMC-11 bridge blocks. Bridge blocks provide a means of communication between the computer and a KMC-11 processor. |
| 14 | DDAUXA | If this bit is set, the device is currently bridged to a KMC-11. (This bit has no meaning if bit DDAUX is zero.) |
| 15 | DDSTAT | This bit is cleared by the general-purpose monitor routines whenever the device is closed. The bit is not actually used by the monitor but can be assigned by the driver to represent a condition that should be automatically reset each time the device is closed. |

DDJBNO This byte contains the job number (multiplied by two) of the user job that owns the device (either explicitly through the ASSIGN command or implicitly through an OPEN). If the value here is zero, then the device is free and is not owned by any job. If the value here is one, then the device has been disabled and is not available for use. (Since this byte is specified as the job number *multiplied by two,* the value in this byte for an enabled device will normally be even. However, under certain special circumstances, the value here can be an odd number greater than one. This indicates that the device is owned by a monitor component (such as DECnet) rather than a user job.)

DDUNT This byte contains the unit number of the device.

DDTIME This word contains the system time at which the device was assigned. It is used to charge device time to the owning job.

DDCNT This word controls access to the device. If ownership of the device is implicit (through an OPEN), the low-order byte is used to maintain a count of the number of users that have the device open. If ownership of the device is explicit (through an ASSIGN), the low-order byte is zero, but one or more flag bits are set in the high-order byte. Each of these flag bits is defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8-12 | | (Unused) |
| 13 | DDCONS | If this bit is set, the device is the owning job's console terminal. The bit is used to differentiate a terminal owned by a job for general purpose I/O operations and the terminal on which the job was created. |
| 14 | DDUTIL | If this bit is set, the device is temporarily assigned to the file processor. |
| 15 | DDASN | If this bit is set, the device has been explicitly assigned to the job through use of the ASSIGN command. |

DDFLAG This word contains device-specific status flag bits. It is not used by the monitor.

DDBUFC This 3-word area is required if the device driver uses the common monitor character buffering routines (FETCH, STORE, CLRBUF and so on) that store characters in small buffer chains. These three words contain the necessary buffer chain block (BCB). (See Section 5.3.3 for a discussion of monitor buffer usage and the format of the BCB.)

DDHORZ This byte contains the number of positions (or columns) remaining on the print line (if any) of the device and corresponds to the current horizontal position of the device print head. A value of zero indicates the right hand margin. That is, no more characters can be printed on the line. It is the responsibility of the device driver to maintain this byte if it is applicable.

DDHORC This byte contains the line width of the device, plus one. Normally, this byte contains some predetermined value, corresponding to the maximum horizontal position of the unit's print head. However, the driver can choose to make this value variable at the time the device is opened. If so, it is the responsibility of the driver to modify this byte appropriately and then to reset it to the default value (if any) when the device is closed.

DDHORC+1 The remainder of the DDB is used to hold device-specific information. The driver can set up this area in any way it wishes.

## 6.4 General Driver Structure

Device drivers running under RSTS/E must conform to a specific format, with certain required program sections (PSECTs) and processing routines. In addition, these pieces must be named in accordance with strict global naming conventions. All required global symbols incorporate the 2-character ASCII name of the device type for which the driver is written. (This is denoted throughout this section as "xx".)

The remainder of this section discusses the required program sections and service routines of a RSTS/E device driver.

NOTE

This discussion is intended to be a general overview of driver structure. It is not meant to be a step-by-step description of how to write a device driver; such a discussion is beyond the scope of this document.

### 6.4.1 Driver Index

Every RSTS/E device driver is assigned a unique driver "index" during system generation. Indexes are non-negative, even numbers and are used to access values within the various driver data and dispatch tables maintained by the monitor. The driver index for each device unit is always stored in the unit's device data block (if any), as well as in any file control block (FCB) that the device driver may generate. (See Section 9.1.6 for a discussion of file control blocks.)

Driver indexes are defined symbolically as "IDX.xx", where "xx" is generally the physical device type name. Four driver indexes are permanently assigned. These are as follows:

| *Value* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | IDX.SY | This is the index for all disk types. Since disk devices do not have DDBs, the driver index is stored in the FCB of any open disk file. |
| 2 | IDX.KB | This is the index for all terminal devices on the system. |
| 4 | IDX.NL | This is the index for the null device. |
| 6 | IDX.PK | This is the index for all pseudo-keyboards. |

All other driver indexes are allocated starting with a value of ten and increasing by two until all separate drivers have been assigned a unique index.

### 6.4.2 PSECT Usage

Device drivers must be divided into program sections, or PSECTs, each with a specific name and serving a specific purpose. Each PSECT is located in and mapped with a particular portion of the RSTS phase of the monitor, as noted in the following subsections. Note, therefore, that if data space is enabled for the monitor, the various I&D mapping conventions within the RSTS phase dictate a corresponding mapping for each of the PSECTs. (See Section 2.2.1 for a discussion of I&D space and related monitor memory mapping conventions.)

Five PSECTs are possible. Two are required and three are optional.

#### 6.4.2.1 Driver Code (xxDVR) -

The driver code PSECT (xxDVR) contains all processing code and is required for all drivers. It is located in the read-only portion of the monitor's address space and is dynamically mapped, as needed, using kernel APR 5. If data space is enabled for the monitor, xxDVR is mapped in both instruction and data space.

#### 6.4.2.2 Interrupt Dispatch Code (xxDINT) -

The interrupt dispatch PSECT (xxDINT) contains code necessary to map and dispatch to the interrupt service routine in the xxDVR program section. It is required for all device drivers and is located in the read-only portion of the permanently mapped monitor root. If data space is enabled for the monitor, xxDINT is mapped in both instruction and data space.

The interrupt vectors for all devices handled by the driver point to this PSECT. If more than one controller is processed by the driver, each interrupt vector can map to a separate entry point within xxDINT. Alternately, the unit number can be encoded into the low-order four bits of the processor status word (following the transfer address) within the 2-word vector. When an interrupt occurs, the unit number can then be decoded before control is passed to the interrupt service routine in xxDVR.

#### 6.4.2.3 Permanently Mapped Code (xxDISP) -

The permanently mapped code PSECT (xxDISP) was added in version 7.1 specifically to take further advantage of the data space feature of the memory management hardware. It is intended to hold any driver code that must reside in the monitor's permanently mapped root and that needs mapping only in instruction space. It is an optional PSECT, but by moving code from xxDINT to xxDISP, a corresponding amount of data space is made available for additional small buffers.

#### 6.4.2.4 Read/Write Data (xxDCTL) -

The read/write data PSECT (xxDCTL) is optional and is used when the driver must maintain general read/write data apart from the specific data kept for each unit. If all information can be stored in the device data blocks, this PSECT is not necessary. The xxDCTL program section is located in the read/write portion of the monitor's address space, within the permanently mapped root. If data space is enabled for the monitor, xxDCTL is mapped in data space only.

#### 6.4.2.5 Read-Only Data (xxDTBL) -

The read-only data PSECT (xxDTBL) is optional. It is located in the read-only portion of the monitor's permanently mapped root and is used to hold data that must be available to other device drivers and to other parts of the monitor. If data space is enabled for the monitor, xxDTBL is mapped in data space only.

### 6.4.3 Service Routine and Monitor Dispatch Tables

A RSTS/E device driver consists of a collection of service routines that perform the device-specific processing in response to user requests for I/O functions, device-generated interrupts, or other asynchronous events.

There are twelve types of driver service routines. Of these, eight are required and four are optional. Each service routine must begin at a specifically named entry point. The entry points into all drivers are then cataloged in a series of dispatch tables -- one table per function. Each table is accessed by driver index.

Before dispatching to a service routine, the monitor performs any general processing such as validating arguments, locating the appropriate device data block, and so on. If there is no appropriate device-specific processing to be performed by a particular required routine, the driver will exit. Likewise, if the associated function is illegal for the device, the driver will return an error and exit.

#### 6.4.3.1 Assign (ASN$xx and ASNTBL) -

The ASN$xx service routine is entered to perform any device-specific functions at the time the device is assigned. It is required for all device drivers, and is entered when the device is assigned, either explicitly (with the ASSIGN command) or implicitly (with the OPEN command when it is not already assigned).

When control is transferred to this entry point, the monitor has already located the device data block and has updated the owner information. Since this service routine is entered before the device is opened for the first time, it is a good place for general device initialization of nonstandard, user-written drivers. Generally, however, no device-specific processing is required for the assign function and the ASN$xx routine usually simply exits.

[One exception to this is the null device driver. Since the null device cannot be assigned, ASN$NL cancels any assignment processing previously done by the monitor.]

The device assignment entry points for all device drivers are cataloged in the monitor dispatch table, ASNTBL, ordered by driver index.

#### 6.4.3.2 Deassign (DEA$xx and DEATBL) -

The DEA$xx service routine is entered to perform any device-specific functions at the time the device is deassigned. This routine is required for all device drivers, and is entered when the device is either explicitly deassigned (with the DEASSIGN command) or implicitly deassigned (with the CLOSE command when it is no longer open by any other users).

When control is transferred to this entry point, the monitor has already located the device data block and has updated the ownership information. Generally, no device-specific processing is required for the deassign function and DEA$xx usually simply exits.

The device deassignment entry points for all device drivers are cataloged in the monitor dispatch table, DEATBL, ordered by driver index.

#### 6.4.3.3 Open (OPN$xx and OPNTBL) -

The OPN$xx service routine is entered to perform any device-specific initialization at the time an OPEN command is issued for the device. It is required for all device drivers.

When control is transferred to this entry point, the monitor has already located the device data block and has updated the ownership information. A copy of the user's FIRQB has also been loaded into the job work block. Typical device-specific functions performed at this entry point include validating device-dependent arguments, updating information in the DDB, and initializing the device to its required state.

The device open entry points for all device drivers are cataloged in the monitor dispatch table, OPNTBL, ordered by driver index.

#### 6.4.3.4 Close (CLS$xx and CLSTBL) -

The CLS$xx service routine is entered to perform any device-specific functions necessary at the time a CLOSE command is issued for the device. This routine is required for all device drivers.

When control is transferred to this entry point, the monitor has already located the device data block and has updated the ownership information. It has also updated any required accounting information and has zeroed the channel number in the job's I/O block. Typical device-specific functions performed at this entry point (if no other user has the device open) include resetting the device to its idle state, discarding any remaining input data, releasing any monitor buffers, and restoring default values (if any) in the DDB.

The device close entry points for all device drivers are cataloged in the monitor dispatch table, CLSTBL, ordered by driver index.

#### 6.4.3.5 I/O Service (SER$xx and SERTBL) -

The SER$xx service routine is entered whenever an I/O request for a read or a write is issued for the device. This routine is required for all device drivers.

When control is transferred to this entry point, the monitor has already verified that the device is open and that the caller has the necessary access rights for the requested function.

The job has been stalled waiting for I/O and will remain stalled until the driver calls the IOEXIT routine, indicating that the I/O is complete. A copy of the user's XRB has also been loaded into the job's work block.

Processing within this service routine depends on whether or not the device does DMA transfers. If it does, the job is locked in core, the transfer is started, and the routine exits using IOREDO. When the transfer is complete, SER$xx is reentered. The routine then exits using IOEXIT to indicate that the transfer is complete and to unstall the job.

If the device does not do DMA transfers but uses buffered asynchronous transfers instead, processing depends on the requested function. For a read, SER$xx determines whether there is already enough data currently in small buffers to satisfy the request. If so, the data is transferred to the caller and the service routine exits using IOEXIT. If there is not enough data currently in small buffers, the routine will transfer what data it has to the user's buffer and then exit using routine IOREDO. When the interrupt processing has buffered enough data to complete the request, SER$xx is reentered, the remainder of the data is transferred to the caller, and the routine exits using IOEXIT. The job remains stalled throughout this process. (See Section 6.1.2.)

If the requested function is a write, the data is transferred to the small buffer chain and output to the device is started. If there are not enough small buffers, the service routine exits using IOREDO. When the necessary buffers become available, SER$xx is reentered, the data is transferred to the buffers, output is started, and the routine exits using IOEXIT. (See Section 5.3.3 for a complete discussion of the use of monitor buffer space for buffering data.)

To ensure that the calling job does not remain stalled indefinitely if the device does not respond, the I/O service routine typically initiates a timeout period after starting the data transfer. (See Section 6.4.3.9 for details on the timeout process.)

The I/O service entry points for all device drivers are cataloged in the monitor dispatch table, SERTBL, ordered by driver index.

#### 6.4.3.6 Interrupt Service (INT$xx) -

The INT$xx service routine is entered every time the device generates an interrupt. This routine is required for all device drivers. Execution of the routine is controlled by the hardware rather than through a monitor dispatch table. The interrupt vectors of all devices handled by the driver point to the interrupt dispatch code in the xxDINT PSECT of the driver. Control then passes to the central driver code at this entry point.

Since the interrupt service routine runs at a higher processor priority than the bulk of the monitor, it should keep processing to a minimum. Thus, any processing that does not have to be done at interrupt level is typically coded into a level three queue service routine (entry point nnn$xx -- see Section 6.4.3.7) that can then be scheduled by INT$xx through the level three queue mechanism. When the interrupt service routine exits through the level three queue code, the monitor reenters the driver at the indicated entry point to complete processing.

When the interrupt service routine (or an associated level three queue routine) detects a condition for which the central driver has been waiting (such as the availability of additional small buffers or the end of a data transfer), it calls routine IOFINI just prior to exiting. IOFINI flags the stalled user job as runnable by setting the appropriate bit in the job's JBSTAT entry. When the job scheduler is run, it will attempt to run the job. Note, however, if JFREDO is set in the JDFLG word of the job's data block, the job will not run. Instead, the central driver will be reentered at its SER$xx entry point. The driver can then complete any necessary processing before unstalling the calling job.

### 6.4.3.7 Level Three Queue Reentry (nnn$xx) -

The optional nnn$xx entry points are used primarily to reenter a driver to complete complex interrupt processing at a lower processor priority level.

Reentry is accomplished through use of the level three queue mechanism (see Section 2.3.2). To determine whether there are outstanding monitor-level processes that should be scheduled, the level three queue code tests two flag words, L3QUE and L3QUE2. Thus, a total of 32(10) processes can be scheduled in this fashion. A number of the flag bits are reserved for standard device drivers and monitor processes, such as the scheduler and the memory manager. The remaining bits are available for use by nonstandard device drivers.

When a driver contains a level three queue service routine, the entry point name is of the form "nnn$xx" — where "nnn" is a 3-character mnemonic associated with the L3QUE flag bit and "xx" is the device name. The L3QUE bit name is then "Qxxnnn".

### 6.4.3.8 Error Logging (ERL$xx and ERRTBL) -

The ERL$xx service routine is entered whenever the driver itself wishes to enter an error in the system error log. It is optional and is necessary only if the driver does its own error logging. However, if the service routine is not included, the entry point symbol ERL$xx must still be defined globally, equated to zero.

The system generation procedure defines a special EMT code (LOG$xx) for drivers that log their own errors. When this EMT is issued by the driver, processing is interrupted and control transfers to the monitor's central error logging facility. The error logger then transfers control to the driver's ERL$xx entry point. After error processing is complete, control returns again to the main error logger, which sends any error message to ERRCPY. The device driver mainline then resumes execution at the place where it was interrupted.

The error logging entry points for all device drivers are cataloged in the monitor dispatch table, ERRTBL, ordered by driver index. If the driver does not contain this entry point, the ERRTBL entry will be zero.

### 6.4.3.9 Timeout (TMO$xx and TMOTBL) -

The TMO$xx service routine is entered whenever the device's timeout period has expired. It is a required routine for all device drivers. Device timeouts are used primarily for two reasons: to trap devices that do not respond to an I/O request and to perform periodic device-specific checks and functions.

Every unit of every device configured on the system has an entry in the monitor timeout table, TIMTBL. To activate a standard timeout, the driver loads the proper entry in TIMTBL with a positive nonzero value representing the timeout period, in seconds. The monitor clock handler decrements every activated entry once each second. When an entry goes to zero, the appropriate driver is entered at its TMO$xx entry point. Some device drivers may retry a timeout several times, but ultimately, if the device does not respond, the driver will typically release all monitor buffers, log a "hung" device error, and unstall the calling job.

To activate a periodic timeout, the driver loads the proper TIMTBL entry with a negative nonzero value. This signals the monitor that the driver is to be entered at its TMO$xx entry point every second.

To cancel a timeout, the driver loads the appropriate entry in TIMTBL with a value of zero.

The timeout entry points for all device drivers are cataloged in the monitor dispatch table, TMOTBL, ordered by driver index.

#### 6.4.3.10 Sleep Check (SLP$xx and SLPTBL) -

The SLP$xx service routine is entered whenever a user of the device requests a conditional sleep. The driver can then determine (using whatever criteria it wishes) whether or not the job should be permitted to suspend execution. It is an optional routine.

The sleep check entry points for all device drivers are cataloged in the monitor dispatch table, SLPTBL, ordered by driver index. If the driver does not contain this entry point, the symbol SPL$xx will be undefined and the SLPTBL entry will be zero.

#### 6.4.3.11 Special Service (SPC$xx and SPCTBL) -

The SPC$xx service routine is entered whenever a request for special function handling is issued for the device. It is required for all device drivers.

When control is transferred to this entry point, the monitor has already loaded a copy of the user's XRB into the job work block. If the driver does not support special function handling for the device, it should return an error to the user and exit.

The special service entry points for all device drivers are cataloged in the monitor dispatch table, SPCTBL, ordered by driver index.

#### 6.4.3.12 UNIBUS Mapping Register (UMR$xx and UMRKB) -

The UMR$xx service routine is entered when a UNIBUS mapping register (UMR) becomes available. It is an optional routine.

When a driver is denied access to a UNIBUS mapping register because there are none available, the monitor flags the fact that such a request has been made. Later, when a UMR becomes available, the monitor begins execution of all drivers that contain a UMR$xx service routine. Note that entry here does not guarantee that the driver now has use of the available UMR. The driver must reissue the request, and the request may still fail if another driver has already allocated the free UMR.

The UNIBUS mapping register entry points for all device drivers are cataloged in the monitor dispatch table, UMRKB, ordered by driver index. If the driver does not contain this entry point, the symbol UMR$xx will be undefined and the UMRKB entry will be zero.

## 6.5 Additional Monitor Support Tables

In order to provide generalized device processing wherever possible, the monitor maintains several tables containing device characteristics and information. Tables used to process user device specifications were discussed in Section 6.2. The remainder of this section presents the tables used in general processing of user requests for device functions.

In most cases, these tables contain one entry per device type and are accessed by driver index. Two exceptions to this are CSRTBL and TIMTBL. Each of these latter tables contains one entry for every unit of every nondisk device on the system. The first entry for each separate device type (that is, the entry for unit 0) is defined by a global symbol of the form CSR.xx and TIM.xx, respectively. Entries for the remaining units immediately follow the entry for unit 0.

These tables reside in the read/write portion of the monitor's permanently mapped root.

### 6.5.1 Device Dependent Flags (FLGTBL)

Each entry in FLGTBL contains the device dependent flags and the driver index for the associated device type. The flags are in the high byte and the driver index is in the low byte.

When a device is opened, the flags from this table are returned to the calling job in word FQFLAG of the FIRQB. The setting of each of the flag bits is defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | DDNFS | The device is a non-file-structured device and does not require a file name. If a file name is supplied, it will be ignored. |
| 9 | DDRLO | The device is inherently read-locked. It cannot do input. |
| 10 | DDWLO | The device is inherently write-locked. It cannot do output. |
| 11 | FLGPOS | The device maintains its own horizontal position. The current horizontal position is stored in byte DDHORZ of the DDB. |
| 12 | FLGMOD | The device uses modifiers specified at offset XRMOD of the XRB. (See the *RSTS/E System Directives Manual* for a discussion of modifiers.) |
| 13 | FLGFRC | The device is byte-oriented. It does not require that a specific number of characters be transferred on each I/O request. |
| 14 | FLGKB | The device is a terminal. |
| 15 | FLGRND | The device is inherently a random-access device. |

### 6.5.2 Line Width (SIZTBL)

Each entry in SIZTBL contains the line width, or maximum horizontal position, of the associated device type. Three types of values are possible:

5*14.+1 The device is a blocked device (such as a disk or magnetic tape) and has no line width.

width+1 The device line width is fixed at the size indicated.

0 The device line width is variable and is maintained by the driver in byte DDHORC of the DDB.

### 6.5.3 I/O Buffer Size (BUFTBL)

Each entry in BUFTBL contains the default buffer size (in bytes) for the associated device type. When a user job opens a device, it can specify any buffer size it wishes. However, if none is specified, the value stored in BUFTBL is used as a default and is loaded into the FIRQB, at offset FQBUFL, before the driver is called.

Values here must be even and can range from 2 to 32,766(10).

### 6.5.4 JBWAIT/JBSTAT Status Bits (JSBTBL)

Each entry in JSBTBL contains the bit to set in the user job's JBWAIT entry while it is stalled for I/O from the associated device type. (See Section 3.1.6 for a description of the JBWAIT table.)

### 6.5.5 Timeout Counters (TIMTBL)

There is one entry in TIMTBL for every unit of every nondisk device configured on the system. Each entry contains a timeout counter for the associated unit.

To activate a timeout, the device driver sets the proper TIMTBL entry to a positive nonzero value. The monitor then decrements this counter once each second. If the value goes to zero, the device driver is entered at its TMO$xx entry point. If the counter value is negative, the driver is entered at its TMO$xx entry point every second. The driver can cancel the timeout by loading the TIMTBL entry with a value of zero. (See Section 6.4.3.9.)

The table is initially loaded with zeros. A timeout period can range from 1 to 32,767(10) seconds.

Since there is one entry for each unit, different timeouts can be activated simultaneously for different units of the same device type.

### 6.5.6 Control and Status Register Addresses (CSRTBL)

There is one entry in CSRTBL for every type of device controller configured on the system. Each entry contains the address of the control and status register (CSR) for the associated unit.

This table is allocated during system generation and contains entries for all devices -- both disk and nondisk. However, only RSTS-standard device CSR addresses are defined. Entries for devices supported by nonstandard drivers contain zeros unless they are patched to contain the proper address. Since the table resides in the read-only portion of the monitor, it cannot be modified during timesharing.

# Part III
# DISK I/O AND THE FILE PROCESSOR

Disks are random-access devices. They are the fastest, most reliable and most durable type of peripheral device. Disks can also store more data than any other device.

Access to disk media under RSTS/E is normally controlled by the file processor (FIP) and by the disk I/O subsystem. To read or write disk data, the user job must first call the file processor to associate the disk data (a disk file, disk directory, or the whole disk) with a channel. The file processor does access and existence checking and then "opens" the requested channel. Actual reading and writing of data are handled by the disk I/O subsystem. This set of routines uses information set up by FIP to map logical user I/O requests into physical disk reads and writes. When processing is complete, FIP "closes" the file and releases the channel.

This part of the *RSTS/E V8.0 Internals Manual* presents some basic terminology related to RSTS/E disks (Chapter 7), descriptions of the actual on-disk structures (Chapter 8), overviews of the file processor (Chapter 9), an overview of the disk I/O subsystem (Chapter 10), and a discussion of directory and data caching (Chapter 11).

# CHAPTER 7
# Disk Terminology [V8.0]

Disks are the most important form of mass storage device in today's computer systems. Thus, the terminology developed to discuss various disk-related concepts has become quite extensive

This chapter defines some of the terms used within the RSTS/E environment to describe certain types of disks as well as the various subdivisions of disk data files.

## 7.1 Disk Types

Physically, disks come in a variety of types: fixed or moving head, single or multi-platter, and so on. In addition to these physical categories, disks are separated into types according to their use and their on-disk structure.

### 7.1.1 Public versus Private Disks

RSTS/E is a disk-based system. That is, during timesharing, some parts of the monitor and run-time system code are always in memory, while other parts of the system, including user programs, are stored on disk and are loaded into memory only when needed. In addition, user programs that have suspended execution for one reason or another can be temporarily stored on disk (or "swapped out") in order to make room for other programs that are eligible to run. Thus, every RSTS/E system requires at least one disk, known as the system disk.

The system disk must be physically mounted and accessible at all times during timesharing. It contains the monitor code, as well as other system files and programs. It also contains information listing all users allowed to log onto the system, and provides storage for files created by those users.

A single disk may not satisfy the storage requirements of all system users. Additional disks that are shareable by all system users are known as public disks. They are considered extensions of the system disk and should also be physically mounted and accessible during timesharing.

The system disk and all public disks are treated by RSTS/E as one unit, known as the public structure. A good deal of system file activity takes place on the public structure. Since it is the largest constantly available medium of file storage, it is generally the busiest.

Not all the disks used on a system must be part of the public structure. Some can be private disks that belong to a single user, or to a group of users. Only users with "accounts" on a private disk pack can create files there. Private disks can be mounted or dismounted at any time during timesharing. Since access is restricted on private disks, such packs are useful for the storage of files that contain sensitive information.

### 7.1.2 File-Structured versus Non-File-Structured Disks

Disks can be either file-structured or non-file-structured. A RSTS/E file-structured disk is divided into separate areas called files, with each file containing a group of related data. These files are stored so that RSTS/E can locate and access any file individually. The mechanism for doing this is a list (called a directory) of all the files on the disk.

Non-file-structured disks contain no means for identifying and locating individual files. RSTS/E, therefore, must access data on such a disk as if it were one large file.

NOTE

> Many operating systems have file structures that are totally incompatible with the structure used by RSTS/E, and disks generated under those systems cannot be accessed using RSTS/E file-structured operations. They must be treated as non-file-structured devices.

## 7.2 Logical and Virtual Blocks

Under RSTS/E, disk data is never accessible a byte or a word at a time. Instead, all disks are divided (either by the disk hardware itself or artificially by the software) into a collection of logical blocks (LB). A logical block consists of 256(10) contiguous words of disk data and is the smallest unit of data accessible to the software. Each logical block is assigned a unique 23-bit logical block number (LBN). LBNs start at zero and increase by one, ordered in such a fashion that the data contained in block LBN n+1 is physically adjacent to and immediately following the data contained in block LBN n.

NOTE

> Throughout this document, the terms "block" and "logical block" are used interchangeably.

Every disk type has a maximum logical block number (MLBN) corresponding to the highest logical block number possible on that particular type of disk. Since logical block numbers begin with zero, a disk's MLBN is equal to its capacity in blocks, minus one.

All data accessible through the RSTS/E file-structured disk operations is contained in files. A file is an ordered set of virtual blocks (VBs), where a virtual block is equal in size to a logical block. Each virtual block in a file is assigned a unique number called its virtual block number (VBN). VBNs are numbered consecutively from 1 to n, where n is the total number of logical blocks in the file. All user-level file-structured access to data within a file is by VBN. The file processor and the disk I/O subsystem map the specified VBN to a unique LBN.

Under the RSTS/E file structure, the blocks of a file are pure data and contain no linking or structural information.

## 7.3 Clusters

The logical block is the smallest unit of data accessible to the software and is the fundamental unit with which all RSTS/E disk drivers deal. However, the nondriver disk management software frequently requires an even larger unit with which to work. Thus, most device, directory, and file information is accessed by cluster.

A cluster is a collection of sequential logical blocks. The number of blocks in a cluster (the "cluster factor") is always a power of two — 1, 2, 4, 8, 16, 32, and so on. By restricting the number of blocks in a cluster to a power of two in this fashion, cluster factors that are not of equal size are at least guaranteed to be multiples of one another.

There are a variety of different cluster factors, used for different purposes.

### 7.3.1 Device Clusters

The primary storage unit of a RSTS/E disk is the device cluster (DC). It is the smallest allocatable unit of disk data for a particular disk type and is the unit used in all on-disk linking and structural information.

Every disk type has a permanently assigned device cluster size (DCS) (less than or equal to sixteen) chosen in such a way that the device cluster number (DCN) remains a 16-bit number. DCNs start at zero and increase by one until all logical blocks of the disk are contained in a device cluster.

### 7.3.2 Pack Clusters

When a disk pack is initialized with the RSTS/E file structure, it is assigned a pack cluster size (PCS). A pack cluster (PC) is the smallest unit that can be allocated on a particular disk pack or cartridge. Each PC is represented by one bit in the pack's storage allocation table (see Section 8.6.4).

The pack cluster size assigned to each pack is chosen so the pack cluster number (PCN), like the device cluster number, remains a 16-bit number. PCNs start at zero and increase by one, up to the maximum pack cluster number, as described below.

The pack cluster size must be greater than or equal to the device cluster size, with a maximum of sixteen. The ratio of the pack cluster size to the device cluster size (PCS/DCS) is a constant known as the cluster ratio (CLURAT). This constant is calculated when a disk is mounted and is used frequently by the file processor in various arithmetic manipulations.

Because DCN 0 of each disk pack contains the boot block for the pack, the logical blocks in DCN 0 are not available for data storage. Thus, PCN 0 is always aligned with DCN 1 and the logical blocks in DCN 0 are not contained in any pack cluster. Note that if the device cluster size is greater than one, DCS-1 logical blocks can never be used to store data and are therefore "lost."

In addition to these lost blocks at the front of the disk pack, more logical blocks can be "lost" at the end of the pack. If the total number of logical blocks on the pack is not an even multiple of the pack cluster size, a certain number of blocks will be left over after all pack clusters have been assigned. Since data on a particular pack cannot be allocated in amounts less than the pack cluster size, these blocks will be unaccessible. Therefore, the maximum pack cluster number (MPCN) is the total number of complete pack clusters, which may not, in fact, represent the total capacity of the disk pack.

Since the pack cluster is the primary storage allocation unit for a pack, it is desirable for PCs to align on the physical disk so they do not cross a disk track or cylinder boundary. However, since one device cluster is unaccessible preceding PCN 0, the pack clusters can be aligned in such a way that they split across track and/or cylinder boundaries. Clusters that split across a track boundary do not normally cause much of a problem; accessing a different track usually involves merely an electronic switch to the next read/write head. Accessing a different cylinder, however, involves actual arm movement (on moving head disks) which can result in a relatively long delay before the data transfer can be completed.

### 7.3.3 File Clusters

The blocks of a file are also grouped into clusters. A file's cluster size is specified when the file is created and must be greater than or equal to the pack cluster size.

The cluster size of a file has important ramifications. Since retrieval information for the file is stored according to file cluster, a larger cluster size reduces the amount of retrieval information required and therefore decreases overall file access time. When a file is open for access, retrieval information for seven file clusters is maintained in memory-resident data structures. Thus, accessing a block in any one of those seven clusters is relatively fast. However, if the desired block resides somewhere else, another set of retrieval pointers must be fetched from the disk. (This process is known as a "window turn" and is discussed in detail in Chapter 9.) A larger file cluster size increases the chances that the desired block is already described in memory.

Another advantage to a larger file cluster size is that less directory space is required to catalog the file. On the other hand, any unused blocks at the end of the last cluster of the file are completely wasted.

### 7.3.4 Directory Clusters

RSTS/E on-disk directories are themselves files and, as such, are structured in clusters. A directory cluster size must be greater than or equal to the pack cluster size for the pack, but it can be no larger than sixteen. The directory cluster size is specified when the directory is created.

Since a directory can consist of a maximum of seven clusters, the cluster size determines the total number of files that can be cataloged.

(Chapter 8 discusses the RSTS/E directory structure in detail.)

### 7.3.5 Cache Clusters

The RSTS/E caching feature stores blocks of disk data in central memory for direct access during future disk I/O operations. There are two types of caching: data caching, whereby blocks of data from user files are stored, and directory caching, whereby directory information and data from monitor files is stored.

When caching is enabled, the cache software receives all data transfer requests that would otherwise be directed to the disk driver. Read operations requesting data already in memory can then be satisfied without placing a load on the disk driver. All write operations are "written through" the cache. That is, the blocks to be written are copied first to central memory and then to the disk.

Data caching is done using a unit known as a cache cluster — a sequential collection of 1, 2, 4, or 8 logical blocks. Cache cluster numbers start at zero (aligned with DCN 1) and increase by one until all blocks of the disk are contained in a cache cluster. When a read request is received for a block that is not already in memory, the cache cluster that contains the desired block is read and stored. Thus, if the cache cluster size is eight and a read request is received for LBN 10, blocks 8 to 15 will be read and stored in central memory.

The cache cluster size must be greater than the device cluster size and, for optimal performance, should be less than or equal to the pack cluster size.

(Caching is discussed in detail in Chapter 11.)

## 7.4 File Processor Blocks

While all disk accesses at the driver level are done by logical block, all nondriver monitor software references what are known as file processor (or FIP) blocks (FB). Each FIP block is equal in size to a logical block (256(10) words) and is assigned its own 23-bit FIP block number (FBN).

FIP block numbers start at zero and increase by one so that there is one unique LBN for each FBN. However, while FBN 0 corresponds to LBN 0, FBN 1 corresponds to the first block of DCN 1. Thus, for device cluster sizes greater than one, DCS-1 logical blocks are lost and do not correspond to a FIP block. (Figure 7-1 presents this graphically for a device cluster size of four.)

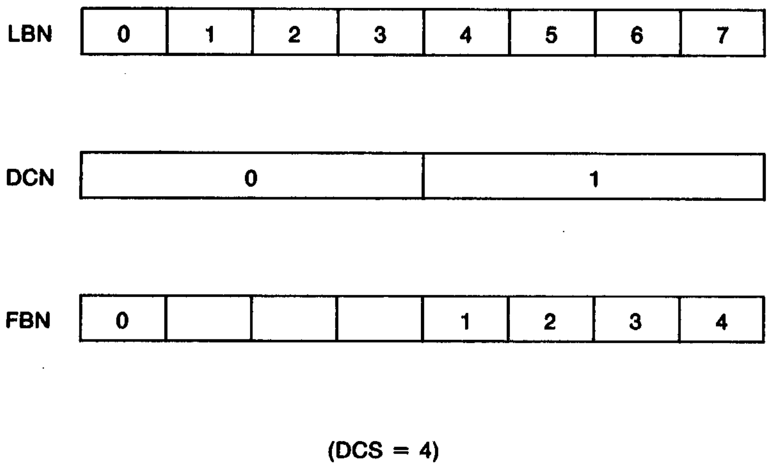


**Figure 7-1:** Relationship of Logical Blocks to Device Clusters and FIP Blocks

The file processor calculates FBNs from the on-disk linking and structural information (always stored as 16-bit DCNs) using the following algorithm:

```
 IF DCN = 0 THEN FBN = 0
              ELSE FBN = (DCS * (DCN-1)) + 1
```

All nondriver monitor software deals exclusively with FBNs. It is a starting FBN, along with the total number of blocks to read or write, that is passed to the disk I/O subsystem during execution of an I/O operation. The disk I/O subsystem then computes the correct LBN from the FBN, using the appropriate device cluster size. The correct LBN is then passed to the proper disk driver for the actual physical operation.

This scheme of having two types of block numbers may very well seem confusing and unnecessary. Indeed, if the device cluster size is equal to one, the FBNs and LBNs are identical. Why, then, the extra level of complexity?

When the file processor and on-disk structures were originally developed, the largest disk supported by RSTS/E had a capacity of only 40,000 blocks. Thus, one word of storage for a logical block number seemed adequate. However, as larger and larger disks were introduced, it became clear that sixteen bits of storage were not enough. To support these larger disks, the RSTS/E development group was faced with a major decision: either redesign the existing on-disk file structure (thus invalidating the disks of all current users), or find an upwards compatible method of extending the file structure. It was decided to extend the existing structure.

Device clusters were devised as a means of mapping logical block numbers back into sixteen bits for use in on-disk linking and structural information. However, at that time, the first block of the master file directory had to be addressed by a value of one, be it a 16-bit value or a 23-bit value.* Thus, DCN 1 had to map into a block number of one. This could have been accomplished by specifying DCN 0 as a 1-block cluster. However, there are advantages to keeping device clusters in pure linear alignment with logical blocks, particularly in the handling of non-RSTS/E disks. Thus, the concept of 23-bit block numbers (manipulated completely within the file processor) was developed.

## 7.5 File Processor Unit Numbers

Each disk unit configured on the system is assigned a file processor (or FIP) unit number (FUN). FUNs are logical device unit numbers used internally for all disk operations. They are numbered starting from zero and are incremented by one until every disk unit on the system has been assigned a unique number. (Note that the system disk does *not* necessarily have an FUN of zero.)

The majority of the disk device control tables are ordered by FIP unit number. Thus, given a FUN, the file processor and the disk I/O subsystem can immediately access information concerning the specific disk unit's status and characteristics.

* This restriction has since been lifted.

# CHAPTER 8
# On-Disk Structures [V8.0]

The RSTS/E disk structure uses a 3-level directory hierarchy to organize user accounts and files. The primary directory structure is the master file directory (MFD). The MFD catalogs group file directories (GFDs), the second level of the directory structure. Each GFD, in turn, catalogs user file directories (UFDs), the third level of the directory structure. UFDs catalog user files.

There is one (and only one) MFD on each disk pack. The MFD can catalog up to 255(10) GFDs, and each GFD can catalog up to 255(10) UFDs.

There is one UFD for each user account which can contain files. Each account is specified by a group and user number (also referred to as a project-programmer number, or PPN) of the form "[group,user]". Since access to the RSTS/E system is controlled through the use of these PPNs, the MFD/GFD structure on the system disk also acts as the master list of users that are permitted to log onto the system.

This chapter discusses the RSTS/E on-disk data structures. The first two sections cover some basic concepts: Section 8.1 discusses blockettes, the basic unit of the directory structure; Section 8.2 presents the format of directory linkage information. The details of the master file directory are covered in Section 8.3, the group file directory is described in Section 8.4, and the user file directory is described in Section 8.5. Section 8.6 discusses the minimal RSTS/E file structure.

## 8.1 Directory Blockettes

A disk block is 256(10) words long. Each disk block within a directory is further subdivided into 32 8-word "blockettes" numbered from 0 to 31. Blockettes are important because they are the primary unit on which directories are built.

NOTE

> Blockettes are occasionally referred to as "entries." However, data files are also commonly referred to as directory entries. Thus, to avoid any unnecessary confusion, the term "blockette" is used throughout this document.

There are several types of blockettes, each with a different format and each serving a different purpose. Those blockettes that are specific to MFDs, GFDs, or UFDs are discussed in the appropriate sections later in this chapter. Three types of blockettes are common to all structures, however, and are discussed in the following subsections.

### 8.1.1 File Directory Cluster Map (FDCM)

With certain exceptions, blockette 31 of every logical block within a directory contains the file directory cluster map (FDCM) for that directory. As mentioned in Chapter 7, a RSTS/E directory is a file and is structured in clusters. The FDCM contains the retrieval pointers necessary to access these clusters.

Figure 8-1 shows the format of the file directory cluster map. The first word contains the directory cluster size — a power of two, between 1 and 16. This word is used to generate linking information within the directory but is not needed to interpret such information. The cluster size is specified when the directory is first created and cannot be changed.

The high-order bit of the cluster size is used as a flag to indicate the type of structure for which the cluster map is specified. If the bit is set, the cluster map is within either an MFD or a GFD.

The remaining seven words in the FDCM contain the device cluster numbers of the start of each directory cluster. Nonexistent clusters are indicated by a DCN of zero. Note that a cluster map is never sparse. That is, if the DCN of directory cluster n is nonzero, then the DCN of every preceding cluster is nonzero also.

| | | | |
|---|---|---|---|
| 1 | flag | Directory cluster size | 0 |
| 3 | DCN of directory cluster 0 | | 2 |
| 5 | DCN of directory cluster 1 | | 4 |
| 7 | DCN of directory cluster 2 | | 6 |
| 11 | DCN of directory cluster 3 | | 10 |
| 13 | DCN of directory cluster 4 | | 12 |
| 15 | DCN of directory cluster 5 | | 14 |
| 17 | DCN of directory cluster 6 | | 16 |

**Figure 8-1:** File Directory Cluster Map

The cluster maps in each directory block of a particular directory are identical. Each time a directory is extended (which can occur a maximum of seven times over the lifetime of the directory), all of the cluster maps are updated. (Since a directory is limited to seven clusters of sixteen logical blocks each, there can be a maximum of 112(10) logical blocks — and cluster maps — in a directory.)

Linking information within a directory consists of bits that select one retrieval pointer from the FDCM, bits that select a block within the cluster, and bits that select a blockette within the block. Thus, given a link and the cluster map, only one disk access is required to move from one directory block to another. (The structure of directory linking information is discussed in Section 8.2.)

### 8.1.2 Attribute Blockette

The RSTS/E directory structure contains various pieces of extended information concerning groups, user accounts, files, and even the pack itself. This extended information is stored in attribute blockettes that can be linked together for a particular group, user or file.

The format of an attribute blockette is shown in Figure 8-2. Following the figure is a description of each entry.

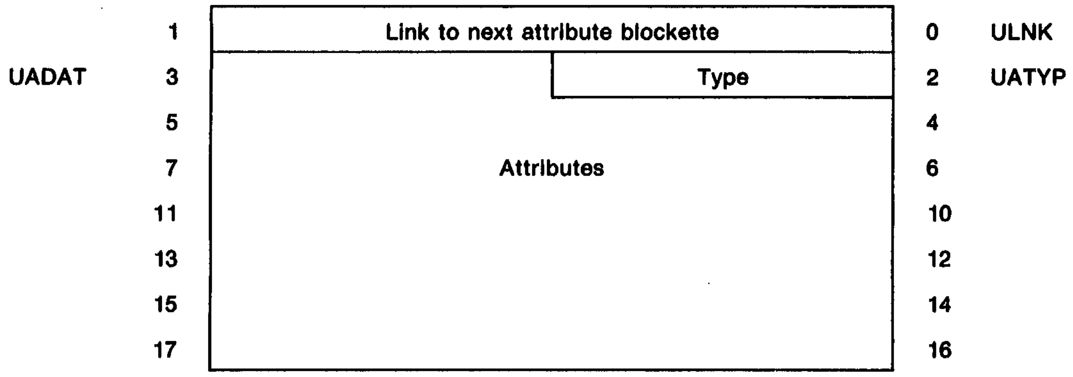


**Figure 8-2:** Attribute Blockette

*Offset* *Symbol* *Contents*

0 ULNK Attribute blockettes are chained together as a singly linked list. This word contains the directory link pointing to the next blockette in the list. If this is the last blockette, the link will be null.

2 UATYP This byte contains a code (in the range of 1 to 255(10)) identifying the nature of the data stored in bytes 3 to 16. Note that the defined codes vary in meaning according to the particular structure to which the blockette is linked. [Note also that file attribute blockettes (for files accessed under the RMS data management system) do not have this field; the byte at this offset is included in the attribute data field, giving a full seven words for file attribute data.]

The defined codes are as follows:

For pack attributes:

| | |
|---|---|
| 1-255 | (reserved for future use) |

For group attributes:

| | |
|---|---|
| 1-255 | (reserved for future use) |

For account attributes:

| | |
|---|---|
| 1 | Quotas |
| 2 | Privilege mask |
| 3 | ASCII password |
| 4 | User name |
| 5-199 | (reserved for future use) |
| 200-255 | (available for user-specific data) |

3-16 UADAT This area holds the attribute data which varies according to the defined code type.

### 8.1.3 Holes

Not all blockettes in a directory block contain useful information. Those that do not are known as "holes" and represent available blockettes that can be used for later entries as new files are created and added to the directory. When a file is deleted from the directory, the blockettes allocated to it revert to holes, available for reuse at a later time.

By convention, if the first two words of a blockette are zero, the blockette is a hole. The remaining six words are irrelevant.

## 8.2 Directory Links

Within the RSTS/E directory structure file entries are linked together, as are the blockettes that describe each entry. All links, regardless of their purpose, have the same format, shown in Figure 8-3.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UL.BLO | | | | UL.CLO | | | UL.ENO | | | | | (Flags) | | | |

**Figure 8-3:** Directory Link Word

Each link word defines a cluster, a block within the cluster, and a blockette within the block. Thus, given a link and a cluster map, it is possible to move from one directory block to any other directory block with only one disk access.

The fields contained in a link word are defined as follows:

*Field* *Contents*

UL.BLO These four bits select the block within the directory cluster. Since the maximum directory cluster size is sixteen, the legal range of values for this field is from 0 to 15(10).

UL.CLO These three bits select one out of seven possible directory clusters. They are used to index into the file directory cluster map to select one of the starting device cluster numbers. The legal range of values is from 0 to 6.

UL.ENO These five bits select the blockette within the directory block. The legal range of values is from 1 to 30. (Blockette 0 of block 0 of cluster 0 (the label blockette of each structure), along with blockette 31 in each block (the cluster map), are never the targets of link words.)

(Flags) These four bits are normally zero, permitting the blockette number in bits 4 to 8 to be used as a direct index into the cluster map. There are circumstances, however, when these bits are used as special flags.

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | UL.USE | This bit is used to ensure that the blockette does not appear as a "hole" when the link is null. |
| 1 | UL.BAD | If this bit is set, the file contains a bad block. |
| 2 | UL.CHE | This bit is a cache-related flag. It's meaning depends on the type of blockette in which the link word is located. |

3 UL.CLN This bit is used by the disk initialization utilities to indicate whether the disk pack needs to be rebuilt (or "cleaned").

A null link word is essentially zero. It points to the directory label blockette — a conveniently useless pointer. The special bit flags do not have to be zero for a link to be null, however. In fact, bit 0 is frequently set to prevent the blockette from appearing as a hole. Null links must have bits 4 to 15 clear.

## 8.3 Master File Directory (MFD)

The master file directory (MFD) is the root of the RSTS/E directory structure. It catalogs group file directories (GFDs), which in turn catalog user file directories (UFDs), which catalog files.

The MFD is created by the DSKINT program, either as an option of the INIT.SYS initialization code or as an on-line utility. DSKINT creates a minimal MFD cataloging one group file directory (group 0) for nonsystem disks, or two group file directories (groups 0 and 1) for system disks. Additional GFDs are created as required during normal timesharing.

The master file directory can consist of up to seven clusters, with a cluster size of from 4 to 16. Thus, there are 112(10) blocks possible in the MFD. Of these, the first three are always used.

Block 0 of cluster 0 (pointed to by the pack label, see Section 8.6.2) contains the MFD label in blockette 0 and the cluster map in blockette 31. The next two blocks (blocks 1 and 2) are not formatted into blockettes (and do not contains cluster maps) but are formatted instead as tables. The second block contains the starting device cluster numbers of all allocated group file directories and the third block contains directory link words pointing to the group attribute chain for each allocated GFD.

The remaining blockettes in block 0, along with those in blocks 3 and beyond, are used to store group and pack attributes. Attribute blockettes are described in Section 8.1.2.

Figure 8-4 shows the general structure of the MFD. Each entity is discussed in detail in the following sections.

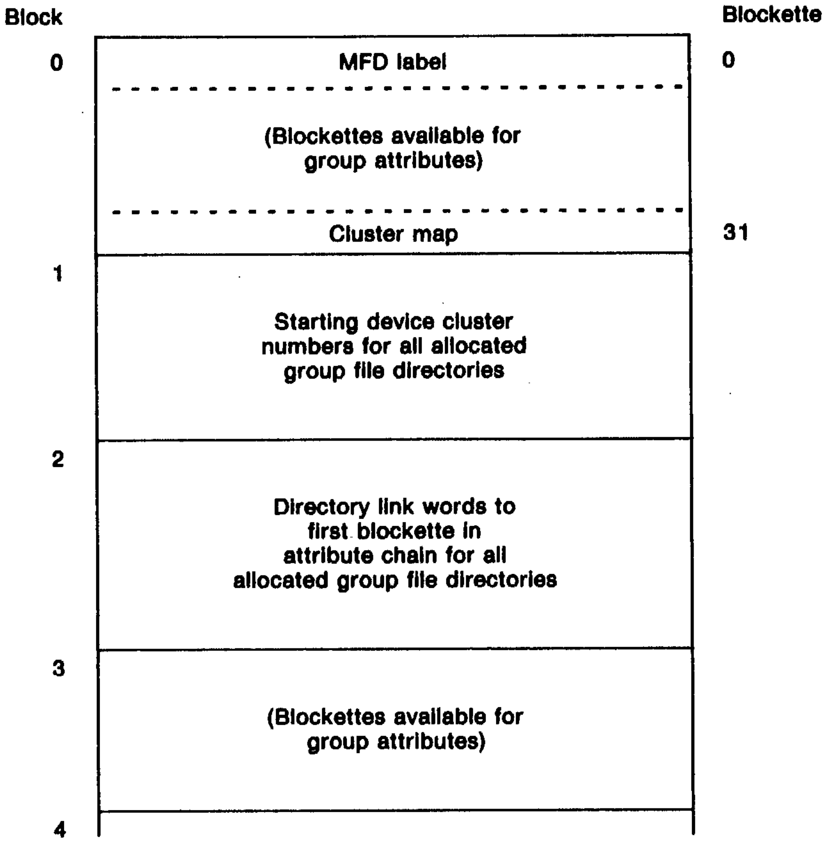

Block
Blockette
0
MFD label
0
(Blockettes available for group attributes)
Cluster map
31
1
Starting device cluster numbers for all allocated group file directories
2
Directory link words to first blockette in attribute chain for all allocated group file directories
3
(Blockettes available for group attributes)
4

**Figure 8-4:** General MFD Structure

### 8.3.1 MFD Label Blockette

Within the MFD, blockette 0 of block 0 of cluster 0 contains the MFD label. The primary function of this label is to permit the master file directory to be found using disk reconstruction tools.

Figure 8-5 shows the format of the MFD label blockette. Following the figure is a description of each entry.

| | | | | |
|---|---|---|---|---|
| 1 | 0 | | 0 | |
| 3 | -1 (to mark blockette in use) | | 2 | |
| 5 | 0 | | 4 | |
| 7 | 0 | | 6 | |
| 11 | 0 | | 10 | |
| 13 | Link to pack attributes | | 12 | MALNK |
| 15 | 255(10) | 255(10) | 14 | LPPN |
| 17 | "MFD" (in RAD50 format) | | 16 | LID |

**Figure 8-5:** MFD Label Blockette

| *Offset* | *Symbol* | *Contents* |
|---|---|---|
| 0 | | (unused) |
| 2 | | This word must be nonzero to mark the blockette in use. |
| 4-10 | | (unused) |
| 12 | MALNK | This word contains the directory link work to the pack attribute blockette (if any). Attribute blockettes are discussed in Section 8.1.2. |
| 14 | LPPN | These two bytes each contain a value of 255(10), indicating that this label is the MFD label. |
| 16 | LID | This word contains the 3-character string "MFD" in RAD50 format. |

### 8.3.2 GFD Device Cluster Numbers

Block 1 of cluster 0 of the master file directory is structured as a table, consisting of one 1-word entry for each possible group. There can be up to 255(10) groups, numbered from 0 to 254.

Each entry within the table is indexed by the group number, multiplied by two, and contains the starting device cluster number (DCN) of the group file directory. If no GFD has been allocated for a particular group, the corresponding entry within this table contains a value of zero.

The 256(10)-word table is terminated with an entry of zero.

### 8.3.3 Group Attribute Links

Block 2 of cluster 0 of the master file directory is structured as a table, consisting of one 1-word entry for each possible group. There can be up to 255(10) groups, numbered from 0 to 254.

Each entry within the table is indexed by the group number, multiplied by two, and contains the directory link word to the beginning of the group attribute chain (if any) within the MFD. (Attribute blockettes are described in Section 8.1.2.) If no group file directory has been allocated for a particular group, or the group has no attributes associated with it, the corresponding entry within this table contains a value of zero.

The 256(10)-word table is terminated with an entry of zero.

## 8.4 Group File Directories (GFDs)

Group file directories (GFDs) are the second level of the RSTS/E directory structure. They are used to catalog user accounts, which in turn are used to catalog user files. There can be up to 255(10) GFDs — one for each possible group, numbered from 0 to 254. Each GFD can catalog up to 255(10) user accounts.*

The starting device cluster number of every GFD allocated on a particular disk pack is stored in the second block (block 1 of cluster 0) of the pack's master file directory (see Section 8.3.2).

* These limits, of course, are theoretical. In practice, the actual limits are determined by the size of individual files and directories and by the total amount of available disk space.

A group file directory is created when the first user account is created (through the "create user account" system directive) with the corresponding group number. Once created, a GFD is never deleted.

Like the master file directory, a GFD can consist of up to seven clusters, with a cluster size of from 4 to 16. Thus, there are 112(10) blocks possible in a GFD. Of these, the first three are structured very much like the first three blocks of the MFD.

Block 0 of cluster 0 contains the GFD label in blockette 0 and the cluster map in blockette 31. The next two blocks (blocks 1 and 2) are not formatted into blockettes (and do not contain cluster maps) but are formatted instead as tables. Block 1 contains the starting device cluster numbers of all allocated user file directories. Block 2 contains the directory link word of the user name blockette (within the GFD) for each user account.

The remaining space in block 0, along with that in blocks 3 and beyond, is used to store name, accounting and attribute blockettes describing each user account.

Figure 8-6 shows the general structure of the GFD. Each entity is discussed in detail in the following sections.

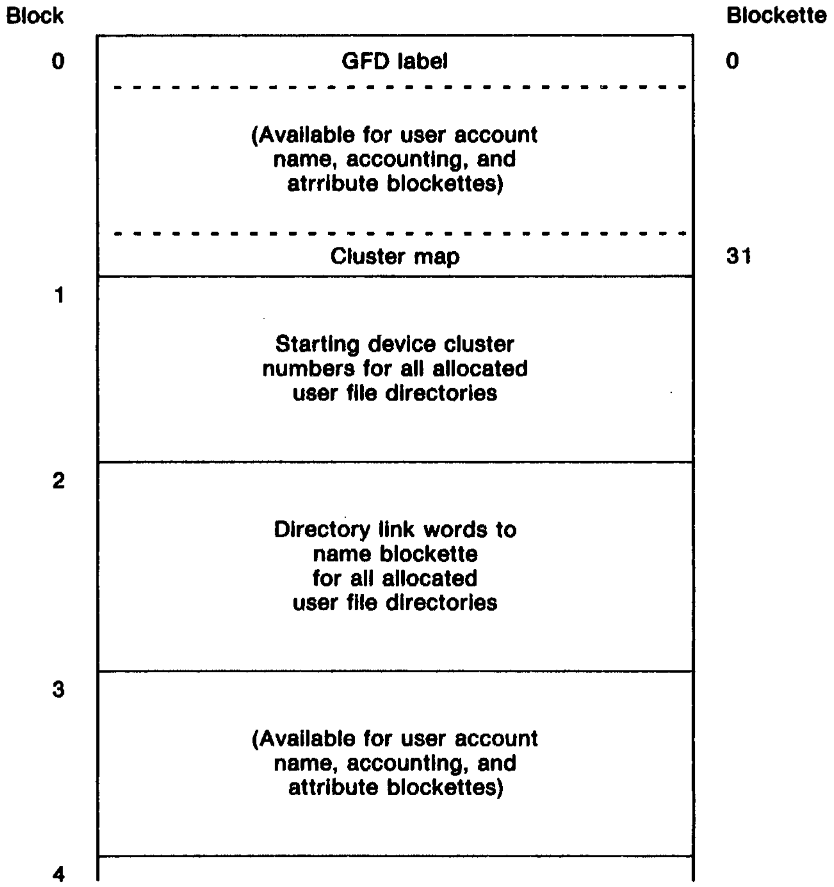

Block
Blockette
0
GFD label
0
(Available for user account name, accounting, and atrribute blockettes)
Cluster map
31
1
Starting device cluster numbers for all allocated user file directories
2
Directory link words to name blockette for all allocated user file directories
3
(Available for user account name, accounting, and attribute blockettes)
4

**Figure 8-6:** General GFD Structure

### 8.4.1 GFD Label Blockette

Within the GFD, blockette 0 of block 0 of cluster 0 contains the GFD label. The primary function of this label is to permit group file directories to be found using disk reconstruction tools.

Figure 8-7 shows the format of the GFD label blockette. Each field is described below.

| | | | | |
|---|---|---|---|---|
| 1 | 0 | | 0 | |
| 3 | -1 (to mark blockette in use) | | 2 | |
| 5 | 0 | | 4 | |
| 7 | 0 | | 6 | |
| 11 | 0 | | 10 | |
| 13 | 0 | | 12 | |
| 15 | Group number | 255(10) | 14 | LPPN |
| 17 | "GFD" (in RAD50 format) | | 16 | LID |

**Figure 8-7:** GFD Label Blockette

| *Offset* | *Symbol* | *Contents* |
|---|---|---|
| 0 | | (unused) |
| 2 | | This word must be nonzero to mark the blockette in use. |
| 4-12 | | (unused) |
| 14 | LPPN | This word contains the value 255(10) in the low byte (indicating that this label is a GFD label) and the group number in the high byte. |
| 16 | LID | This word contains the 3-character string "GFD" in RAD50 format. |

### 8.4.2 UFD Device Cluster Numbers

Block 1 of cluster 0 of the group file directory is structured as a table consisting of one 1-word entry for each possible user account. There can be up to 255(10) users, numbered from 0 to 254, for each group.

Each table entry is indexed by the user number, multiplied by two, and contains the starting device cluster number (DCN) of the corresponding user file directory. If no user file directory has been created for a particular user number, the corresponding table entry contains a value of zero.

The 256(10)-word table is terminated with an entry of zero.

### 8.4.3 User Name Blockette Links

Block 2 of cluster 0 of the group file directory is structured as a table consisting of one 1-word entry for each possible user account. There can be up to 255(10) users, numbered from 0 to 254, for each group.

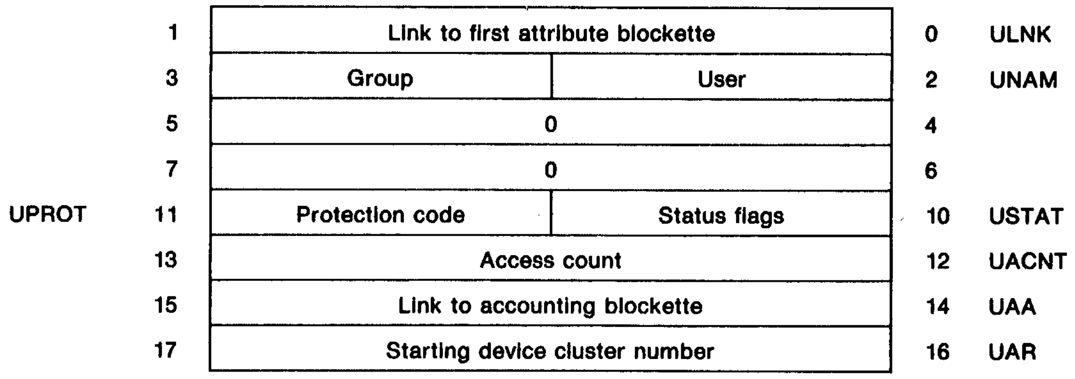
Each table entry is indexed by the user number, multiplied by two, and contains the directory link word to the user name blockette (within the GFD) for the corresponding user account. If no user file directory has been created for a particular user number, the corresponding table entry contains a value of zero.

The 256(10)-word table is terminated with an entry of zero.

### 8.4.4 User Name Blockette

One name blockette exists for each user file directory cataloged by the GFD. Name blockettes are accessed through the directory link words listed in block 2 of cluster 0 of the GFD (see Section 8.4.2) and contain information needed to identify accounts and (on the system disk) to control login access to the system. When a user file directory is accessed, the name blockette is read into memory; certain fields are used to maintain dynamic information concerning how the UFD was accessed, as well as other characteristics of the account.

Figure 8-8 shows the format and content of a user name blockette. Each field in the blockette is discussed below.



| | | | | | |
|---|---|---|---|---|---|
| | 1 | Link to first attribute blockette | | 0 | ULNK |
| | 3 | Group | User | 2 | UNAM |
| | 5 | 0 | | 4 | |
| | 7 | 0 | | 6 | |
| UPROT | 11 | Protection code | Status flags | 10 | USTAT |
| | 13 | Access count | | 12 | UACNT |
| | 15 | Link to accounting blockette | | 14 | UAA |
| | 17 | Starting device cluster number | | 16 | UAR |

**Figure 8-8:** User Name Blockette

| *Offset* | *Symbol* | *Contents* |
|---|---|---|
| 0 | ULNK | This word contains the directory link word to the first blockette in the user account attribute chain (see Section 8.1.2). Since each user account requires at least one attribute blockette (containing the account password), this link is never null. |
| 2 | UNAM | These two bytes contain the group and user number (also known as the project-programmer number, or PPN) of the account. The low byte contains the user (or programmer) number, and the high byte contains the group (or project) number. |
| 4-6 | | (unused) |
| 10 | USTAT | This byte contains the status flags for the account. The flag definitions are consistent with those defined for individual files within a UFD, and therefore, some are not relevant to the UFD. The setting of these bits pertinent to a user name blockette is as follows: |

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | US.OUT | This bit is of historical significance only. It is always zero in a user name blockette. |
| 1 | US.PLC | The UFD has been "placed" on the disk. |
| 2 | US.WRT | The UFD has been opened for write access. |
| 3 | US.UPD | The UFD has been opened in update mode. This bit is always zero in a user name blockette. |
| 4 | US.NOX | The UFD cannot be extended. This bit is always one in a user name blockette. |
| 5 | US.NOK | The UFD cannot be deleted or renamed. This bit is always one in a user name blockette. |
| 6 | US.UFD | This bit is of historical significance only. It is always one in a user name blockette. |
| 7 | US.DEL | The UFD has been marked for deletion. This bit is always zero in a user name blockette. |

| | | |
|---|---|---|
| 11 | UPROT | This byte contains the account protection code, used to control read and write access to the UFD. It is ignored in a user name blockette since reading a UFD as a file is a privileged operation and the normal protection mechanisms do not apply. |
| 12 | UACNT | This word contains the current access count for the account, used to determine whether or not the account is in use. It is incremented by one each time the UFD is opened as a file and decremented each time it is closed. The value is increased by 400(8) every time a user logs into the account and decreased when the user logs out. Thus, the low byte is a count of the number of I/O channels that have the UFD open as a file, and the high byte is a count of the number of logged-in users. |
| 14 | UAA | This word contains a link to the accounting blockette for this account. Since each account must have an accounting blockette, this link is never null. |
| 16 | UAR | This word contains the starting device cluster number (DCN) of the user file directory. (Note that this is the same value as found in the user entry in block 1 of cluster 0 of the GFD. See Section 8.4.2.) |

### 8.4.5 User Accounting Blockette

There is one accounting blockette for each user account cataloged by the GFD. The accounting blockette is used to store accumulated resource usage information for the account. CPU time, connect time, kilo-core-ticks, and device time counters in the accounting blockette have counterparts in the monitor's memory-resident job tables. While the job exists, the in-core counters are maintained in the job's secondary job data block, JDB2 (see Section 3.1.3). When the job logs out, or is killed, the in-core counters are added to the counters in the user accounting blockette and updated on the disk.

Figure 8-9 shows the format and contents of the user accounting blockette. Each field is discussed below.

| | | | | |
|---|---|---|---|---|
| 1 | (link) | | 0 | ULNK |
| 3 | Accumulated CPU time | | 2 | MCPU |
| 5 | Accumulated connect time | | 4 | MCON |
| 7 | Accumulated kilo-core-ticks | | 6 | MKCT |
| 11 | Accumulated device time | | 10 | MDEV |
| 13 | CUP Time (MSB) | Kilo-core-ticks (MSB) | 12 | MMSB |
| 15 | Logout disk block quota | | 14 | MDPER |
| 17 | UFD cluster size | | 16 | UCLUS |

**Figure 8-9:** User Accounting Blockette

*Offset* *Symbol* *Contents*

0 ULNK For compatibility with the file accounting blockette, this word is reserved for a link to a file attribute blockette. This link is always null in the user accounting blockette.

2 MCPU This word contains the low-order sixteen bits of the accumulated CPU time for the account. CPU time is stored in tenths of a second and represents the total amount of time that this account has been used for running jobs. The high-order six bits of this 22-bit counter are contained in bits 10 to 15 of word 12 (MMSB) of the blockette.

4 MCON This word contains the accumulated console connect time for the account. Connect time is stored in minutes and represents the total amount of time that any job has been logged into this account. (Connect time is only accumulated for jobs that are logged in.)

6 MKCT This word contains the low-order sixteen bits of the accumulated kilo-core-ticks for jobs running under this account. One kilo-core-tick is the use of one K-word of memory while executing for one-tenth of a second. Using two K-words for one-tenth of a second is two kilo-core-ticks, and so on. This unit of measure gives a more accurate picture of the use of system resources. The high-order ten bits of this 26-bit counter are contained in bits 0 to 9 of word 12 (MMSB) of the blockette.

10 MDEV This word contains the accumulated device time for jobs running under this account. Device time is measured in device-minutes, which is the use of one device (exclusive of the job's console terminal) for one minute. Using two devices for one minute is two device-minutes, and so on.

14 MDPER This word contains the total number of blocks that can be stored by this UFD when the account is logged out. This quota is specified when the account is created but can be changed using a FIP directive. The quota can have any value from 0 to 65,535(10) blocks, with a value of zero indicating an infinite quota — that is, the account can own any number of disk blocks.

16 UCLUS This word contains the UFD cluster size. The cluster size is specified when the account is created and cannot be changed without deleting the account. This value determines the approximate number and total size of all files that can be cataloged by the UFD.

## 8.5 User File Directories (UFDs)

User file directories (UFDs) are the third level of the RSTS/E directory structure. They are used to catalog files, storing file identification information, retrieval pointers, and protection and status bits that provide access to and protection of file data.

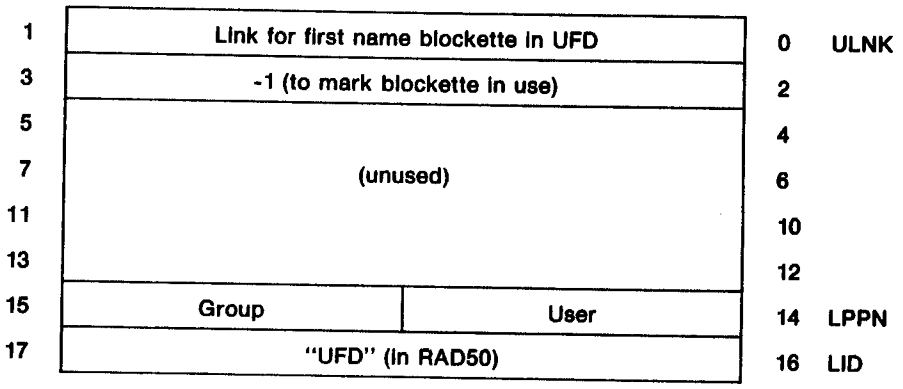
The UFD contains name, accounting and attribute blockettes similar to those found in the group file directory. In addition, the UFD contains retrieval blockettes that provide the information necessary for mapping file virtual block numbers (VBNs) to disk logical block numbers (LBNs).

The basic UFD structure is created when the user account is created with the "create user account" system directive. The UFD need not catalog any files to exist.

Once an account exists, it cannot be deleted until it is zeroed — that is, until all the files it catalogs have been deleted. When an account is zeroed, the caller can request that the file processor deallocate all UFD clusters. Even after the UFD is zeroed, however, the account itself must be explicitly deleted using the "delete user account" system directive. Deletion of an account implies deallocation of the GFD name and accounting blockettes for the account.

### 8.5.1 UFD Label Blockette

The UFD label blockette serves as the root for the list of UFD name blockettes. The label blockette is created when the UFD is created. There is one label blockette per UFD and it is always the first blockette in the UFD — blockette 0 of block 0 of cluster 0. Figure 8-10 shows the format and contents of the UFD label blockette. Following the figure is a description of each entry.



| | | | | |
|---|---|---|---|---|
| 1 | Link for first name blockette in UFD | | 0 | ULNK |
| 3 | -1 (to mark blockette in use) | | 2 | |
| 5 | (unused) | | 4 | |
| 7 | | | 6 | |
| 11 | | | 10 | |
| 13 | | | 12 | |
| 15 | Group | User | 14 | LPPN |
| 17 | "UFD" (in RAD50) | | 16 | LID |

**Figure 8-10:** UFD Label Blockette

*Offset* *Symbol* *Contents*

0 ULNK This word is the actual root of the list of file name blockettes. If there are no files in this account, this word contains a null link.

2 Since the link to the first file name blockette can be zero, this word must be nonzero to mark this blockette in use.

4-12 (unused)

14 LPPN This word contains the group and user number (also known as the project-programmer number, or PPN) of the account. It is stored here as an aid to pack reconstruction programs. The low byte contains the user (or programmer) number and the high byte contains the group (or project) number.

16 LID This word contains the characters "UFD", stored in RAD50 format. It is stored here as an aid to pack reconstruction programs.

### 8.5.2 File Name Blockette

There is one file name blockette for each file cataloged by the UFD. Each name blockette contains the basic information needed to identify the file and control access to it.

Figure 8-11 shows the format and contents of the file name blockette. Following the figure is a description of each entry.

| | | | | | |
|---|---|---|---|---|---|
| | 1 | Link to next file name blockette | | 0 | ULNK |
| | 3 | File name (in RAD50) | | 2 | UNAM |
| | 5 | | | 4 | |
| | 7 | File type (in RAD50) | | 6 | |
| UPROT | 11 | Protection code | Status flags | 10 | USTAT |
| | 13 | Read-regardls cnt | Access count | 12 | UACNT |
| | 15 | Link to file's accounting blockette | | 14 | UAA |
| | 17 | Link to file's 1st retrieval blockette | | 16 | UAR |

**Figure 8-11:** File Name Blockette

*Offset* *Symbol* *Contents*

0 ULNK File name blockettes are chained together in a singly linked list. Therefore, this word contains a link to the next blockette in the list. If this is the last file in the directory, this word will contain a null link. New files are normally added to the end of the list and hence, files are cataloged in historical order of creation — that is, the name blockette for the most recently created file will appear at the end of the list. This can be changed, however, by specifying the "new files first" (NFF) option during the disk initialization process or by using the special "beginning of directory" option when creating a file. If either of these options is chosen, the name blockettes of newly created files are linked to the front of the list.

The low-order four bits of this word are always used as flags (see Section 8.2), defined here as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | UL.USE | This bit is always set to mark the blockette in use. |
| 1 | UL.BAD | If this bit is set, the file contains a bad block. |
| 2 | UL.CHE | If this bit is set, data from the file is to be cached whenever data caching is enabled. The caching mode used is determined by bit 2 at offset UAA. |
| 3 | UL.CLN | (unused) |

2 UNAM These two words contain the name of the file, stored in RAD50 format. File names can be from 1 to 6 characters long. Since a file name is required to have at least one character, the word at offset 2 is guaranteed to be nonzero. This is important since it is possible for the link word to be zero; if the file name could also be zero, the blockette might appear to be a hole.

6 This word contains the file type, stored in RAD50 format. A file type can be 0 to 3 characters long. If there is no type, this word will contain a zero.

10 USTAT This byte contains the status flags for the file. The setting of these flags is defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | US.OUT | The data space for this file is physically on another disk. (This bit is of historical significance only. Previously, system files SWAPn.SYS, OVR.SYS, ERR.SYS, and BUFF.SYS could reside on a separate swapping disk. Swapping disks were considered a logical extension of the system disk and files residing there were included in the UFD for account [0,1] on the system disk.) |
| 1 | US.PLC | The file has been "placed" on the disk. |
| 2 | US.WRT | The file has been opened for write access. |
| 3 | US.UPD | The file has been opened in update mode. |
| 4 | US.NOX | The file was created as a contiguous file and cannot be extended. |
| 5 | US.NOK | The file cannot be deleted or renamed. This bit is provided as a protection for system files. Only the disk initialization code can delete critical system files. |
| 6 | US.UFD | This bit is always zero in a file name blockette. |
| 7 | US.DEL | The file has been marked for deletion. Files cannot be deleted until the access count at offset 12 (UACNT) is zero. If the access count is nonzero, the file processor will honor a request to delete the file by setting this bit. The file will then be deleted when the last user closes the file. |

11 UPROT This byte contains the protection code flags for the file. The setting of these flags is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | UP.RPO | The file is read-protected against its owner. If bit 14 (UP.RUN) is also set, the file is both read- and write-protected against its owner. |
| 9 | UP.WPO | The file is write-protected against its owner. If bit 14 (UP.RUN) is also set, the file is execute-protected against its owner. |
| 10 | UP.RPG | The file is read-protected against its group. If bit 14 (UP.RUN) is also set, the file is both read- and write-protected against its group. |

| | | |
|---|---|---|
| 11 | UP.WPG | The file is write-protected against its group. If bit 14 (UP.RUN) is also set, the file is execute-protected against its group. |
| 12 | UP.RPW | The file is read-protected against the world. If bit 14 (UP.RUN) is also set, the file is both read- and write-protected against the world. |
| 13 | UP.WPW | The file is write-protected against the world. If bit 14 (UP.RUN) is also set, the file is execute-protected against the world. |
| 14 | UP.RUN | The file is executable. |
| 15 | UP.PRV | The file is to be overwritten with zeros when it is deleted. If bit 14 is also set, this bit indicates that the executable file is privileged. |

12 UACNT This byte contains the file access count. It is incremented by one when the file is opened and decremented when the file is closed.

The access count controls file deletion. The file will not be deleted until access count goes to zero. If the access count is nonzero when a delete request is issued, the file is marked for deletion by setting bit 7 (US.DEL) in the status byte. The file is then deleted after the final close — when the access count goes to zero.

13 This byte contains a count of the number of users that currently have the file open in "read-regardless" mode.

14 UAA This word contains a link to the file's accounting blockette. Since each file must have an accounting blockette, this word will never contain a zero.
The low-order four bits of this word are always used as flags (see Section 8.2), defined here as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | UL.USE | (unused) |
| 1 | UL.BAD | If this bit is set, the file contains a bad block. |
| 2 | UL.CHE | If bit 2 is set at offset ULNK, this bit is used to determine the mode in which data from the file will be cached. If this bit is set, data will be cached in sequential mode; otherwise, data will be cached in random mode. (This bit has no meaning if bit 2 is clear at offset ULNK.) |
| 3 | UL.CLN | (unused) |

16 UAR This word is the root of the linked list of retrieval blockettes for this file. This link will be null only for zero length file.

### 8.5.3 File Accounting Blockette

There is one accounting blockette for each file cataloged by the UFD. The accounting blockette contains information about the file's length and last access date, as well as other static information.

Figure 8-12 shows the format and contents of the file accounting blockette. Each field is discussed below.

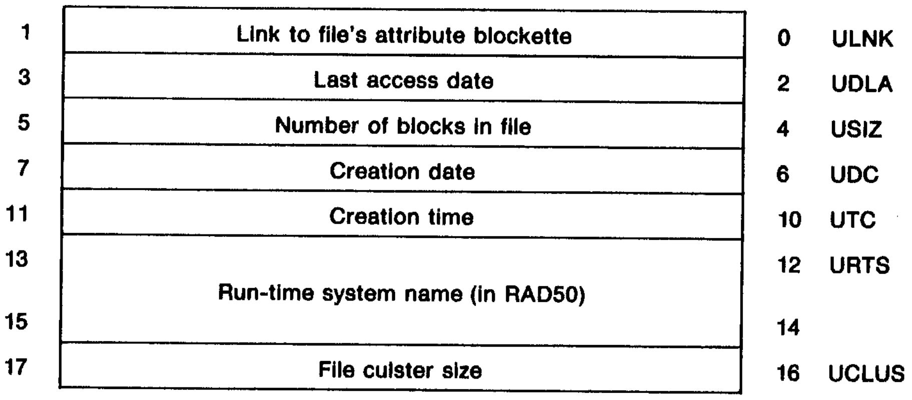


**Figure 8-12:** File Accounting Blockette

*Offset* *Symbol* *Contents*

0 ULNK This word contains the link to the file's attribute blockette (if any). Files accessed under the RMS file system are required to have attribute blockettes storing record and file attributes. If the file has such attributes, this word contains the link to the first attribute blockette. If the file does not have attributes, the link will be null.

2 UDLA This word contains the date the file was last accessed, stored in RSTS/E internal format. It is set equal to the system date whenever the file is closed. This procedure can be modified, however, by specifying the "date of last write" (DLW) option during the disk initialization process. If the DLW option is chosen, this field is only updated when the file is modified. The last access date is used by selective backup utilities to delete files that have not been used for some period of time.

4 USIZ This word contains the size of the file, in logical blocks. It is set to one when the first cluster is allocated and incremented by the file cluster size on each subsequent allocation of a cluster. During normal file creation and extension (when the file is not pre-extended), USIZ will be smaller than the actual number of blocks in the file by some number of blocks less than or equal to the file cluster size, minus one. This word is updated to reflect the actual number of blocks written either when the file is closed or when it is extended (depending on the mode in which the file was opened).

6 UDC This word contains the file creation date, stored in RSTS/E internal format.

10 UTC This word contains the file creation time, stored in RSTS/E internal format.

12 URTS If the file is an executable file, these two words contain the name of the run-time system under which the file must run. The run-time system name can be from 1 to 6 characters long and is stored in RAD50 format.

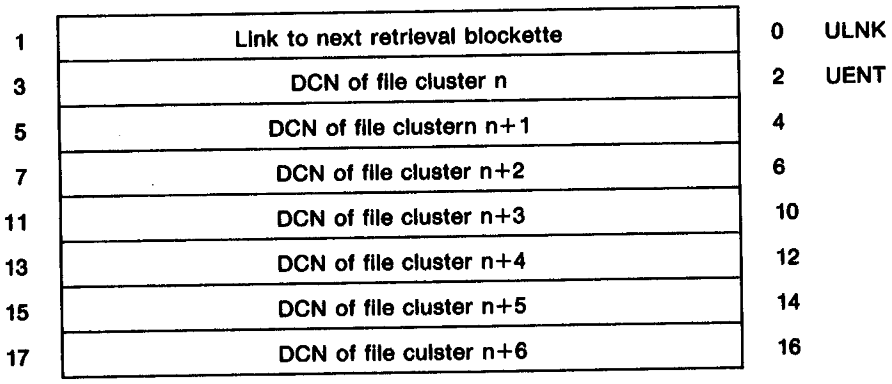
If the file is not an executable file, this two word area can be used to store the high-order eight bits of the file size for files larger than 65,535(10) blocks. In this case, the first word (offset 12) must be zero, and the low-order byte of the second word contains the most significant bits of the file's (24-bit) size. Note that this means that executable files can never be larger than 65,535 blocks long.

16 UCLUS This word contains the cluster size of the file. The file cluster size is specified when the file is created. It must be greater than or equal to the pack cluster size and cannot be changed without recreating the file.

### 8.5.4 File Retrieval Blockette

Retrieval blockettes provide the necessary information to access the blocks of the file. They store retrieval pointers, in the form of device cluster numbers (DCNs), required to map virtual block numbers (VBNs) to file processor block numbers (FBNs). FBNs are translated to physical disk addresses of file data by the disk I/O subsystem.

Figure 8-13 shows the format and contents of the file retrieval blockette. Following the figure is a description of each entry.



| | | | |
|---|---|---|---|
| 1 | Link to next retrieval blockette | 0 | ULNK |
| 3 | DCN of file cluster n | 2 | UENT |
| 5 | DCN of file clustern n+1 | 4 | |
| 7 | DCN of file cluster n+2 | 6 | |
| 11 | DCN of file cluster n+3 | 10 | |
| 13 | DCN of file cluster n+4 | 12 | |
| 15 | DCN of file cluster n+5 | 14 | |
| 17 | DCN of file culster n+6 | 16 | |

**Figure 8-13:** File Retrieval Blockette

| *Offset* | *Symbol* | *Contents* |
|---|---|---|
| 0 | ULNK | Retrieval blockettes are chained together as a singly linked list. This word contains the link to the next blockette in the list. The linked list can be as long as required to describe the file, consistent with the capacity of the UFD. If this is the last retrieval blockette for the file, this word will contain a null link. |
| 2-16 | UENT | These seven words contain the retrieval pointers. Each retrieval pointer consists of the device cluster number of one file cluster. Each retrieval blockette can store up to seven pointers. Thus, each blockette can map up to 7*FCS virtual blocks of the file. As with the directory cluster map, this set of pointers (or window) is never sparse. That is, if the DCN of file cluster n is nonzero, then the DCN of every preceding cluster is nonzero also. |

## 8.6 Minimal File Structure

When a disk is initialized to the RSTS/E file structure by DSKINT, the disk initialization code, a minimal file structure is recorded on the disk. The following items are included in that minimal structure.

- Boot Block. The first logical block on all RSTS/E disks (LBN 0) contains a bootstrap loader. A complete description of this block as found on the system disk is given in Section 8.6.1.
- Pack Label. The first logical block of device cluster 1 on all RSTS/E disks contains the pack label. Section 8.6.2 gives a complete description of the pack label.
- Master File Directory and Group File Directories. The MFD has entries for group 0 and (on the system disk) group 1. The GFDs for these groups have entries for the following user accounts:

  [0,1] — The system files account

  [1,2] — The system library account (on the system disk only)
- User File Directories. The UFDs for the above accounts are configured as follows:

  [0,1] — File entry for BADB.SYS (see Section 8.6.3) File entry for SATT.SYS (see Section 8.6.4)

  [1,2] — The system library UFD (on the system disk) is created by DSKINT but contains no file entries.
- File Data. The only files created during the disk initialization process are the storage allocation table (SATT.SYS) and the bad block file (BADB.SYS).

### 8.6.1 Boot Block

The first logical block (LBN 0) of every RSTS/E file-structured disk is the "boot" block. On the system disk, the boot block contains device-specific code (known as a "bootstrap") for loading (or "booting") the RSTS/E system initialization code (INIT.SYS). On any nonsystem disk (either public or private), the boot block contains code which prints a message informing the user that the disk is a data disk and cannot be booted.

The boot block on a system disk has a specific format (shown in Figure 8-14). The first 20(10) words comprise a standard header containing device-specific parameters, transfer addresses, and so on. The area following the header contains the actual bootstrap code, and following that is another data area defining the disk location of the INIT code to be loaded. (Each field in the boot block is discussed in detail below.)

When the system disk is booted, the hardware reads LBN 0 of the disk into memory, starting at location 0. Control then transfers to location 0 and the bootstrap code begins execution.

The first thing that the bootstrap does is to relocate itself into high memory, beginning at location 157000(8). This frees the lower portion of memory for the INIT program. It also copies the date and time from location 1000(8) into the 3-word area (at offset B.DATE) at the very top of the relocated boot block. Control then transfers to the main bootstrap code at offset B.RUN.

The bootstrap loader itself is primarily a set of routines that perform a series of functions permitting the transfer of data between the disk and memory.

NOTE

> A bootstrap can be designed to *write* to the disk as well as read from it. In fact, the RSTS/E crash dump program uses this same mechanism to create the system crash file. This section, however, is restricted to a discussion of the bootstrap on the system disk, used to load the system initialization code.

First, a device reset routine is called to prepare the device prior to the transfer operations. This routine (B.RSET) issues a controller reset function (if one is required) and does any other necessary setup. The block map is then used to set up the logical block number (B.BLKL), the memory address (B.MEML), and the word count (B.TWC ) fields of the header. A device specific I/O routine (B.READ) is then called to perform a single transfer of data between the disk and memory.

If the entire file cannot be loaded with one transfer (that is, it is not a contiguous disk file), the above procedure is repeated until all the required portions of the file have been loaded.

When the load is complete, the bootstrap jumps to the transfer instruction (offset B.JMP), control goes to the transfer address (specified at offset B.XFER), and the INIT program is started.

| | | | |
|---|---|---|---|
| | NOP instruction | | B.BOOT |
| | Branch to setup code | | |
| | 6 | | B.VE04 |
| | HALT | | |
| | 12(8) | | B.VE10 |
| | HALT | | |
| | Device cluster size | | B.DCS |
| | Device CSR base | | B.CSR |
| | Device name (in ASCII) | | B.NAME |
| | JMP @(PC)+ / MOV (PC)+,PC | | B.JMP |
| | Transfer address | | B.XFER |
| | Booted unit number | | B.UNIT |
| | Bootstrap memory address (in MMUs) | | B.MMU |
| B.WFUN | Write function | Read function | B.RFUN |
| | Primary function code | | B.FUNC |
| | Logical block number (low order) | | B.BLKL |
| | (high order) | | B.BLKH |
| | Memory address (low order) | | B.MEML |
| | (high order) | | B.MEMH |
| | Word count | | B.TWC |
| | Bootstrap mainline code | | B.RSET<br>B.READ<br>B.SPEC |
| | Relocation code | | B.MOVE |
| | Device Reset and I/O Driver Routines | | |
| | Block map | | B.MAP |
| | 0 | | B.DATE |
| | 0 | | |
| | 0 | | |

**Figure 8-14:** System Disk Boot Block

| *Offset* | *Contents* |
|---|---|
| B.BOOT | This word contains a NOP instruction. This is required by the DIGITAL standard for hardware bootstraps. |
| B.BOOT+2 | This word contains a branch to offset B.MOVE. (Note that B.MOVE is a "floating" offset that varies with the size of the bootstrap mainline code.) |
| B.VE04 | This word contains the value 6 and the following word contains a HALT instruction. Thus, if the CPU traps to location 4, processing will halt. |

B.VE10 This word contains the value 12(8) and the following word contains a HALT instruction. Thus, if the CPU traps to location 10(8), processing will halt.

B.DCS This word contains the device cluster size of the disk. Note that this is the *device* cluster size, not the pack cluster size.

B.CSR This word contains the bus address of the disk controller CSR register.

B.NAME This word contains the disk device name, stored as two ASCII bytes.

B.JMP This word initially contains the PDP-11 instruction "JMP @(PC)+", which transfers control to the location specified in the following word, B.XFER. When a save-image library (SIL) is installed (using the INSTALL option of INIT) or the hardware configuration is modified (using the HARDWR option of INIT), this word is changed to "MOV (PC)+,PC". The effect of these two instructions is the same, but the new instruction format is used as a flag to INIT to signify that installation-specific information must be processed (see Section 18.3.2).

B.XFER This word contains the address to which control will be transferred when the bootstrap load has completed. The transfer address is supplied by the HOOK program when the bootstrap is initialized on the disk.

B.UNIT Bits 0 to 2 of this word are used to store the unit number from which the device was booted. This value is set to zero when the boot block is first initialized on the disk and is updated by the bootstrap mainline code when the disk is booted.

B.MMU This word contains the memory address of the bootstrap, in memory management units (MMUs). This word is used by the RSTS crash code to execute the boot code under memory management. In the system disk boot block, this word is always zero.*

B.RFUN This byte contains the value of the CSR function code to execute a read operation.

B.WFUN This byte contains the value of the CSR function code to execute a write operation.

B.FUNC This word contains the CSR function code of the first required disk operation. In the system disk boot block, this is the read operation.

B.BLKL These two words contain the logical block number on the disk

B.BLKH at which to start the transfer.

B.MEML These two words contain the physical memory address at which

B.MEMH to start the transfer.

B.TWC This word contains the number of words to be transferred.

B.RSET The area beginning at this location contains the bootstrap mainline code. It can vary in length but the absolute combined maximum length of the mainline code, the relocation code, and the driver code is 712(8) bytes.

* Bootstraps used to write data to disk can execute in 18-bit address space using the memory management hardware. Thus, this address is necessary for mapping such code.

B.MOVE The area beginning at this offset contains the code used to relocate the entire boot block into high memory beginning at location 157000(8). Once the entire bootstrap is relocated, program control tranfers into the bootstrap general driver routine (B.DVR).

B.DVR The area beginning at this offset contains the device reset and I/O driver routines. The actual beginning and length of this area can vary, but the absolute combined maximum length of the mainline code, the relocation code, and the driver code is 712(8) bytes.

B.MAP The area between the end of the driver code and this offset contains the block map. The block map is set up in reverse order, from offset 770(8), downward. This permits the maximum amount of space for the various pieces of the bootstrap code. Each entry in the map consists of three words (reading from high to low address):

- Word count
- High-order of the logical block number
- Low-order of the logical block number

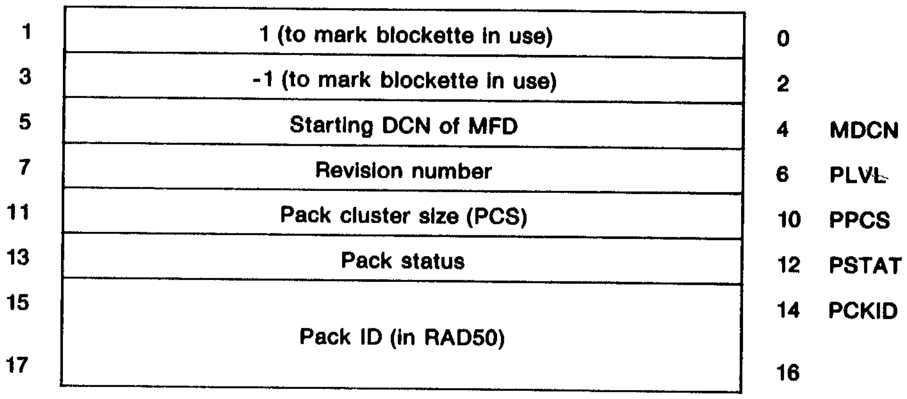
Thus, each entry describes a contiguous section of the disk. A single word of zero terminates the map.

B.DATE The three words beginning at this offset are used to store date and time information.

### 8.6.2 Disk Pack Label

The first logical block of device cluster 1 on every RSTS/E file-structured disk contains the pack label. The pack label is created when the disk is initialized using DSKINT — either as an option of the system initialization code or as an on-line utility. It contains basic information required to mount and access the pack. It also contains a pointer to the master file directory on the pack.

Figure 8-15 shows the format and contents of the disk pack label. Following the figure is a description of each entry.



| | | | |
|---|---|---|---|
| 1 | 1 (to mark blockette in use) | 0 | |
| 3 | -1 (to mark blockette in use) | 2 | |
| 5 | Starting DCN of MFD | 4 | MDCN |
| 7 | Revision number | 6 | PLVL |
| 11 | Pack cluster size (PCS) | 10 | PPCS |
| 13 | Pack status | 12 | PSTAT |
| 15 | Pack ID (in RAD50) | 14 | PCKID |
| 17 | | 16 | |

**Figure 8-15:** Disk Pack Label

| *Offset* | *Symbol* | *Contents* |
|---|---|---|
| 0-2 | | (Unused but must be zero) |
| 4 | MDCN | This word contains the device cluster number of the start of the master file directory on this pack. When the pack is mounted, the file processor stores this value in memory for subsequent use. |
| 6 | PLVL | This word contains the revision level of the directory structure on this pack. The high byte contains the major version number and the low byte contains the minor version number. The major version number is changed each time a change in directory structure is made that is not upwardly compatible with the previous version. The minor version number is set to one each time the major version number changes and is incremented each time an upwardly compatible structure change is made. Only disks containing the latest revision level can function as system disks. |
| 10 | PPCS | This word contains the pack cluster size (PCS) as defined in Section 7.3.2. The PCS is specified during the disk initialization process and cannot be changed. |
| 12 | PSTAT | This word contains the pack status bits. The setting of these bits is as follows: |

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-8 | | (unused) |
| 9 | UC.TOP | This bit is the "new files first" (NFF) bit. If the NFF option is chosen during the disk initialization process, newly created files will be linked to the beginning of their UFDs, rather than at the end. |
| 10 | | (unused) |
| 11 | UC.DLW | This bit is the "date of last write" (DLW) bit. If the DLW option is chosen during the disk initialization process, the last access date field (UDLA) of file accounting blockettes is only updated when a file is written to. Normal procedure is to update the UDLA field any time a file is closed. |
| 12 | UC.RO | If this bit is set, the pack is a read-only pack. |
| 13 | UC.NEW | If this bit is set, the pack was initialized to a file structure supported by RSTS/E V8.0 or later. (That is, the word at offset 6 contains a valid revision level.) |

14 UC.PRI This is the public/private bit. A pack is declared public or private during the disk initialization process. If this bit is set, the pack is private; otherwise it is public. Note that the system disk is always created as a private disk. When it is used as a system disk, it is treated as public, but when mounted on another system, it is treated as private. Only public, nonsystem disk packs have this bit clear.

15 UC.MNT This bit is set when the pack is mounted and cleared when it is dismounted, providing a certain level of pack protection. If an unmounted disk has bit 15 set, it was not properly dismounted and should be rebuilt (see Section 18.4.5). Either the system crashed, or the pack was physically dismounted before it was logically dismounted.

14 PCKID These two words contain the pack identification. The pack ID is specified during the disk initialization process and is used when the pack is mounted to verify that the correct pack has, in fact, been mounted. The pack ID can also be used as a logical device name. Thus, the actual physical unit need not be known by applications code. The pack ID can be up to six characters long and is stored in two words, in RAD50 format.

### 8.6.3 Bad Block File

Most large disk packs contain a number of blocks that cannot be read for one reason or another. RSTS/E keeps track of these blocks in the bad block file (BADB.SYS), stored in user account [0,1].

There are three ways in which bad blocks are detected:

1. The pack manufacturer verifies each pack and records bad blocks before each pack is shipped.
2. The disk initialization code, DSKINT, performs pattern tests that write and then verify a series of bit patterns to detect bad blocks.
3. Bad blocks are occasionally detected during normal use.

Since the minimum unit of disk storage allocation is the pack cluster (PC), any PC which contains a bad block is allocated to the bad block file. Once a bad block is allocated to BADB.SYS, it cannot be deallocated.

BADB.SYS has a file cluster size equal to the pack cluster size and is a standard, noncontiguous file. It is treated as a normal file in every respect except that the file "data" is of no consequence. The device cluster numbers contained in the file's retrieval blockettes serve as the list of clusters containing bad blocks. The file has the "no delete" bit (US.NOK) set in its status byte and cannot be deleted during normal timesharing.

### 8.6.4 Storage Allocation Table

The storage allocation table (SAT) is used to control allocation of space on a disk. It is cataloged in user account [0,1] as file SATT.SYS, and is created by the disk initialization code, DSKINT. The file is structured as a bit map, with each bit representing a pack cluster. A set bit indicates that the corresponding pack cluster is allocated. Initially, the SAT reflects only the space allocated to the minimal MFD/GFD structure, the UFD for user account [0,1], any bad blocks detected by DSKINT, and the space allocated to the SAT itself.

Since the SAT controls allocation for the disk on which it resides, its size depends on the size of the disk and the pack cluster. However, since there are 256(10) words or 4096(10) bits per SAT block, and the device clustering scheme described in Section 7.3.1 ensures that there are never more than 65,535(10) pack clusters on a pack, the SAT is guaranteed never to be longer than sixteen blocks.

Bits within the storage allocation table are numbered from zero to the maximum pack cluster number (MPCN), as defined in Section 7.3.2. When broken into fields as shown in Figure 8-16, the pack cluster number (PCN) can be used to access the SAT directly:

- Bits 12 to 15 define the virtual block number within the file (from 0 to 15).
- Bits 3 to 11 define the byte offset within the virtual block (from 0 to 777(8)).
- Bits 0 to 2 define the bit within the the byte (from 0 to 7).

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block | | | | Byte | | | | | | | | | Bit | | |

**Figure 8-16:** Splitting the Pack Cluster Number into Fields for Use in Accessing the Storage Allocation Table

SATT.SYS is a contiguous file and its base FIP block number (FBN) is stored in memory resident monitor tables when the pack is mounted. This permits the file processor routines concerned with the allocation of disk space to access the complete bit map without reference to directory information. Although the file is contiguous, it does have retrieval blockettes for file structure consistency.

# CHAPTER 9
# The File Processor [V8.0]

The file processor (FIP) is one of the major components of the RSTS/E monitor. It is responsible for all operations that access and/or affect the on-disk directory structures: opening and closing files for access, creating and deleting files and user accounts, and so on. Less obviously, FIP also handles many system functions not directly associated with file activity, such as adding and deleting Concise Command Language (CCL) commands, locking jobs in memory, and opening and closing I/O channels for non-file-structured devices.

NOTE

> A commonly held misconception is that the file processor handles file I/O. This is wrong. The file processor deals with only the on-disk directory structure. Reading and writing of file data is done by the disk I/O subsystem (see Chapter 10).

Both user jobs and monitor processes can request services from the file processor.

This chapter discusses FIP's in-core data structures (Section 9.1) and presents a general overview of its structure and processing (Section 9.2).

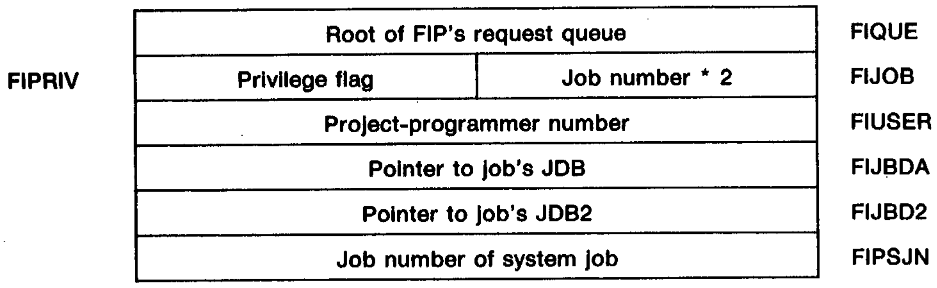
## 9.1 Data Structures

As one of the major RSTS/E monitor components, the file processor maintains many different types of data and information. This section discusses the most important of FIP's data structures.

### 9.1.1 FIP Control Area

The file processor owns a control area in which it maintains various pieces of information regarding the job for which it is currently operating. This control area resides in the low-core portion of the RSTS phase of the monitor.

Figure 9-1 shows the format of the FIP control area. Following the figure is a description of each entry.



| | | | |
|---|---|---|---|
| | Root of FIP's request queue | | FIQUE |
| FIPRIV | Privilege flag | Job number * 2 | FIJOB |
| | Project-programmer number | | FIUSER |
| | Pointer to job's JDB | | FIJBDA |
| | Pointer to job's JDB2 | | FIJBD2 |
| | Job number of system job | | FIPSJN |

**Figure 9-1:** FIP Control Area

*Location* *Contents*

FIQUE This word contains the root of the FIP request queue. It points to the first file processor request (see Section 9.1.2). If there are currently no requests to be processed, this word contains a zero.

FIJOB This byte contains the job number, multiplied by two, of the user job for which FIP is currently processing a request.

FIPRIV This byte contains a flag indicating the privilege level of the job specified in location FIJOB. If the value in this byte is zero, the job is privileged; if the value is nonzero, the job is nonprivileged.

FIUSER This word contains the account number under which the job specified in FIJOB is logged into the system. The low byte contains the programmer (or user) number and the high byte contains the project (or group) number.

FIJBDA This word contains a pointer to the job data block (JDB) for the user job specified in FIJOB.

FIJBD2 This word contains a pointer to the secondary job data block (JDB2) for the user specified in FIJOB.

FIPSJN If the current FIP request was not issued by a user but is instead from a portion of the monitor known as a "system job," this word contains the number of the appropriate system job. In this case, the values in locations FIJBDA and FIJBD2 contain the addresses of dummy job data blocks that permit the system job to emulate a user job. (See Section 9.2.1.)

### 9.1.2 FIP Request Queue (FIQUE)

The file processor receives the majority of its requests through a queuing mechanism. Individual requests are linked into a queue pointed to by location FIQUE in the FIP control area (see Section 9.1.1). Requests are then processed on a first in, first out basis.

A "request" in FIQUE consists of a job work block (WRK), containing a copy of the job's file request queue block (FIRQB). The FIRQB contains the requested function code, along with any appropriate parameters. (Copying the FIRQB into the work block permits the job to be swapped out while the request is being processed.) The work blocks are queued in FIQUE using the first word as a link word. An entry is not removed from the queue until all function processing complete. Thus, the first entry in the queue is always for the request currently being serviced.

The last request in the queue has a link word of zero.

### 9.1.3 FIP Block Sub-block (FBB)

Throughout its processing, the file processor needs to maintain retrieval information for various pieces of disk data. For consistency, FIP stores this type of retrieval information in a standard construct known as a FIP block sub-block, or FBB. Every 2-word FBB provides the necessary data to retrieve a block from any disk on the system with a single disk access.

There are two types of FBB: one for retrieving file data and the other for retrieving directory data. These two types differ only in the content of the first byte.

Figure 9-2 shows the format of an FBB. Following the figure is a description of each entry.

| | | | |
|---|---|---|---|
| F.FBNM | FIP block number (MSB) | Unit / offset | F.UNT |
| | FIP block number (LSB) | | F.FBNL |

**Figure 9-2:** FIP Block Sub-Block

*Offset* *Contents*

F.UNT In a data FBB, this byte contains the FIP unit number (or FUN, see Section 7.5) of the disk on which the block of data is located. In a directory FBB, this byte contains the offset, divided by two, of the pertinent directory blockette.

F.FBNM This byte contains the high-order eight bits of the FIP block number (FBN, see Section 7.4) of the block of data. The low-order sixteen bits are at offset F.FBNL.

F.FBNL This byte contains the low-order sixteen bits of the FIP block number (FBN, see Section 7.4) of the block of data. The high-order eight bits are at offset F.FBNM.

### 9.1.4 Internal Work Structures

Throughout its processing, the file processor uses many internal work buffers and data items. This section discusses the more important of these structures.

#### 9.1.4.1 Directory Buffer (FIBUF) -

While it is processing a function that requires access to the on-disk file structure, FIP keeps a copy of the current directory block in the 256(10)-word buffer, FIBUF. Thus, since FIP processes file-related functions in a serial fashion, the necessary directory information is (theoretically) in memory for the duration of the process.*

In conjunction with the directory block buffer, FIP maintains three data items containing status information concerning the buffer:

- Location FIBFBN contains the FIP block number of the directory block currently stored in FIBUF.
- Location FIBUNT contains the FIP unit number of the disk from which the directory block in FIBUF was read.
- Location FIBMOD contains a flag indicating whether the contents of FIBUF have been modified and, therefore, must be rewritten to the disk when processing is complete.

#### 9.1.4.2 Disk SAT Buffer (SATBUF) -

In addition to the directory block stored in FIBUF (see Section 9.1.4.2), the file processor keeps a copy of the currently relevant block of the disk's storage allocation table in the 256(10)-word buffer, SATBUF. FIP also maintains three data items containing status information concerning this buffer:

* In reality, this is only true for certain unfragmented directories. It does ensure, however, that the directory cluster map is in memory throughout processing of the function.

- Location SATFBN contains the FIP block number of the block of data currently stored in SATBUF.
- Location SATUNT contains the FIP unit number of the disk from which the block in SATBUF was read.
- Location SATMOD contains a flag indicating whether the contents of SATBUF have been modified and, therefore, must be rewritten to the disk when processing is complete.

#### 9.1.4.3 Miscellaneous Data Items -

Other important data items maintained by the file processor for quick reference are as follows:

- Location SYSUNT contains the FIP unit number (or FUN) of the system disk.
- Table MFDPTR consists of one 1-word entry for every disk configured on the system. Each entry contains the starting device cluster number of the master file directory on the corresponding disk. If a disk is not logically mounted, its MFDPTR entry will contain a zero.
- Location FIPFUN contains the function number of the request that FIP is currently processing.

### 9.1.5 Overlay Structures

To reduce the overall size of the resident monitor code, portions of the file processor frequently reside on disk and are brought into memory only when they are needed for execution. FIP uses a number of internal structures when processing these overlays. In addition to a 256(10)-word buffer to hold disk-resident overlays while they are in memory, FIP also uses a set of tables that indicate which service routines are disk-resident and which are memory-resident. These tables also contain information concerning the physical addresses of routines located in memory, along with disk-retrieval information for those routines that reside on disk.

#### 9.1.5.1 Overlay Buffer (OVRBUF) -

When a disk-resident service routine is required to process a particular FIP function, it is read into memory (one disk block at a time) into the FIP overlay buffer (OVRBUF). This buffer is 256(10) words long. It is allocated by the system initialization code (INIT.SYS) during system startup and is mapped, as needed, using APR 5.

In conjunction with this buffer, FIP maintains location FIPVBN which contains the virtual block number (within the monitor save-image library) of the disk-resident routine that is loaded into OVRBUF.

#### 9.1.5.2 Overlay Virtual Addresses (FIPTBL) -

Table FIPTBL contains the virtual disk addresses (within the monitor save-image library) of all FIP function service routines. The table consists of one 1-word entry for every defined FIP function code. It is filled in when the monitor SIL is linked during system generation and is the same for all systems, irrespective of which service routines are made memory-resident.

Service routine overlays are restricted to one logical block. They can, of course, be shorter. If so, one logical block can contain two or more overlays, as long as no overlay crosses a block boundary. For this reason, each 1-word FIPTBL entry can be thought of as containing two fields: The high-order seven bits contain the virtual block number (VBN) at which the corresponding service routine is located; the low-order nine bits contain the offset (within the block) at which the routine starts. (See Figure 9-3.)

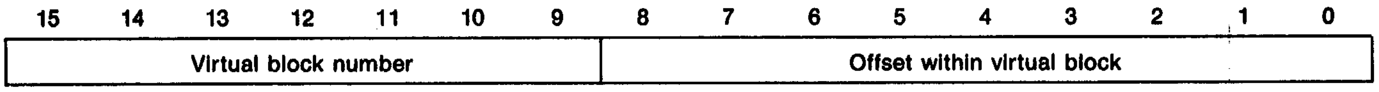


**Figure 9-3:** FIP Overlay Virtual Addresses

### 9.1.5.3 Overlay Actual Addresses (OVBASE) -

Table OVBASE contains the actual addresses of all FIP overlays — both disk- and memory-resident. It contains one 2-word entry for every logical block (within the monitor save-image library) containing FIP service routines. It is created during system generation and initialization when the FIP overlays are designated as either disk- or memory-resident.*

Figure 9-4 shows the general format of each 2-word entry. The high-order bit of the first word is used as a flag to indicate whether the block is disk- or memory-resident:

- If the bit is set, the logical block is memory-resident. The low-order fifteen bits of the first word contain the block's virtual memory address. The second word contains the block's physical memory address, in memory management units (MMUs) — that is, the value to be used to map the block with kernel APR 5.
- If the bit is not set, the logical block is disk-resident and the entry is a FIP block subblock (FBB, see Section 9.1.3), specifying the disk address of the logical block.

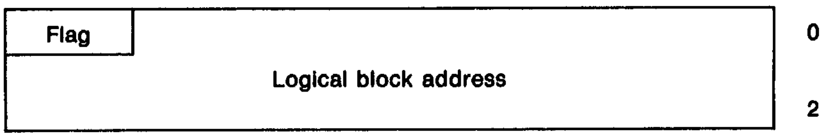


**Figure 9-4:** FIP Overlay Actual Addresses

## 9.1.6 File-Related Structures

When a disk file is opened for access, the file processor generates certain data structures that are essential for later I/O operations on the file. There are two such structures: the file control block (FCB) and the window control block (WCB).

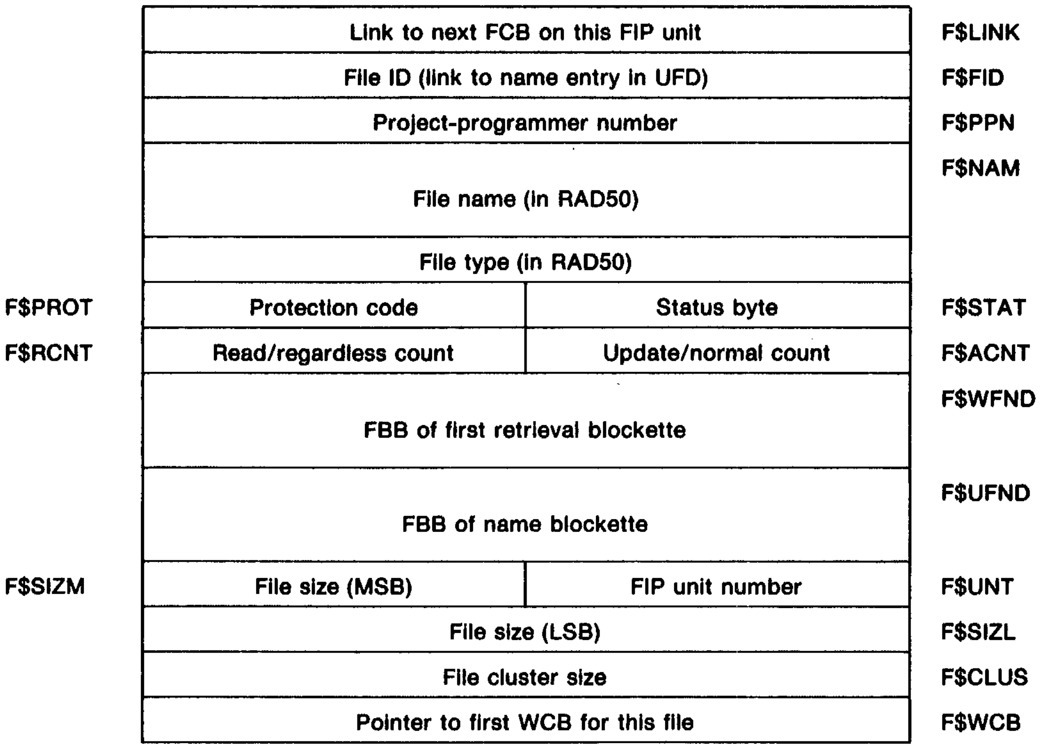
* Note that this table is on a block-by-block basis. Therefore, service routines that reside in the same logical disk block must also share the same residency status — they must all be disk-resident or they must all be memory-resident.

### 9.1.6.1 File Control Block (FCB) -

There is one file control block for every open file. The FCB is allocated from the FIP small buffer pool when the file is first opened and contains all the UFD information necessary to manipulate the file, except for the actual file retrieval pointers. (Retrieval information is kept in the window control block (WCB), discussed in Section 9.1.6.2.) When the same file is opened on another channel, or by another user, all the necessary file information is already resident in memory and no directory search is needed. In fact, successive opens use the same actual file control block. Thus, there is no significant overhead involved in opening a file for additional users.

The file control blocks for all currently opened files are kept in a linked list. Each disk unit has its own list. The base address of each list is maintained in a table (FCBLST), accessed by the FIP unit number (FUN) of the disk unit, multiplied by two.

Figure 9-5 show the format and content of the file control block. Each field is discussed below.



| | | | |
|---|---|---|---|
| | Link to next FCB on this FIP unit | | F$LINK |
| | File ID (link to name entry in UFD) | | F$FID |
| | Project-programmer number | | F$PPN |
| | File name (in RAD50) | | F$NAM |
| | File type (in RAD50) | | |
| F$PROT | Protection code | Status byte | F$STAT |
| F$RCNT | Read/regardless count | Update/normal count | F$ACNT |
| | FBB of first retrieval blockette | | F$WFND |
| | FBB of name blockette | | F$UFND |
| F$SIZM | File size (MSB) | FIP unit number | F$UNT |
| | File size (LSB) | | F$SIZL |
| | File cluster size | | F$CLUS |
| | Pointer to first WCB for this file | | F$WCB |

**Figure 9-5:** File Control Block

*Offset* | *Contents*

F$LINK — This word contains a pointer to the next file control block on this FIP unit. If this is the last FCB in the list, this word will contain a zero.

F$FID — This word contains the file identifier for this file. It is significant mainly in that it can be used in place of the file name for subsequent opens. There is no performance gain in using this file ID rather than the file name. It is provided for compatibility with RSX.

F$PPN This word contains the project-programmer number (PPN) of the account in which the file resides. The low byte contains the programmer (or user) number, and the high byte contains the project (or group) number.

F$NAM This 3-word area contains the file name and type, in RAD50 format.

F$STAT This byte contains the file's status flags as found in the file name blockette in the appropriate UFD (see Section 8.5.2).

F$PROT This byte contains the file's protection code as found in the file name blockette in the appropriate UFD (see Section 8.5.2).

F$ACNT This byte contains a count of the number of users that currently have this file open for normal or update access.

NOTE

The various modes of access possible on opening a file are discussed in the *RSTS/E Programming Manual.*

F$RCNT This byte contains a count of the number of users that currently have this file open for "read-regardless" access.

F$WFND This 2-word area is the FIP block sub-block (FBB) of the file's first retrieval blockette (see Section 9.1.3).

F$UFND This 2-word area is the FIP block sub-block (FBB) of the file's name blockette (see Section 9.1.3).

F$UNT This byte contains the FIP unit number of the disk unit on which the file resides.

F$SIZM This byte contains the high-order eight bits of the file size.

F$SIZL This word contains the low-order sixteen bits of the file size (in blocks). The high-order eight bits of this 24-bit value are found at offset F$SIZM.

F$CLUS This word contains the file's cluster size.

F$WCB This word contains a pointer to the first window control block (WCB) for this file (see Section 9.1.6.2).

#### 9.1.6.2 Window Control Block (WCB) -

The window control block (WCB) contains the file retrieval information for an individual user of an open file. A window control block is allocated from the FIP small buffer pool for each simultaneous opening of a file. The job I/O block (IOB) of each user points to the WCB.

The window control blocks are kept in a linked list pointed to by the file control block. The addition of a new user is accomplished by linking a new WCB to the end of the list.

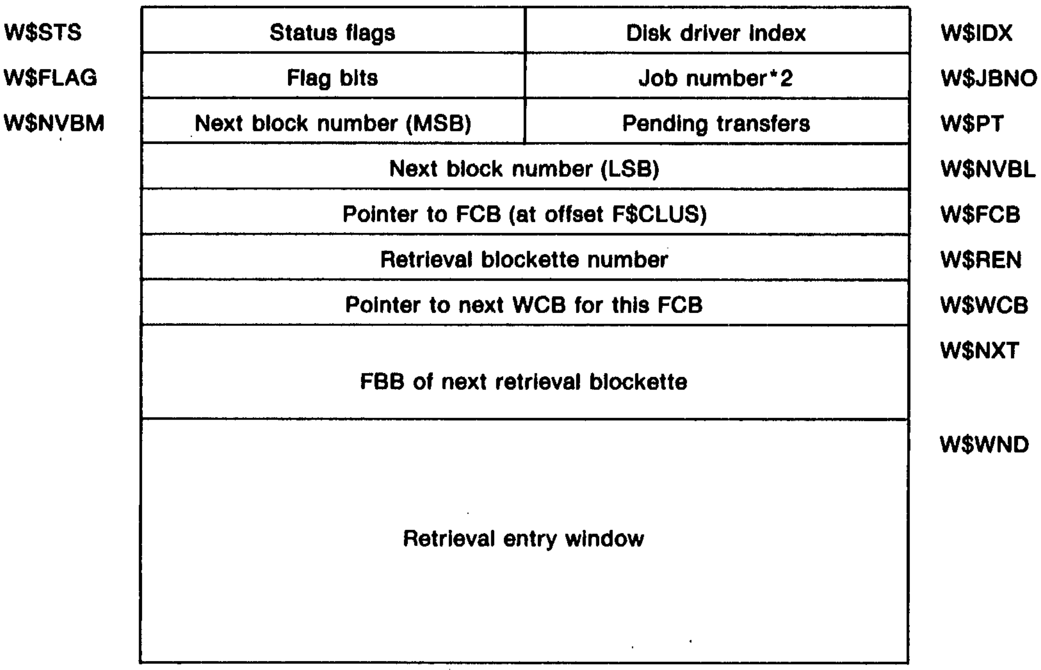
Figure 9-6 shows the format and content of the window control block. Each field is discussed below.

*Offset* *Contents*

W$IDX This byte contains the driver index. It is always zero, indicating that the device associated with the WCB is a disk device.

W$STS This byte contains the file status bits for this user. The setting of these bits is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | DDNFS | The disk was opened as a non-file-structured device. The remaining bits apply as if the disk was opened as a file-structured device. |
| 9 | DDRLO | The file is read-protected against the user. |
| 10 | DDWLO | The file is write-protected against the user. |
| 11 | WC$UPD | The file is opened for update. |
| 12 | WC$CTG | The file is contiguous. |
| 13 | WC$LCK | The current block of the file is implicitly locked (see field W$FLAG, below). |
| 14 | WC$UFD | The file is a user file directory, opened in non-file-structured mode. |
| 15 | WC$USE | This user received the original write privileges to the file. |



| | | | |
|---|---|---|---|
| W$STS | Status flags | Disk driver index | W$IDX |
| W$FLAG | Flag bits | Job number*2 | W$JBNO |
| W$NVBM | Next block number (MSB) | Pending transfers | W$PT |
| | Next block number (LSB) | | W$NVBL |
| | Pointer to FCB (at offset F$CLUS) | | W$FCB |
| | Retrieval blockette number | | W$REN |
| | Pointer to next WCB for this FCB | | W$WCB |
| | FBB of next retrieval blockette | | W$NXT |
| | Retrieval entry window | | W$WND |

**Figure 9-6:** Window Control Block

W$JBNO This byte contains the job number (multiplied by two) of the file user that owns this WCB.

W$FLAG This byte contains the file status flags and locked block information for this user. It serves two purposes. First, it defines the blocks that are included in either an implicit or an explicit lock.* Second, it contains additional status flags similar to those in W$STS. The format of this byte is as follows:

*Bit* *Symbol* *Meaning*

0-4 WC$LLK These five bits specify the number of blocks that are included in the current implicit lock. The value can range from 0 to 31(10).

5 WC$EXT If this bit is set, the WCB is followed immediately in memory by another small buffer's worth of information about explicitly locked blocks. This extended portion of the WCB consists of a table of 2-word entries. There can be up to seven entries, each defining a range of blocks that are explicitly locked. The table is terminated by a word containing a value of zero. The format of each 2-word entry is as follows:

| *Offset* | *Contents* |
|---|---|
| 0 | This byte contains the high-order eight bits of the block number of the first explicitly locked block. |
| 1 | This byte contains the number of blocks included in the lock. |
| 2 | This word contains the low-order sixteen bits of the block number of the first explicitly locked block. |

6 WC$DLW If this bit is set, the file has either been written to or extended. The file size and last access date need to be updated in the UFD when the file is closed.

7 WC$NFC If this bit is set, the file is opened as non-file-structured in cluster mode.

W$PT This byte contains a count of the number of transfers pending on the file for this user. When an I/O operation is requested, this byte is set to a value of one. If the request requires more than one physical transfer, the value is increased as necessary. (An I/O request can require more than one tranfer if it crosses a cluster boundary.)

W$NVBM This byte contains the high-order eight bits of the next block number to use for sequential access.

W$NVBL This word contains the low-order sixteen bits of the next block number to use for sequential access.

W$FCB This word contains a pointer to the file control block for the file (at offset F$CLUS).

W$REN This word contains the number of the current retrieval blockette, identifying the retrieval window currently stored at offset W$WND.

* Under RSTS/E, a file can be accessed simultaneously by multiple users. The file processor uses an implicit interlock mechanism that prevents different users from writing to the same block at the same time. In addition, user jobs can request an explicit lock of a range of blocks within the files they have open. See the RSTS/E Programming Manual for details.

W$WCB This word contains the pointer to the next WCB in the linked list for this file control block. Since WCBs are allocated from small buffers (which always fall on even 32(10)-byte boundaries) the low-order five bits of this pointer are always zero. Thus, they are used here as extended flags that are masked off before the word is used as a pointer. These extended flags are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | WC$RR | If this bit is set, the file is open in read-regardless mode. |
| 1 | WC$SPU | If this bit is set, the file is open in the special update mode. |
| 2 | WC$AEX | If this bit is set, the file size and retrieval pointers must be updated in the UFD immediately whenever the size of the file changes. If the bit is clear, the UFD information is updated only when an actual extend takes place — that is, when the extend amount exceeds the free space (if any) in the last cluster of the file. In all cases, the directory information is updated when the file is closed. |
| 3 | WC$CHE | If this bit is set, the file is open for user data caching. The type of caching is determined by bit 4 (WC$CSQ). |
| 4 | WC$CSQ | When bit 3 (WC$CHE) is set, this bit defines the type of data caching to be done on this file: If the bit is set, the file will be cached sequentially; if it is clear, the file will be cached randomly. (This bit has no meaning if bit 3 is not set.) |

W$NXT This 2-word area contains the FIP block sub-block (FBB, see Section 9.1.3) of the file's next UFD retrieval blockette. It permits immediate access to the next retrieval blockette if the file is being accessed sequentially.

W$WND This 7-word area contains the current retrieval window for this user, specifying the device cluster number of the first block of each cluster in this window.

## 9.2 Overview of Processing

The file processor is a separate monitor phase that generally runs asynchronously from the rest of the RSTS/E monitor. It maintains its own data base and its own processor stack. As part of the monitor, FIP runs at processor priority level three.

Both user jobs and monitor processes can request service from the file processor.

NOTE

It is beyond the scope of this document to provide a step by step description of each file processor function. However, certain general functions are significant.

### 9.2.1 FIP Interface

File processor functions are requested by user jobs through the standard EMT mechanism (described in Chapter 5) using EMT 0. Request parameters and return data are passed in the user job's file request queue block (FIRQB). When a user request is issued, the FIP trap handler validates the request parameters and then copies the FIRQB into the job's work block (WRK).

At this point, actual entry into the file processor depends on whether the requested function is synchronous or asynchronous (see Section 9.2.2). A synchronous file processor function is one that requires no I/O for completion. Note that this implicitly means that the required service routine must be memory-resident.

When the requested function has completed, the job is made runnable and control returns to the EMT handler, which invokes the scheduler through the level three queue mechanism.

Portions of the monitor can also issue requests to the file processor through the "system job" mechanism. Using this mechanism, a monitor component can emulate a user job — complete with primary and secondary job data blocks and a work block loaded with a "copy" of a FIRQB. A system job is flagged by an odd value at offset FQJOB in the work block. Remember that this byte ordinarily contains the job number, *multiplied by two,* and is generally an even number for user jobs.

Entry to FIP from a monitor process does not go through the EMT handler, but is accomplished with a direct transfer of control to the appropriate file processor module, determined by whether the function is synchronous or asynchronous.

When FIP detects a request from a system job, it uses the value (minus one) in FQJOB as an index into table SJBTBL containing a list of job data block addresses for all known system jobs. Once this address is obtained, processing continues normally. When the requested function is complete, FIP uses the value (minus one) in FQJOB as an index into table SJBL3Q to locate the L3QUE bit it must set to restart the proper monitor component.

### 9.2.2 General Processing

There are two different ways to enter the file processor, depending on whether the required function is asynchronous or synchronous. As mentioned in Section 9.2.1, a synchronous function is one that requires no I/O for completion. In general, only the local MESAG functions and some of the UUO subfunctions are synchronous.

To determine whether a particular UUO subfunction is synchronous or asynchronous, the UUO dispatch routine uses the subfunction code to index into table UUOSNC. This table contains one 1-byte entry for each UUO subfunction. If the entry is nonzero, the subfunction is synchronous; otherwise, it is asynchronous.

In addition to a function's inherent processing (that is, whether or not it must do I/O to complete the request), a function is not considered synchronous unless its service routine is also currently memory-resident (see Section 9.2.3). If the routine is not currently memory-resident, the function is processed as if it were asynchronous.

### 9.2.2.1 Asynchronous Processing -

If the requested FIP function is asynchronous, control is transferred to the file processor through the "standard" FIP entry — a subroutine call to routine FIPSYS. Note that this routine is located in the monitor's permanently mapped root (not with the rest of FIP) so that requests can be queued to the file processor from other portions of the monitor.

FIPSYS enters the caller's work block in the file processor's queue (FIQUE). Bit JSFIP is then set in the job's JBWAIT entry, stalling the job pending completion of the requested function. At this point, the job may also become eligible to be swapped out.

If there are no other requests in FIQUE, the file processor is not currently processing a request and control is transferred to FIP's startup entry point (FIPGO). FIPGO is responsible for ensuring that the required service routine is in memory and then dispatching to it.

If processing of the requested function requires FIP to suspend execution at any point (to do a disk access, for example), control passes to the level three return code at RTI3. Execution is later resumed through the standard level three queue mechanism used for scheduling monitor components.

When the requested function has finished, control is transferred to routine FIEXIT. This routine loads the function completion status into the first word (offset FIRQB) of the work block and "posts" any other required data (see Section 5.1.3). The routine then sets the JSFIP bit in the job's JBSTAT entry, indicating the job is now runnable, and sets the level three queue bit for the job scheduler.

After this, FIEXIT checks the condition of both FIBUF and SATBUF to determine whether the contents of either buffer has been modified. If so, the modified buffer is written back to disk.

At this point, the work block for the request just completed is removed from FIQUE. If more requests are pending in the queue, control transfers back to entry FIPGO and the next entry is processed immediately. When FIQUE is finally empty, processing terminates through the level three return code at RTI3.

Note that asynchronous processing within FIP is serial in nature. That is, it processes only one request at a time, and each request must come to a logical conclusion before the next is started. Since the RSTS/E user file directory structure is a linked list, this sequential processing provides the security necessary for multiuser file access. However, it can also result in sluggish system repsonse if one or more users continually request functions, such as wildcard directory lookups, that require multiple disk accesses to complete.

### 9.2.2.2 Synchronous Processing -

If the requested function is synchronous and the required service routine is already resident in memory, control is transferred to the file processor through what is referred to as the "back door" entry. In this case, a subroutine call is made directly to the resident service routine. If the request is from a user, there is no need to stall the job because the request is processed immediately, making the job runnable again.

When processing is complete, the service routine exits to the point at which it was called. If the request was from a user job, this return point will be the EMT trap handler, which then exits through the level three return code at RTI3.

### 9.2.3 Overlay Processing

As mentioned in Chapter 2, the file processor is structured much like the overall monitor, with a permanently mapped root and dynamically mapped overlays. The root (mapped with kernel APR 6) contains the main FIP control module (CTL), along with the FIP small buffer pool. The overlays (mapped with kernel APR 5) consist of the individual service routines needed to process the various user requests. At system generation time, any number of these service routines can be optionally specified as disk-resident.

When FIP receives a request, it uses the subfunction code found at offset FQFUN of the requestor's work block to index into table FIPTBL (see Section 9.1.5.2). The entry in FIPTBL supplies the number of the virtual block (within the monitor SIL) in which the service routine is located, along with the offset (within the block) to the start of the routine. Using the virtual block number as an index, FIP then examines table OVBASE (see Section 9.1.5.3) to determine whether the logical block (and, therefore, the service routine) is memory-resident and to obtain its actual address:

- If the logical block is memory-resident, the associated OVBASE entry contains both its virtual memory address and its physical memory address (in memory management units, or MMUs).
- If the logical block is disk-resident, the associated OVBASE entry contains the disk retrieval information necessary to locate the block on disk and read it into the overlay buffer (OVRBUF).

If the desired block is designated disk-resident, FIP will check to determine whether it is already in memory (in the overlay buffer, OVRBUF). If it is not, it is read from the disk.

After the logical block has been located and brought into memory (if necessary), FIP maps kernel APR 5 to the start of the block — either to the start of OVRBUF or to the physical address specified in OVBASE. Control is then transferred to location 120xxx(8), where "xxx" is the offset to the service routine as found in FIPTBL.

Disk-resident overlays are always one disk block (256(10) words) long. Note, however, that any of the file processor service routines can be specified as disk-resident. Thus, all overlays, memory-resident as well as disk-resident, are restricted to this size. If a particular function requires more than 256 words, it must be broken into smaller units that can be chained together.

### 9.2.4 Directory Searches

At many stages of processing, FIP must locate the UFD entry for a particular file.

Three items are necessary to specify a file completely: a device designator, an account number, and a file name (including file type) or file identifier. Since duplicate file names can appear in different user file directories, the search must be narrowed to exactly one UFD. This is the purpose of the device designator and account number. Even then, however, FIP must locate the specified UFD before it can search for the file.

In some circumstances, the device cluster number of the start of the user file directory is already available in memory.

- The starting DCN of [1,2] on all disks is stored in UNTLIB, the unit library table. (See Chapter 10.)

- For each job currently logged onto the system, the starting DCN of the associated UFD (on the system disk) is stored at offset J2UFDR in the job's secondary job data block (JDB2). (See Section 3.1.3.)

If the device cluster number of the beginning of the user file directory is not available in memory, FIP must search the on-disk directory structure of the appopriate disk. This is done as follows:

- The starting device cluster number of the master file directory is retrieved from table MFDPTR (see Section 9.1.4.3).
- FIP examines the second block of the MFD, indexing by the group number (that is, the "project" number of the PPN), multiplied by two. This provides the starting device cluster number of the proper group file directory.
- The first cluster of the GFD is read into memory and FIP examines the second block, indexing by the user number (that is, the "programmer" number of the PPN), multiplied by two. This provides the starting device cluster nuumber of the proper user file directory.
- The first cluster of the UFD is read into memory.

Once the UFD has been located and read into memory, searching for the file name is a straightforward scan of the name blockettes within the UFD.

### 9.2.5 Window Turning

The file processor maintains one set of file retrieval pointers in core (in the window control block) for every individual user of an open file. This gives each user immediate access to (up to) seven file clusters. However, if an I/O request is issued for a block not described by the current set of retrieval pointers (or window), a 'window turn' must be performed.

Given the virtual block number of the desired file block, FIP can readily calculate the number of the required file cluster. This, in turn, provides the required retrieval window number. If other users have the same file open for access, FIP scans the list of WCBs to determine whether the required retrieval window is already in memory. (The retrieval window number is at offset W$REN in the window control block.) If the required window is in memory, the information is copied into the current user's WCB.

If the desired window is not available in memory, FIP uses the directory linkage information (at offset W$NXT) to scan forward through the UFD until the required retrieval blockette is obtained. FIP will, however, attempt to optimize this process, whenever possible, by using intermediate retrieval windows from the WCBs of other users.

# CHAPTER 10
# Disk I/O Subsystem [V8.0]

The disk I/O subsystem is that portion of the monitor that processes user and monitor requests to read and write disk data, along with other disk-related I/O functions. It is essentially a generic disk driver, handling functions common to all disk transfers, as well as those peculiar to each physical device. (Note, however, that it does not conform to the general device driver standards, as specified in Chapter 6.) In particular, the disk subsystem is responsible for the following functions:

- Examining the caller's transfer request block (XRB) and window control block (WCB) to determine which actual disk block is to be accessed.
- Issuing requests to FIP for any necessary window turns or file extensions.
- Invoking the disk caching code (if enabled).
- Optimizing requested transfers so as to minimize disk head movement.

This chapter discusses the general structure of the disk I/O subsystem (Section 10.1) and the data structures it uses (Section 10.2). It then presents an overview of I/O processing (Section 10.3).

## 10.1 General Structure

The disk I/O subsystem is divided into two levels of processing: the logical driver and the physical driver.

The logical disk driver (SYDVR) acts as the interface between jobs requiring disk I/O and the actual physical controller drivers. It processes individual I/O requests for file-structured data, verifying that the transfers are valid and handling general disk and file-related bookkeeping. In particular, SYDVR is responsible for the following:

- Converting file-related information (such as virtual block numbers) to disk-related information (such as FIP block numbers).
- Splitting noncontiguous transfers into two or more contiguous subtransfers.
- Requesting FIP window turns and file extends, as necessary.
- Allocating and building the required disk request queue entry blocks (DSQs) for each transfer.

The physical disk driver, on the other hand, has no knowledge of file structure but is merely concerned with disk-specific information. This level of processing consists of three major portions:

- The physical driver front end module (DISK). This module provides a layer of processing common to all disk controllers. In particular, it is responsible for invoking the caching code (if enabled), converting all logical parameters to disk-specific parameters, and forwarding requests to the appropriate device drivers.
- The caching module (if enabled during system generation). This software is included here as part of the physical level because it can, in effect, be viewed as the

equivalent of a physical driver. In this case, the disk type is central processor memory.

- The individual device drivers. There is one driver for every separate disk type on the system. Drivers are responsible for all controller-specific functions, such as readying the device, initiating transfers, and detecting transfer completion.

The logical disk driver does not need to be invoked for every request for disk data. Certain monitor components that already have access to disk-specific information, such as the file processor and the swap manager, can invoke the disk I/O subsystem directly at the physical level.

## 10.2 Data Structures

In addition to the standard device control data structures (see Chapter 6), the disk I/O subsystem uses three types of memory-resident data structures to control processing:

- A set of permanently allocated control tables, containing the status and characteristics of every disk unit on the system.
- The disk request queue entry block (DSQ), allocated dynamically from the small buffer pool and used to store information necessary for processing user I/O requests.
- The structures necessary for caching. (Caching is discussed separately in Chapter 11.)

### 10.2.1 Control Tables

The disk I/O control tables are generated during the system generation process and contain control information for all disk devices configured on the system. They are accessed by the FIP unit number (or FUN, see Section 7.5) of the disk.

### 10.2.1.1 Unit Count Table (UNTCNT) -

The unit count table (UNTCNT) is the main control table for disk operations. It contains one 1-word entry for each disk unit and controls the availability and status of each device.

The low-order nine bits of each entry contain a count of the number of users that have files open on the device. The remaining seven bits are used as flags indicating the status of the unit. These flags are obtained from the pack label in logical block 1 when the pack is mounted.

The format of each UNTCNT entry is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-8 | UC.CNT | These bits contain a count of the number of "users" that have files open on this device. Note that a single file opened by one user on two channels constitutes two "users." In effect, this is the number of window control blocks in use for files on this unit. |
| 9 | UC.TOP | If this bit is set, the user file directories on the pack are ordered with the newest files first. |

10 UC.WLO If this bit is set, the unit is write-locked and is, therefore, a read-only device. The system disk can never be write-locked.

11 UC.DLW If this bit is set, the date of last access in the file accounting blockettes should be updated only when files on this disk are modified.

12 UC.NFS If this bit is set, the unit is being processed as a non-file-structured device by one or more users.

13 UC.LCK To dismount a mounted disk, the low-order bits of the UNTCNT entry (UC.CNT) must be zero. When they are not zero, but a dismount is desired, this bit can be used to prevent further files from being opened on the unit. With UC.LCK set, an attempt by a nonprivileged user to open a file on the unit will fail. If the disk is a public disk, setting this bit will also logically remove it from the public structure.

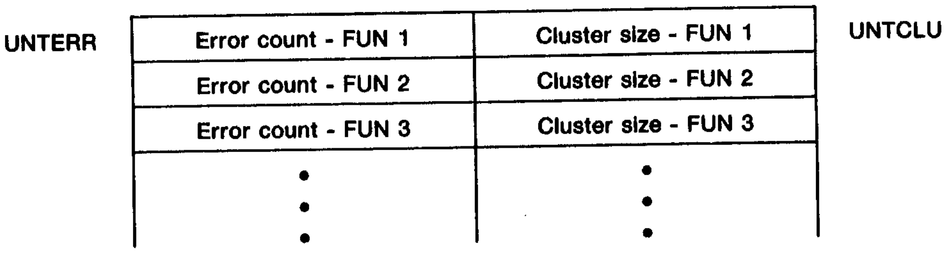
14 UC.PRI This is the public/private bit. If it is set, the disk is a private pack. Otherwise, it is public. When the disk is mounted, this bit is normally set according to the state of the public/private bit in the disk pack label, although this can be overridden.

15 UC.MNT If this bit is set, the disk is not currently mounted, and the remaining flags in this entry should be considered random. It is cleared when the disk is mounted, either explicitly or implicitly, by opening it as a non-file-structured device.

### 10.2.1.2 Unit Cluster and Unit Error Table (UNTCLU/UNTERR) -

The unit cluster (UNTCLU) and unit error (UNTERR) table contains one 2-byte entry for each disk unit configured on the system. It is used to store the pack cluster size of the pack currently mounted on the unit, as well as the error count. While there is only one combined table, each of the two items is accessed using a different label (see Figure 10-1).

The pack cluster size (accessed by UNTCLU+FUN*2) is obtained from the disk pack label when the pack is mounted. The error count (accessed by UNTERR+FUN*2) is set to zero when the pack is mounted and is used to maintain the total count of all device errors that occur on the unit.



| | |
|---|---|
| Error count - FUN 1 | Cluster size - FUN 1 |
| Error count - FUN 2 | Cluster size - FUN 2 |
| Error count - FUN 3 | Cluster size - FUN 3 |
| •<br>•<br>• | •<br>•<br>• |

**Figure 10-1:** Unit Cluster and Unit Error Table

### 10.2.1.3 Device Cluster and Cluster Ratio Table (DEVCLU/CLUFAC) -

The device cluster (DEVCLU) and cluster ratio (CLUFAC) table contains one 2-byte entry for each disk unit configured on the system. It is used to store the device cluster size and cluster ratio of the unit. While there is only one combined table, each of the two items is accessed using a different label (see Figure 10-2).

The device cluster size (accessed by DEVCLU+FUN*2) is specified for the unit during the system generation process and generally cannot be changed. The cluster ratio (accessed by CLUFAC+FUN*2) is equal to the pack cluster size divided by the device cluster size and is calculated when the pack is mounted.

| CLUFAC | DEVCLU |
|---|---|
| Cluster ratio - FUN 1 | Cluster size - FUN 1 |
| Cluster ratio - FUN 2 | Cluster size - FUN 2 |
| Cluster ratio - FUN 3 | Cluster size - FUN 3 |
| •<br>•<br>• | •<br>•<br>• |

**Figure 10-2:** Device Cluster and Cluster Ratio Table

### 10.2.1.4 Unit Owner and Unit Options Table (UNTOWN/UNTOPT) -

The unit owner (UNTOWN) and unit options (UNTOPT) table contains one 2-byte entry for each disk unit configured on the system. It is used to store the job number of the job that owns the disk pack, as well as the option settings for that pack. While there is only one combined table, each of the two items is accessed using a different label (see Figure 10-3).

The job owner byte (accessed by UNTOWN+FUN*2) contains the job number (multiplied by two) of the job that currently owns the associated disk pack. Only non-file-structured packs can be owned exclusively by one job.

The option byte (accessed by UNTOPT+FUN*2) contains various flag bits specified when the disk pack is mounted. The setting of these bits is defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | UO.CLN | The disk pack needs to be rebuilt. |
| 1 | UO.INI | The disk pack is being initialized with the on-line DSKINT program. |
| 2-3 | | (Unused) |
| 4 | UO.NCF | Directory information from this unit should not be cached, even if caching is enabled. |
| 5 | UO.NCD | Data files on this unit should not be cached, even if caching is enabled. |
| 6 | UO.WCF | All file processor writes to this unit should be write checked. |

7 UO.WCU All writes to this unit should be write checked.

| | | | |
|---|---|---|---|
| UNTOPT | Options - FUN 1 | Owner - FUN 1 | UNTOWN |
| | Options - FUN 2 | Owner - FUN 2 | |
| | Options - FUN 3 | Owner - FUN 3 | |
| | •<br>•<br>• | •<br>•<br>• | |

**Figure 10-3:** Unit Owner and Unit Options Table

### 10.2.1.5 Disk Index Table (DSKMAP) -

The disk index table (DSKMAP) contains one 1-word entry for each disk unit configured on the system. Each entry (accessed by DSKMAP+FUN*2) contains the disk type index of the corresponding disk unit. This index identifies which physical controller driver should be used for data transfers and other device specific functions. Note that this index should not be confused with the driver index, as specified at offset DDIDX in the device data block (see Chapter 6). The driver index in the DDB is zero for all disk devices.

### 10.2.1.6 Unit Size Table (UNTSIZ) -

The unit size table (UNTSIZ) contains one 1-word entry for each disk unit configured on the system. Each entry (accessed by UNTSIZ+FUN*2) contains the size of the corresponding disk unit (in device clusters). It is set by the system initialization code and generally cannot be changed.

### 10.2.1.7 Unit Library Table (UNTLIB) -

The unit library table (UNTLIB) contains one 1-word entry for each disk unit configured on the system. Each entry (accessed by UNTLIB+FUN*2) contains the device cluster number of the first cluster of the UFD for the system library — account [1,2]. This value is initialized when the pack is first mounted or when the [1,2] account is created.

Only the system disk is required to have a [1,2] account. Thus, this value can be zero if the corresponding disk does not have such a UFD.

### 10.2.1.8 SAT Size Table (SATSIZ) -

The SAT size table (SATSIZ) contains one 1-word entry for each disk unit configured on the system. Each entry (accessed by SATSIZ+FUN*2) contains the maximum pack cluster number of the corresponding unit. This value is calculated, using the corresponding entries from the UNTSIZ and CLUFAC tables, when the pack is mounted.

### 10.2.1.9 SAT Pointer Table (SATPTR) -

The SAT pointer table (SATPTR) contains one 1-word entry for each disk unit configured on the system. Each entry (accessed by SATPTR+FUN*2) contains the device cluster number of the last allocated pack cluster. When a request is received by FIP to allocate a new cluster, the search for available space begins with this device cluster number, thereby helping to prevent excessive disk fragmentation. When the value in SATPTR reaches the maximum, as specified by the corresponding entry in UNTSIZ, it is reset to one.

In addition, when a file is being "placed" on the disk, this value is set to the starting point from which space should be allocated.

#### 10.2.1.10 Count of Free Blocks (SATCTL and SATCTM) -

The SATCTL and SATCTM tables each contain one 1-word entry for each disk unit configured on the system. Corresponding entries (accessed by FUN*2) from the two tables are used together to store an unsigned, 32-bit count of free blocks available on the disk unit. The entry from SATCTL contains the least significant sixteen bits and the entry from SATCTM contains the most significant sixteen bits.

These values are initialized when the pack is mounted and are updated as blocks are allocated and deallocated.

#### 10.2.1.11 Start of the SAT (SATSTL and SATSTM) -

The SATSTL and SATSTM tables contain one 1-word entry for each disk unit configured on the system. Corresponding entries (accessed by FUN*2) from the two tables are used together to store the unsigned, 24-bit FIP block number of the start of the storage allocation file (SATT.SYS) on the disk. The entry from SATSTL contains the least significant sixteen bits, and the entry from SATSTM contains the most significant eight bits.

### 10.2.2 Disk Request Queue Entry Block (DSQ)

Disk request queue entry blocks (DSQs) are used to store request parameters and completion information for individual disk transfers. They are linked and relinked into various queues as the associated requests move through the disk I/O subsystem.

Various monitor processes have their own permanently allocated DSQs. Other processes allocate them dynamically, as needed, from the general small buffer pool.

Figure 10-4 shows the format of the disk request queue entry block. Note that the contents of many of the fields within the DSQ change as the request passes through the various processing stages. The contents of each field while the request is in the logical disk driver are discussed below.

*Offset* — *Contents*

DSQ — This word contains a link to the next DSQ in the current queue. If this is the last DSQ in the queue, or the DSQ is not in a queue, this word will contain a zero.

DSQJOB — This byte contains the job number (multiplied by two) of the requestor. (This value will be odd if the original requestor is a "system job." See Section 9.2.1.)

DSQERR Certain disk transfer errors are "recoverable." If such an error is encountered, the disk driver will retry the requested operation. This byte contains the retry count. It is initially set to a value of -9 (or -2, if the pack is being initialized) and is incremented on each transfer attempt. If the count goes to zero, the operation is aborted and a positive error code is returned in the byte. (Unrecoverable error conditions cause an immediate abort of the request.)

DSQL3Q This word contains a pointer to the mask of bits to set in the level three queue flag word (L3QUE) when the I/O operation has completed.

DSQUNT This byte contains the internal FIP unit number of the disk.

| | | | |
|---|---|---|---|
| | Queue link word | | DSQ |
| DSQERR | Retry count/error | Job number*2 | DSQJOB |
| | Pointer to L3Q bits to set | | DSQL3Q |
| DSQFBM | FIP block number (MSB) | FIP unit number | DSQUNT |
| | FIP block number (LSB) | | DSQFBL |
| DSQMAM | Buffer address (MSB) | RH11 code | DSQRFN |
| | Buffer address (LSB) | | DSQMAL |
| | Transfer word count | | DSQCNT |
| DSQFAR | Fairness count | Function code | DSQFUN |
| | Miscellaneous pointer | | DSQMSC |
| | Total transfer word count | | DSQTOT |
| | Physical disk address | | DSQPDA |
| | Optimization word | | DSQOPT |
| DSQSAV | Saved function | Unit number*2 | DSQOUN |
| | Offset pointer | | DSQPTO |
| DSQPUN | Unit number | Offset retry counter | DSQCTO |

**Figure 10-4:** Disk Request Queue Entry Block

DSQFBM This byte contains the high-order eight bits of the 24-bit FIP block number of the beginning of the transfer. The sixteen low-order bits are located at offset DSQFBL.

DSQFBL This word contains the low-order sixteen bits of the 24-bit FIP block number of the beginning of the transfer. The eight high-order bits are located at offset DSQFBM.

DSQRFN This byte contains the RH11 function code. If the device does not use the RH11, the driver will zero this byte.

DSQMAM This byte contains the high-order six bits of the 22-bit physical memory address to or from which the transfer is to take place. The low-order sixteen bits of the address are located at offset DSQMAL.

DSQMAL This word contains the low-order sixteen bits of the 22-bit physical memory address to or from which the transfer is to take place. The high-order six bits of the address are located at offset DSQMAM.

DSQCNT This word contains the number of words to be transferred. If the disk I/O subsystem has to split the original request into two or more transfers, this value will be less than the count specified by the user.

DSQFUN This byte contains the function code. Valid codes are as follows:

| | | |
|---|---|---|
| WFUN.C | 0 | Write data with write check |
| WFUN | 2 | Write data |
| RFUN | 4 | Read data |
| RFUN.C | 6 | Write check |

DSQFAR This byte contains the queue fairness counter. It is used during the head movement optimization process, discussed in detail in Section 10.3.2.

DSQMSC This word is used to hold miscellaneous information. It commonly contains a pointer to the file control block or window control block associated with the transfer.

DSQTOT This word contains the total number of blocks to be transferred. It will be less than the word count (DSQCNT) divided by 256(10) if the request must be broken into subtransfers. This is done if the original transfer crosses device cluster boundaries.

DSQPDA This word contains the physical disk address of the transfer. It normally contains the disk track and sector address in the format required by the physical controller driver.

DSQOPT This word contains the cylinder address of the requested disk block and is used during the head movement optimization process, discussed in detail in Section 10.3.2.

DSQOUN This byte contains the physical disk unit number (multiplied by two).

DSQSAV This byte can be used by the controller driver to save the requested function code.

DSQPTO This word contains a pointer to the entry in the offset table for this device type. Each offset table contains a list of head offset specifications (in micro-inches) to be used to correct for minor alignment errors.

DSQCTO This byte contains the number of retries remaining for the current head offset specification.

DSQPUN This byte contains the physical disk unit number.

## 10.3 Overview of Processing

This section presents an overview of the processing involved at both the logical and physical levels of the disk I/O subsystem.

### 10.3.1 Logical Disk Processing

A user request for disk I/O is initiated when the job sets up its transfer request block (XRB) and issues a read or write EMT. The EMT handler performs some basic validation of the request, such as determining that the channel is open and checking the validity of the buffer address and word count. The EMT handler then copies the XRB into the job's work block and passes control to the logical disk driver (SYDVR).

SYDVR does further validation of the request, such as ensuring that the requested function is consistent with the open mode of the file. It then attempts to allocate a small buffer from the monitor's general pool for use as a DSQ. If no small buffers are available, SYDVR exits through IOREDO, stalling the job until a buffer is available (see Section 6.1.2). When a buffer becomes available, SYDVR is reentered and processing continues.

Once a small buffer has been allocated, the user job is stalled and locked in memory until the transfer is complete.

At this point, SYDVR examines the transfer parameters in the caller's work block. If the requested transfer involves noncontiguous device clusters, crossing a device cluster boundary, the request is split into contiguous subtransfers. For each subtransfer, SYDVR builds a separate DSQ (if possible), converting the virtual block number to a FIP block number and loading necessary parameters into the DSQ. If SYDVR is unable to allocate all the small buffers it needs for additional DSQs, it will build as many as it can, keeping track of the unprocessed portion of the transfer through the word count in the work block. As previous subtransfers complete, SYDVR reuses the DSQs it has until the entire request has been transferred.

If the transfer is not mapped by the user's current window control block, SYDVR formats its allocated small buffer as a FIRQB and requests either a file extend or window turn (as appropriate) from the file processor before building the DSQ.

The DSQ is then queued to routine DISK, the physical disk driver front end processor, for routing to the actual device driver. SYDVR then exits through the level three return code (RTI3).

When the DSQ has been processed by the physical driver and the transfer is complete, SYDVR is rescheduled. If SYDVR could not allocate and queue enough DSQs for the entire request, it uses the DSQ just released for another subtransfer.

When the entire request has been transferred, SYDVR posts the return status to the user's XRB and sets the appropriate bit in the job's JBSTAT entry, unblocking the job. All small buffers are returned to the monitor's pool, and SYDVR exits.

### 10.3.2 Physical Disk Processing

The main entry to the physical level of the disk I/O subsystem is subroutine DISK. This subroutine acts as a front end to the cache code and the actual physical device drivers.

Transfer requests to DISK take the form of a queue of DSQs, each describing a contiguous transfer of one or more disk blocks. Upon entry, DISK retrieves the DSQ at the head of its queue and passes it to the cache code (if caching was selected during system generation).

The cache code determines whether the requested data is already in memory. If it is, and the requested function is a read, the cached data is copied to the caller's buffer and the specified L3QUE bits are set to signal completion. If the data is not in memory but is eligible to be cached, the DSQ is passed to the proper driver (as described below), with the L3QUE bits specified to reenter the cache code. When the transfer is complete, the data is "installed" in the cache before the original caller is rescheduled.

If the requested function is a write and the data is in the cache, the caller's data is copied to the cache before the DSQ is passed to the proper driver, thereby "writing through" the cache.

(See Chapter 11 for a complete description of cache processing.)

An actual disk transfer must be executed under the following circumstances:

- Caching has not been enabled.
- The data is not currently in the cache.
- The data is not eligible to be cached.
- The requested function is a write.

In each of these cases, control is transferred to routine GODISK. This routine converts any logical information in the DSQ to device-specific physical information: The FIP block number is converted to actual physical cylinder, track and sector numbers; the FIP unit number is converted to a physical drive number; and the logical function code is converted to the actual code required by the physical device controller.

The DSQ is then queued to the appropriate device driver, as determined by the physical drive number. If the queue is currently empty, indicating that the drive is not busy, the head seek is started as soon as the DSQ is entered in the queue.

When the controller is ready, the physical driver initiates the transfer. When the transfer is complete, the driver moves the DSQ to the completion queue and sets the specified L3QUE bits. This reschedules the caller -- the cache code, the logical disk driver (SYDVR), or a monitor process that invoked DISK directly.

The physical disk drivers attempt to optimize throughput by processing data transfers so as to minimize disk head movement. DSQs are entered into the driver queues sequentially, with the oldest DSQs first. But when the drive is ready to accept another request, a call is made to OPTDSK, the optimization routine. This routine determines which request will cause the least head movement and moves that DSQ to the head of the queue, even if this means deferring an earlier request.

DSQs will not be passed over indefinitely, however. Each DSQ contains a "fairness" counter, initially set to the value found in ..FCNT (a patchable monitor location with a default value of six). Each time the optimization process is executed and a DSQ is processed out of order, the fairness counters of all DSQs that were passed over are decremented. When the counter of the first entry in the queue goes to zero, that request is honored, regardless of the amount of head movement involved.

Swapping requests, file processor requests and requests from user jobs with a priority greater than zero are normally issued with their fairness counters already set to zero. Thus, these DSQs are processed immediately (subject, of course, to their position in the queue), irrespective of the current head position.

# CHAPTER 11
# Caching [V8.0]

As a disk-based system, RSTS/E can frequently suffer from disk I/O overload, whereby the file processor and the disk I/O subsystem are unable to keep up with the demands for directory and file data accesses. The resulting bottleneck can slow the entire system and greatly reduce overall system performance.

One way to alleviate this situation is to buffer frequently accessed disk data in memory. Then, when the data is accessed a second or third time, a physical disk transfer is avoided. This technique, known as "caching," shifts some of the disk workload to the CPU balances the use of both resources.

Note that caching is more useful for read operations than for write operations, since writing cached data still entails accessing the disk. That is, the data must be updated both in the cache and on the disk.

This chapter discusses general caching concepts as used by RSTS/E (Section 11.1). It then describes the data structures used (Section 11.2) and gives an overview of the actual cache processing (Section 11.3).

## 11.1 Basic Concepts

RSTS/E supports two separate forms of caching: directory caching (which stores frequently accessed directory information) and data caching (which stores blocks of data from user files as well as from directories). Non-file-structured disk data (including swap file data) is never cached.

Data is stored in memory in the extended buffer pool.

### 11.1.1 Extended Buffer Pool (XBUF)

After memory has been allocated for the monitor software and other required structures, the system initialization program (INIT.SYS) permits partitioning the remaining available memory into two regions — user memory and the extended buffer pool (XBUF). User memory contains all user job images, including both the job low segments and the shared run-time system and resident library code. The extended buffer pool is used by the system send/receive facility and by the disk caching code.

The amount of memory allocated to the extended buffer pool is a system parameter that is specified before the start of timesharing. This parameter cannot be changed without shutting down the system. If XBUF is configured, it must be at least one K-word, but can be no greater than 512 K-words.

[For a more detailed discussion of the format and use of the extended buffer pool, see Section 5.3.]

In general, all of XBUF is available for caching. On demand, the cache will return buffers to the pool for use by the message facility. As soon as the buffers are no longer needed, however, the cache immediately reuses them. If necessary, an upper limit can be imposed on the number of buffers the cache can use, thereby guaranteeing a certain allocation to the message facility and minimizing the contention for buffer space.

An upper limit can also be imposed on both directory caching and data caching, individually. For example, suppose caching as a whole can use 40 K-words of space, and directory and data caching can each use a maximum of 25 K-words. In this circumstance, if directory caching uses its maximum of 25 K-words, the maximum available for data caching is fifteen K-words. The reverse is also true. In this way, directory and data caching are both guaranteed a minimum allocation, with the amount of overlap permitting the overall cache to dynamically adjust to system and program requirements.

### 11.1.2 Directory Caching

During its normal processing, the file processor uses one 1-block local buffer (FIBUF) to store directory information. Since FIP operates serially, this (theoretically) means that the necessary directory information is in core for the duration of a particular function. In reality, this holds true only for certain unfragmented directories. It does ensure, however, that the directory cluster map is in memory throughout processing of the function.

With directory caching enabled, recently accessed directory blocks are maintained in general buffer space and, therefore, are available for reuse without requiring an additional disk access. This can greatly increase FIP throughput and, therefore, overall system performance. (Also, since caching is handled by the disk I/O subsystem, this process is transparent to FIP.)

The term "directory caching" is misleading since, with this form of caching enabled, *all* disk accesses by the file processor are stored in general buffer space. For this reason, directory caching is sometimes called 'extended FIP buffering. This may seem like a minor detail since the majority of disk accesses requested by the file processor are for directory information. It is significant, however, in light of the fact that disk-resident overlays, the system error file, and the storage allocation table of each disk are data files.

If selected during system generation, directory caching is automatically enabled at system startup. It can also be enabled during timesharing through use of the UTILTY program.

Unless at least nine K-words of XBUF have been allocated to overall caching, the directory cache code will use small buffers from the general pool whenever necessary.

### 11.1.3 Data Caching

Data caching is used to buffer blocks of data from user files as well as from directories.

NOTE

> Data caching implicitly includes directory caching since directories are themselves files. In fact, under RSTS/E, the term "data caching" is synonymous with "extended caching." Selecting data caching during system generation includes the caching of directories and *precludes* the selection of "directory caching." The two types of caching are totally different processes and a RSTS/E system can have one or the other, but not both.

If selected during the system generation process, data caching is automatically enabled at system startup as long as at least two K-words of XBUF have been allocated for caching.

(Unlike directory caching, data caching will not use small buffers from the general pool if XBUF is exhausted.) It can also be enabled after startup through use of the UTILTY program.

Data caching can be enabled for all user files or selectively on a file-by-file basis. An individual file can be selected for caching when it is opened by a privileged user. Or it can be selected for caching through the UTILTY program that sets bit UL.CHE in the first word (ULNK) of the file name blockette in the UFD (see Section 8.5.2). As long as data caching is enabled, a file marked in its UFD entry is always cached regardless of the open mode.

#### 11.1.3.1 Cache Clusters -

When data caching is enabled, disk blocks are read into memory in units of cache clusters: groups of 1, 2, 4 or 8 logical blocks, aligned with DCN 1. When a read request is received for a block currently not in memory, the entire cache cluster containing the desired block is read.

The size of the cache cluster has a significant impact on overall performance. If the cluster size is greater than the file cluster size of a noncontiguous file, some of the blocks installed in the cluster will frequently contain data that is totally unrelated to the file being accessed. On the other hand, if the cache cluster size is too small, extra disk accesses may be necessary to install the data, or the data may not even be cached at all. For example, a read of a file cluster larger than the cache cluster may need to be split into multiple reads. But RSTS/E will not cache a data request if doing so requires more than some predetermined number of physical disk accesses (see Section 11.3.2).

Cache clusters are only relevant to data caching. Directory caching always deals in units of one logical block.

#### 11.1.3.2 Caching Modes -

When a read is requested on a cached file and the data is not currently in memory, the cache code installs the new data if there is free space or if a current cluster is eligible for replacement. Data files can be cached in either of two different modes: sequential or random. The selected mode determines what makes a cluster replaceable in the cache.

Caching a file in random mode means that a cluster is eligible for replacement if it has been in the cache for more than some minimum residency time (usually one minute) without being accessed.

Sequential caching uses the same basic algorithm with one variation. When the last block in a cache cluster is accessed in a sequentially cached file, the "age" of the cluster is set to infinite, making that cluster immediately eligible for replacement. This type of caching can be viewed as "read ahead, trash behind":

- When the first block of the cluster is read, the entire cluster is installed in the cache. The remaining blocks of the cluster are immediately available for access (read ahead).
- When the last block in the cluster is read, none of the data is needed any longer so the entire cluster can be replaced (trash behind).

When a read operation from a sequentially cached file requires more than one cluster to be accessed, no data is installed in the cache for any cluster whose last block was included in the original read request. That is, only the last cluster can be installed and then only if it was not completely passed to the user. One result of this is that a cache cluster size of one essentially eliminates sequentially caching (since every data block is the last block in the cluster).

The cache mode of a particular file is determined by bit UL.CHE in the file attribute link word (UAA) of the file name blockette in the UFD (set with the UTILTY program) or by the mode in which the file was opened. Any cached files without a specified mode are cached in random mode.

The mode of caching can be specified only for data files. Random mode is always used for directory caching.

## 11.2 Data Structures

In addition to storage space in the extended buffer pool, the cache code uses other structures to control the caching process. These other structures consist of the cache control area, along with what are known as cache tags and write cache tags.

### 11.2.1 Cache Control Area

The cache control area contains various data items used to control and monitor the caching function. Figure 11-1 shows the format of this area. Each data item is discussed below. Some of these data items are used only if data caching has been selected.

| | | | |
|---|---|---|---|
| | Primary cache flag | | CH$CTL |
| | Charge time flag | | CH$TIC |
| | Seconds since system boot | | CH$SEC |
| | Active tag list head pointer | | CH$NXT |
| | Active tag list tail pointer | | CH$PRV |
| | Address (in MMUs) of cache tag region | | CH$PAR |
| | Free tag list pointer | | CH$TAG |
| | Small buffer pool flag | | CH$USM |
| | Free write tag list pointer | | CH$WTG |
| | Total number of tags | | CH$LIM |
| | Maximum number of directory tags | | CH$FLM |
| | Maximum number of data tags | | CH$DLM |
| CH$DAT | Data cache flag | Data cache mode | CH$MOD |
| | Cache cluster size | | CH$BSZ |

**Figure 11-1:** Cache Control Area

*Location* *Contents*

CH$CTL This word contains the primary caching control flag. If the value is zero, caching is disabled; if the value is nonzero, caching is enabled.

CH$TIC If this word is nonzero, cache time is charged.

CH$SEC This word contains the approximate number of seconds that have passed since the system was bootstrapped (modulo 64K).

CH$NXT This word is the head pointer for the list of "active" cache tags (see Section 11.2.2). It contains a pointer to the cache tag (at offset CH.PRV) describing the first block of data installed in the cache.

CH$PRV This word is the tail pointer for the list of "active" cache tags. It contains a pointer to the cache tag (at offset CH.NXT) describing the last block of data installed in the cache.

CH$PAR This word contains the address, in memory management units (MMUs), to load into APR 6 when accessing the cache tags.

CH$TAG This word contains a pointer to the first available cache tag.

CH$USM This word contains a flag indicating whether or not attempted allocations from the extended buffer pool should "fall back" to the general small buffer pool (see Section 5.3.3). If the value here is nonzero, buffers will be allocated from the monitor's general small buffer pool. If it is zero, the general pool will not be used.

CH$WTG This word contains a pointer to the first available write cache tag (see Section 11.2.3). This word is used only if data caching has been selected.

CH$LIM This word contains the total number of cache tags that have been configured into the system.

CH$FLM This word contains the maximum number of cache tags that can be used as directory cache tags.

CH$DLM This word contains the maximum number of cache tags that can be used as data cache tags. It is used only if data caching has been selected.

CH$MOD This byte indicates which data files should be cached. If the value here is zero, no files will be cached. If the value is one, only those data files selected for caching will be cached. If the value is -1, all data files will be cached.

CH$DAT If the value in this byte is nonzero, data caching has been selected. If it is zero, data caching has not been selected.

CH$BSZ This word contains the cache cluster size in blocks (a power of two, from 1 to 8). If directory caching has been selected without data caching, the value here will be one.

### 11.2.2 Cache Tags

Cache tags are used to identify the directory and data blocks currently installed in the cache. Tags are also used to keep track of the last time each cache cluster was accessed.

Cache tags reside primarily in the extended buffer pool and are generated by INIT before system startup. (If directory caching alone is selected, tags can reside in the general small buffer pool if XBUF space is exhausted.)

Both available and in-use tags are maintained in doubly linked lists, pointed to by fixed memory locations in the cache control area. The in-use list is kept in order by length of residency in the cache, with the newest entries linked to the head of the list.

The cache tags used for data caching vary slightly in format and content from those used when only directory caching is selected. Their function, however, is basically the same. Figure 11-2 shows the format of directory caching tags, while Figure 11-3 shows the format of data caching tags. Each field is discussed below.

| | | | |
|---|---|---|---|
| | Pointer to the next cache tag in use | | CH.NXT |
| | Pointer to the previous cache tag in use | | CH.PRV |
| CH.MSB | FIP block numb (MSB) | FIP unit number | CH.UNT |
| | FIP block number (LSB) | | CH.LSB |
| | Time of last access | | CH.TIM |
| | Pointer to cached data | | CH.DAT |

**Figure 11-2:** Directory Cache Tag

| | | | |
|---|---|---|---|
| | Pointer to the next cache tag in use | | CH.NXT |
| | Pointer to the previous cache tag in use | | CH.PRV |
| CH.MSB | FIP block numb (MSB) | FIP unit number | CH.UNT |
| | FIP block number (LSB) | | CH.LSB |
| | Pointer to cached data | | CH.DAT |
| | Pointer to DSQs waiting to update cache | | CH.BLS |
| | Tag Type | | CH.TYP |
| | Time of last access | | CH.TIM |

**Figure 11-3:** Data Cache Tag

*Offset* *Contents*

CH.NXT This word contains a pointer to the next cache tag (at offset CH.PRV) in use. If this is the last tag in the list, this word will point to fixed memory location CH$PRV.

CH.PRV This word contains a pointer to the previous cache tag in use. If this is the first tag in the list, this word will point to fixed memory location CH$NXT.

CH.UNT This byte contains the internal FIP unit number of the disk from which the associated cache cluster was read. If the value in this byte is a -1, an error was detected during the disk transfer and the data in the associated cache cluster should be considered invalid.

CH.MSB This byte contains the high-order seven bits of the 23-bit FIP block number of the start of the data in the associated cache cluster. The low-order sixteen bits are located at offset CH.LSB.

CH.LSB This word contains the low-order sixteen bits of the 23-bit FIP block number of the start of the data in the associated cache cluster. The high-order seven bits are located at offset CH.MSB.

CH.TIM This word contains the time (in RSTS/E internal format) that the cache cluster was last accessed.

CH.DAT This word contains a pointer to the cluster buffer in memory that contains the actual cached data.

CH.BLS This word contains a pointer to the list of DSQs that are waiting to update the cache cluster. (Data caching only)

CH.TYP This word contains the tag type — either directory or data cache. (Data caching only)

### 11.2.3 Write Cache Tags

Write cache tags are used to store DSQ information during disk transfers to and from the cache. During these disk transfers, the cache code modifies the DSQ and issues it to the driver. The original contents of the altered DSQ fields are saved in a write tag and then restored when the transfer is complete.

On a read to the cache, the following DSQ fields are altered:

- DSQMAM and DSQMAL are set to point to the cache buffer.
- DSQFBM and DSQFBL are set to point to the start of the cache cluster on disk.
- DSQL3Q is set to return control to the cacher.
- SQCNT is set to the cache cluster size.
- DSQMSC is set to point back to the write tag.

On a write from the cache, only the DSQL3Q and DSQMSC fields are altered.

[See Section 10.2.2 for a description of each of these fields in the DSQ.]

Write tags are located in the extended buffer pool and are generated by INIT before system startup. Available tags are maintained in a singly linked list pointed to by fixed memory location CH$WTG in the cache control area. Write tags in use are pointed to by their associated DSQs.

Since directory caching never does direct transfers from the disk to the cache, it does not use write tags.

Figure 11-4 shows the format and content of a write tag. Each entry is discussed below.

*Offset* *Meaning*

WT.TYP This byte contains a flag that indicates whether this is really a write tag or whether it was borrowed from the cache tag pool.

WT.MAM This byte is used to save DSQMAM.

WT.MAL This word is used to save DSQMAL.

WT.CNT This word is used to save DSQCNT.

WT.L3Q This word is used to save DSQL3Q.

WT.COF This byte is used to save the offset into the cache cluster of the user transfer.

WT.CHM This byte is used to save the caching mode.

WT.CTP This word is used to save the pointer to the corresponding cache tag.

WT.MSC This word is used to save DSQMSC.

| | | | |
|---|---|---|---|
| WT.MAM | Saved DSQMAM | Write tag type | WT.TYP |
| | Saved DSQMAL | | WT.MAL |
| | Saved DSQCNT | | WT.CNT |
| | Saved DSQL3Q | | WT.L3Q |
| WT.CHM | Saved cache mode | Saved cluster offset | WT.COF |
| | Saved pointer to cache tag | | WT.CTP |
| | Saved DSQMSC | | WT.MSC |

**Figure 11-4:** Write Tag

## 11.3 Overview of Processing

Depending on the type of caching selected during system generation, one of two different caching modules is linked into the RSTS phase of the monitor. If directory caching alone is selected, module BUF is linked in. If data caching is selected (which implicitly selects directory caching), module BUFEXT is linked in. Both these modules are included in the disk I/O subsystem (see Chapter 10). They run at processor priority level three and exit through the level three return code (RTI3).

This section presents a brief overview of each of these modules. (No attempt has been made to provide complete details of these highly complex routines.)

### 11.3.1 Directory Cacher (BUF)

When the directory caching module (BUF) is entered, it does a series of preliminary checks to see whether the request is even eligible for caching. To be eligible, the request must be for a single disk block on a mounted file-structured disk. It can originate from either the file processor or from a user, but if it is from a user, it must be for a directory block. If the request does not meet this set of criteria, a normal uncached transfer is done and the module exits.

If BUF determines that the transfer should be cached, processing then proceeds according to whether the requested function is a read or a write.

- *On a Read*
  If the data is already in the cache, it is copied to the user's buffer. The cache tag is updated and relinked to the front of the tag list, and the module exits.

  If the data is not currently in the cache, a normal transfer is done to the user's buffer. The cache is checked again since a simultaneous request for the same block could have installed it. If the data is still not in the cache, BUF tries to allocate a cache tag and buffer. If it is unable to do so, the module exits. Otherwise, the data is copied from the user buffer and the cache tag is linked to the front of the tag list.

- *On a Write*
  If the data is not currently in the cache, a normal uncached transfer is done and the module exits.

  If the data is in the cache, it is overwritten from the user's buffer. The cache tag is updated and linked to the front of the tag list. The data is then written to the disk from the user's buffer. If an error occurred on the transfer, the cache tag is marked invalid and linked to the end of the tag list. The module then exits.

### 11.3.2 Data Cacher (BUFEXT)

When data caching is selected, BUFEXT replaces BUF as the cache module, handling both directory and data caching. This works quite nicely since directories are themselves files.

When BUFEXT is entered, it first checks to determine if the request is eligible for caching. To be eligible, the request must be for disk data from a file on a mounted file-structured disk. The file must be selected for caching, and the transfer must not be a swap or a run-time system or resident library load. If the request does not meet this set of criteria, a normal uncached transfer is done and the module exits.

If BUFEXT determines that the transfer should be cached, processing then proceeds according to whether the requested function is a read or a write.

- *On a Read*
  If the read involves only one cache cluster, BUFEXT determines whether the cluster is already installed in the cache. If it is, the data is copied from the cache cluster to the user's buffer. If the cache mode is sequential and the last block of the cluster was just passed to the user, the cache tag is marked "expired" and linked to the end of the tag list. Otherwise, the time is updated and the tag is linked to the front of the list. The module then exits.

  If the required cache cluster is not currently installed, BUFEXT determines whether the cache mode is sequential and, if so, whether the read accesses the last block of the cluster. If it does, the transfer is not cached; I/O proceeds normally and the module exits. Otherwise, an attempt is made to obtain a cache buffer and tag. The entire cache cluster is read from the disk into the cache buffer, and the required data is transferred to the user's buffer. The cache tag is linked to the front of the tag list and the module exits.

  If the read involves more than one cache cluster, BUFEXT determines whether the cache mode is sequential. If so, a normal uncached transfer is done for all but the last cluster, which is then treated as a single cluster read (see above).

  If the request is a multi-cluster read in random mode, BUFEXT determines how many of the required clusters are currently not in the cache. If more than a predetermined number (normally two) are missing,* the transfer is not cached; I/O proceeds normally and the module exits. Otherwise, an attempt is made to obtain the necessary cache buffers and tags. The clusters are read from the disk and the required data is transferred to the user's buffer. The tags are linked to the front of the tag list and the module exits.

- *On a Write*
  First, the data is written from the user's buffer to the disk. Then, for each cache cluster currently installed in the cache and affected by the write, the necessary data

* This number, known as the cache splitting factor, can be altered with a monitor feature patch.

is copied from the user's buffer to the cache. If an error occurred on a physical disk transfer, the cache tag of the corresponding cluster is marked invalid and linked to the end of the tag list. The module then exits.

# Part IV
# TERMINAL SERVICE AND THE HUMAN INTERFACE

In its most general sense, terminal service refers to the collection of system components that provide all the terminal-specific functions of the RSTS/E operating system. In addition to the software that actually services terminal devices, this broad definition includes things such as the modules that process terminal-oriented system directives (such as "set terminal characteristics") and the Control-T "mini-SYSTAT", as well as the support code for the Forms Management System (FMS-11).

This document, however, uses a more limited definition of terminal service, restricting the term to those modules that constitute the actual terminal handling software. As such, terminal service is a standard device driver, following all the rules and conventions described in Chapter 6. However, since user jobs are generally associated with user terminals on a one-to-one basis, terminal handling under RSTS/E involves more than just physical device I/O. In addition to the standard functions required to support terminal input and output, the RSTS/E terminal service module is responsible for creating new jobs on the system and for detaching existing jobs.

This part of the *RSTS/E V8.0 Internals Manual* discusses the general structure and operation of the terminal service module, together with other aspects of the RSTS/E human interface. Chapter 12 discusses some basic concepts concerning the operation of terminal devices, along with additional features offered by the terminal service software. Chapter 13 presents an overview of the data structures and procedures used by terminal service. Chapter 14 discusses the general RSTS/E human interface and the characteristics of keyboard monitors.

# CHAPTER 12
# Terminal Service Features and Functions [V8.0]

As part of the physical interface between the user and the RSTS/E operating system, the terminal service module is one of the major determining factors in how the RSTS/E environment "feels" to the user. The extent to which terminal service supports the physical characteristics of various types of terminals, along with any additional operational features provided by the software, determines whether the RSTS/E user finds the system "friendly" and easy to use or "unfriendly" and intimidating.

This chapter discusses some basic concepts concerning the physical operation of terminal devices, as well as the major operational features offered by the terminal service module.

## 12.1 Supported Hardware Characteristics

The characteristics of an individual terminal device are determined by the terminal itself, as well as by the communications interface through which the terminal is connnected to the host computer. However, while there is a wide variety of possible terminal/interface combinations, most hardware configurations consists of only a limited set of characteristics and features.

The following sections describe the interface and terminal characteristics supported by the RSTS/E terminal service software.

### 12.1.1 Interface Characteristics

Internally, digital computers transfer data using parallel transmission. That is, there is one wire for each bit of data and an entire byte of data is transferred at one time. This form of transmission is extremely fast. However, when used for external communications (between the computer and its peripheral devices), parallel transmission can become quite costly. As the distance between the computer and the peripheral increases, so does the cost of the multiple wire connections and any amplification necessary to compensate for signal loss over the line. Thus, most external communication is handled using serial transmission where only one data wire is used and data is transferred one bit at a time.

To resolve the differences between these two forms of transmission, computers require "interfaces" to communicate with peripherals. Interfaces convert internal parallel signals into serial signals for transmission and then convert incoming serial signals back into parallel signals during reception. This is only the *minimum* function of an interface, however. Interfaces have many other features and characteristics.

This section presents a brief summary of the principles of data communications and the characteristics of communications interfaces. For a more comprehensive overview refer to the *Terminals and Communications Handbook.*

### 12.1.1.1 Synchronous versus Asynchronous Communication -

Interfaces are usually designed for either asynchronous or synchronous communication. Some interfaces, however, are capable of both. In asynchronous communication, the sender transmits a character whenever one is ready. The sender can transmit characters one after the other or there can be some time interval between characters.

Each character is, in effect, a complete self-contained message. Thus, each character transmitted must also contain information telling the receiver when the character begins and when it ends. The portion of the message which indicates the beginning of the character is called the "start bit," while the portion which indicates the end of the character is called the "stop bit(s)."* These bits serve the same function as the blank spaces between words in written English.

Asynchronous communication is easy to implement. Characters are transmitted as they are ready, and thus, no buffering or temporary storage of data is necessary. But since two out of the ten bits required to transmit an ASCII character serve as start and stop bits rather than data, asynchronous communication tends to be inefficient.

On the other hand, synchronous transmission uses the communication line very efficiently. In synchronous communication, the sender transmits an entire block of characters at a time. However, a much more complicated protocol is required, including special synchronization characters at the beginning and end of the message, to assure the integrity and proper sequencing of the data. Thus, this type of communication is somewhat more difficult to implement.

### 12.1.1.2 Transmission Speeds -

During any form of data communication, the sender and the receiver must agree on the rate at which data is to be transmitted. This is most accurately measured as "bps" (bits per second) but is generally (somewhat less accurately) expressed as "baud rate."* Obviously, the higher the rate of transmission, the more data that can be communicated in a given period of time. But, as the rate of tramission increases, so do the effects of "noise" (or interference) on the line. Higher speed lines require special conditioning to reduce noise and are, therefore, more expensive to install or lease.

Some terminals can receive and transmit at only one speed. These usually operate in (ASR33) teletype mode and transfer data at a baud rate of 110 — roughly equivalent to 10 characters per second. Other terminals, however, are capable of operating over a wide range of speeds. In addition, some can operate with "split speed" settings where data is transmitted and received at different rates.

The data speeds actually used by a particular terminal, however, depend on the line speeds allowed by the communications interface. For example, a VT100 is capable of receiving data at a baud rate of 19,200. However, a DZ11 asynchronous multiline interface cannot operate at this speed. Thus, a VT100 connected through a DZ11 is restricted to a lower maximum speed of 9600 baud.

* There can be more than one stop bit per character.

* Technically, the baud rate refers to the number of signal elements per second. There are some data communication techniques which transfer more than one bit every time the signal changes. Thus, the baud rate could actually be less than the number of bits per second. In reality, however, the baud rate is almost always equal to bits per second.

Because many terminals operate in teletype mode, the system start-up procedure automatically sets the line speed of all interfaces to a default data rate of 110 baud. To change the speed over a particular communications line, the user can either issue the "set terminal characteristics" system directive or use the TTYSET system utility.

### 12.1.1.3 Parity Checking -

Many interfaces (as well as some terminal devices themselves) are designed to do parity checking on incoming data. Parity checking is used to detect single bit errors in transmission. Sometimes a bit can be changed due to noise (or interference) on the line, and since each ASCII character is represented by a unique combination of bits, this makes the transmitted character appear as a completely different character.

Parity checking guards against these single bit errors by counting the number of bits that are set in the received character and testing that number against an odd/even standard. If the device uses odd (or "mark") parity, an odd number of bits should be set; if the device uses even (or "space") parity, an even number of bits should be set. If the number of set bits does not match the appropriate parity sense, the device indicates that the received character is in error. (For example, the VT100 displays a checkerboard character on the screen in place of the received character.)

Some interfaces will generate the appropriate bit for each character they receive from the computer for transmission. To do this, the interface hardware counts the number of bits set in each *7-bit* ASCII character and then sets or clears the eighth (high-order) bit so that the total number of bits set is either odd or even, as expected by the receiving hardware.*

If the sending interface does not generate the parity bit, but the receiving hardware expects it, the software can be conditioned to generate the parity bit before each character is output.

Terminal service does not do parity checking on incoming characters and therefore ignores any parity bits set and sent by the hardware.

### 12.1.1.4 Modem Control and Ring Values -

Computers communicate binary data. That is, a bit of data can have one of only two possible values: a one or a zero. However, communicating with peripherals at a great distance often requires the use of lines which are leased from a "common carrier" (such as the telephone company), and such lines are usually designed to meet the requirements of voice transmission. Voice transmission is *analog*, with each "bit" having a continuous (infinite) range of possible values.

To permit the use of voice communications lines for the transmission of binary data, the communications medium requires the use of a "modem." A modem (*mod*ulation/*dem*odulation) or "data phone" is a device which does the following:

- Converts binary data from the sender into an analog sound signal which can be transmitted over a telephone circuit. This is called "modulation."
- Converts the analog telephone signal back into a binary signal for the receiver. This is called "demodulation."

* Some devices have *selectable* parity sense and can be operated with ODD PARITY, EVEN PARITY, or NO PARITY.

Some modems have special cradles (called "acoustic couplers") that hold the handset of an ordinary telephone, while others are designed to hook directly into the telephone circuitry. In either case, the communications connection is established by dialing a standard telephone number, just as if a normal telephone voice transmission were being made.

Some modems can also perform special functions such as automatic dialing, automatic answering, and automatic disconnecting of the line when the communication is complete.

Most communications interfaces are designed to support the use of modems in the communications circuit. When a modem is used, it must lie between the interface and the communications line. (Both the sender and receiver must be connected to a separate modem.)

Each time a connection is established over a dial-up line controlled by a modem, the system sets certain standard defaults for the various terminal values and characteristics. The user must then make any adjustments necessary for the particular terminal being used. This is done by either issuing the "set terminal characteristics" system directive or running the TTYSET system utility. When the user logs off, the terminal reverts to the standard settings.

However, if certain remote lines are always used for predetermined types of terminals, the system manager (or any privileged user) can use a privileged option of TTYSET to specify "ring values" for those remote lines. These ring values are then used in place of the standard defaults to initialize the line characteristics. Ring values remain in effect for a remote line throughout the current timesharing session, unless they are changed by running TTYSET a second time.

Ring values can be specified for the terminal line width, fill characteristics, and speed, along with a variety of other terminal and interface characteristics.

#### 12.1.1.5 Single Line versus Multiline -

Some interfaces handle only a single line while others handle multiple lines. Multiline interfaces (also known as "multiplexed" interfaces) can usually operate with a maximum of eight or sixteen sublines.

NOTE

The console terminal is always connected to a single line interface.

### 12.1.2 Terminal Characteristics

From the point of viewof terminal service, the communications requirements of a terminal (such as transmission speed, data format, and so on) are determined by the hardware interface. Thus, these characteristics are specified for the interface and communications line, not for the individual terminal. However, certain other features (most notably, the terminal display characteristics) are strictly a function of terminal type and are, therefore, specified on a terminal-by-terminal basis.

#### 12.1.2.1 Full Duplex versus Local Echo -

Generally, characters typed at a terminal are sent only to the computer. The terminal service then echoes each character back so that it will be displayed at the terminal. During this process, terminal service also translates and processes certain characters. For example, a carriage return character (CR, ASCII value 13(10)) is echoed as a carriage return followed by a line feed, and a control character is echoed as an up arrow character (^) followed by the appropriate letter (for example, Control-A echoes as ^A).

Some terminals, however, echo characters locally as they are generated. If the terminal's characteristics have been set to "local copy" within the software, the system does not echo characters received from such a terminal.

### 12.1.2.2 Horizontal Tab Control -

If the terminal has the hardware to process horizontal tabs, the system outputs the horizontal tab (HT, ASCII value 9(10)) character without translation or processing. Otherwise, the HT character is translated to the appropriate number of spaces that will align the cursor with the next horizontal tab stop. When tabs are processed by the software in this fashion, tab stops are set every eight character positions, beginning with position 1.

### 12.1.2.3 Form Feed and Vertical Tab Control -

If the terminal has the hardware to process form feeds and vertical tabs, the system outputs the form feed (FF, ASCII value 12(10)) and vertical tab (VT, ASCII value 11(10)) characters without translation or processing. Otherwise, these characters are translated to four line feed (LF, ASCII value 10(10)) characters.

### 12.1.2.4 Lowercase Input -

If the terminal can transmit the full ASCII character set, the system echoes and uses the lowercase alphabetic characters without translation Otherwise, the system translates any lowercase characters received from the terminal to their uppercase equivalents.

### 12.1.2.5 Lowercase Output -

If the terminal can display lowercase characters, the system transmits all such characters to the terminal without modification. Otherwise, the system translates any lowercase character to its uppercase equivalent before transmitting it to the terminal.

Note that some terminals send both uppercase and lowercase characters but can display only uppercase characters. On such terminals, the echo response to a lowercase character is the uppercase equivalent. No visual indication is given that the character transmitted to the computer was lowercase.

### 12.1.2.6 Scope RUBOUT Processing -

If the terminal is a scope, or CRT (cathode ray tube) display device, the system echoes the rubout character (RUBOUT or DEL, ASCII value 127(10)) as a backspace, followed by a space, followed by another backspace. This sequence erases the deleted character from the screen.

If the terminal is not a scope, a rubout character is echoed by printing a backslash character (\, ASCII value 92(10)) and the last character typed. The last character typed is then removed from the terminal input buffer. Subsequent rubout characters cause the previously typed characters to be printed and then removed from the input buffer. This process continues until a character other than a rubout is received. At that point, another backslash character is echoed (to delimit the characters that have been deleted) before the incoming character is echoed.

### 12.1.2.7 Fill Counts -

Many terminals require time to complete a physical action initiated by a control character. For example, after receiving a carriage return character, an LA-36 requires a certain amount of time for the print head to physically return to the first character position. To permit such terminals to synchronize themselves properly before printing the next character, the terminal service automatically generates a variable number of null characters (NUL, ASCII value 0) as fill characters after outputting various control characters. The number of fill characters generated depends on the control character itself, as well as the timing characteristics of the terminal.

### 12.1.2.8 XOFF/XON Synchronization -

Many terminals can use the "XOFF/XON" (transmit off/transmit on) synchronization protocol. This protocol permits both the terminal and the host computer to exercise some control over the data they receive by telling the sender when to stop and start transmitting characters. This is accomplished by sending the XOFF character (ASCII value 19(10)) to stop transmission and the XON character (ASCII value 17(10)) to resume transmission.

Terminals that initiate the XOFF/XON protocol typically maintain a character input buffer for temporary storage of characters received from the host computer for display. When this buffer is nearly full, the terminal sends an XOFF to the computer. On receiving this character, the terminal service software suspends transmission of display characters (if it has been conditioned to do so). If the system stops transmitting, the terminal can deplete the buffer. When the buffer is nearly empty, the terminal sends an XON character to the computer and terminal service then resumes transmission.

If the software has not been conditioned to detect and process XOFF and XON, these characters are merely treated as normal input. Transmission of output characters is not suspended and the terminal input buffer continues to fill. If the terminal cannot catch up, the buffer will eventually overflow, resulting in lost display characters.

The XOFF and XON characters are equivalent to the Control-S and Control-Q keystrokes, respectively. Since the terminal service software cannot distinguish between characters typed at the keyboard and those generated within the terminal hardware, terminal users can also initiate the process of suspending and resuming output to the display. This is especially useful when a great deal of output is displayed at one of the higher baud rates.

From the other end of the communications line, the computer software can use the XOFF/XON characters to control the transmission of characters from a terminal that responds to the synchronization protocol. If the software can no longer store incoming characters due to a shortage of small buffer space, it sends an XOFF character to the terminal. When the terminal receives the XOFF, it suspends transmission of all characters except XOFF and XON. Characters typed by the user are stored in the terminal keyboard output buffer. If this buffer fills, further keystrokes are discarded. When system small buffer space becomes available, the software sends an XON, the terminal resumes sending characters from the output buffer and user input is again accepted from the keyboard.

If the terminal does not respond to the XOFF/XON protocol, the computer software simply stops accepting input when buffer space is exhausted. Additional input from the user is lost until buffer space is again available.

Under RSTS/E, each direction of this protocol is treated independently. That is, it is possible for a terminal to send (but not respond to) the XOFF and XON characters, and vice versa. Generally, most terminal devices that can be used to enter user input are deemed able to initiate the protocol, thereby permitting the user to enter Control-S and Control-Q to suspend and resume output. On the other hand, if the terminal is a scope, or CRT (cathode ray tube) display device, it is also considered capable of responding to XOFF/XON characters from the host computer.

When an XOFF character is sent, only transmission from the receiver of the XOFF is suspended. For example, when the terminal sends an XOFF to the host computer, display characters are no longer transmitted by terminal service, but the terminal can still transmit user keystrokes to the computer. Thus, the software can be configured so that it will resume transmission upon receiving characters other than an XON, thereby overriding the synchronization protocol. Under RSTS/E, the software can be conditioned in one of two ways: either transmission will resume only after receiving an XON or a Control-C, or it will resume after receiving *any* input character. Note that since the software cannot distinguish between keystrokes entered by the user and data generated within the terminal hardware, whichever option is selected will apply to *all*circumstances under which an XOFF is sent to the computer.*

## 12.2 Software Features

The RSTS/E terminal service software supports a variety of features that can be accessed by a user program, regardless of the actual physical characteristics of the terminal with which the program is operating. Some of these features are options selectable at system generation time. Others are standard on all RSTS/E systems.

Each of these hardware-independent features is discussed briefly in the following sections. Complete details for writing applications that use these features are provided in the RSTS/E Programming Manual and the RSTS/E System Directives Manual.

### 12.2.1 Pseudo-Keyboards

A pseudo-keyboard is a logical device with the characteristics of a terminal but with no associated *physical* terminal. Like a terminal, a pseudo-keyboard has an input buffer and an output buffer, both of which come from the small buffer pool. User programs can send input to and get output from these buffers just as if the program (rather than a human user) were typing at a terminal and viewing the terminal screen. Thus, using a pseudo-keyboard, one user job can do terminal input to other user jobs without tying up a physical terminal.

Up to 127(10) pseudo-keyboards can be configured into a system. Since each copy of the BATCH system utility requires one pseudo-keyboard, the system generation procedure

* There is one exception to this, however. If a terminal device is opened in the special "enable XOFF/XON processing" mode, the *only* characters accepted or processed by terminal service are the XOFF and XON characters. Normally, all other input is discarded. If this mode is used in conjunction with "binary input" mode, other input characters are passed to the user program, but except for XOFF and XON, the special control characters (such as Control-C, escape, and so on) are ignored by terminal service and are not processed in their usual fashion. (See Section 12.2.11.)

automatically configures at least one into all systems. Additional pseudo-keyboards can be configured for other applications that use them.

The system assigns a device name of "PKn:" to each pseudo-keyboard and associates each one with a keyboard unit number, "KBm:" (but not with a physical terminal). For example, the system can associate PK5: with KB8:, where no physical keyboard number 8 exists. But while the associated keyboard unit is not a physical terminal, it *is* a real device as far as the software is concerned. Thus, each pseudo-keyboard has two device data blocks associated with it: one for its KB side and one for its PK side (see Section 13.1).

Using a pseudo-keyboard involves a controlling job and a controlled job. The controlling job opens the pseudo-keyboard (PKn:) and then "types" a LOGIN sequence, thereby creating the controlled job which runs on the associated keyboard unit (KBm:). The controlling job can then execute user programs by "typing" the RUN command, along with any additional required program input, on the pseudo-keyboard. The controlled job, however, is not aware that it is dealing with a pseudo-keyboard. Instead, it does input and output on its own keyboard, KBm:. Figure 12-1 shows the interaction between the controlled and controlling jobs.

A pseudo-keyboard must be opened for access, just like a normal terminal. However, only two open modes are available and they are used to specify the disposition of the controlled job once the pseudo-keyboard is closed. If the open mode is one, the controlled job will be detached when the pseudo-keyboard is closed; otherwise, the controlled job will be killed.

When the PK side of a pseudo-keyboard is open, its KB side functions like a real keyboard. It can be opened, closed, assigned and deassigned. However, when the PK side is closed, the KB side functions like a disabled terminal, and terminal service will not process input from it or send output to it.

Terminal service normally transfers data to a pseudo-keyboard in full duplex. That is, data that is output (or "typed") to the pseudo-keyboard is echoed in the output buffer of the associated keyboard unit and can be read as input (or "viewed") by the controlling job. However, this feature can be disabled (using the SPEC function) if the application is such that only program output from the controlled job is desired.

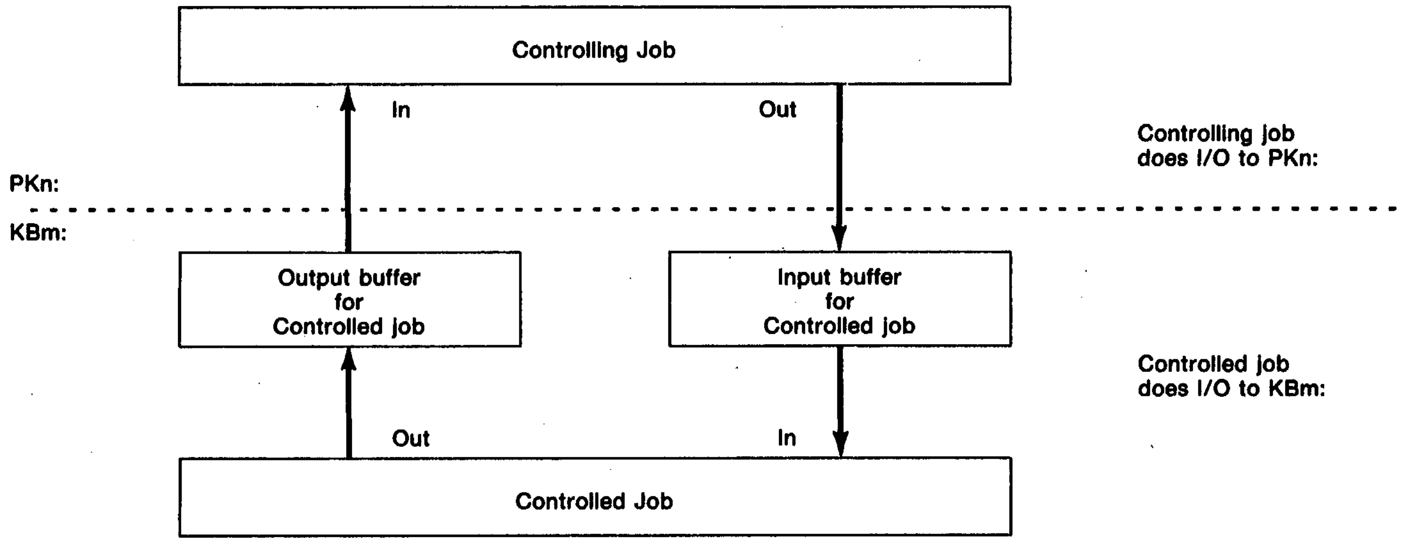

Controlling Job
In
Out
Controlling job does I/O to PKn:
PKn:
KBm:
Output buffer for Controlled job
Input buffer for Controlled job
Controlled job does I/O to KBm:
Out
In
Controlled Job

**Figure 12-1:** Pseudo-Keyboard Operations

### 12.2.2 Multiterminal Handling

Multiterminal handling is a system generation option that allows one user program to interact simultaneously with several terminals on one I/O channel. Using this feature, a single program can service low volume terminal I/O associated with several keyboards, thereby eliminating the need to run separate copies of the same program on each terminal.

A user program controls several terminals by establishing a master keyboard on a channel after assigning various other terminals as slaves. To perform input or output, the program executes record I/O requests on the channel. Input can be requested from a specific terminal or from any terminal with pending input. In the latter case, terminal service performs a round-robin search for terminals in need of I/O processing.

The terminal service can be requested to stall program execution automatically in the absence of keyboard input. The program is made eligible to run again when input is received from the master keyboard or from one of the slaves.

### 12.2.3 Echo Control

Under normal operation, RSTS/E echoes characters as they are typed. A user can also "type ahead." That is, input can be entered faster than the system can process it. When this happens, the terminal service stores the input to wait for processing, but it also echoes each character typed at the current cursor position on the screen. While this process is useful in many applications, it can be undesirable in applications that display prompts and forms.

Echo control is a system generation feature that modifies the way the system handles terminal echoing. Instead of echoing characters as they are typed, the terminal service echoes characters only within a field that the user program has declared as "active." If no field is currently active, the terminal service stores typed characters and echoes them when a field is declared.

To use echo control, a user program must open the terminal (on a nonzero channel) in the echo control mode. Once the terminal is opened, no echoing is done until a field is declared on the screen. Declaring a field entails specifying the field size, the handling of excess characters, and the special "paint" character to be echoed for character deletion sequences. Excess characters can be retained as input for the next field or can be treated as invalid and echoed with the audible BEL character.

In echo control mode, all characters returned to the user program have had the parity bit stripped and have ASCII values in the range of 1 to 127(10). Synchronization and editing characters are not passed to the program. Delimiters are passed to the program but are never echoed.

### 12.2.4 Escape Sequence Processing

An escape sequence is a series of characters preceded by an escape character (ESC, ASCII value 27(10)) and is generally used to represent a control function. Escape sequences can be used both in terminal output, if the terminal hardware recognizes such sequences, and in terminal input, if escape processing has been enabled for the terminal. (Escape sequence processing can be enabled for a terminal either with the "set terminal characteristics" system directive or through a special OPEN mode.)

When escape sequences are sent as output to a terminal that can process them, they are normally used to perform certain control functions on the video screen, such as moving the cursor or erasing part of the screen. The terminal recognizes the ESC as a control prefix and treats subsequent characters as a command.

When an ESC character is encountered in the input stream from a terminal, system processing depends on whether or not escape sequence processing has been enabled for the terminal. If escape sequence processing has not been enabled, the terminal service treats the incoming ESC as an input delimiter and echoes the dollar sign ($) character.

If escape sequence processing has been enabled, terminal service examines the input and passes the program a character string of the following format:

- A character with ASCII value 128(10) — a null with the parity bit set — followed by
- The characters in the sequence (minus the initial ESC), with no data conversion, followed by
- A character with ASCII value 155(10) — an ESC with the parity bit set

None of the characters in the escape sequence are echoed at the terminal.

An escape sequence is terminated when the terminal service has received a character string that matches one of several known sequence formats.

### 12.2.5 Private Delimiters

A private delimiter is a character used as a terminal input delimiter within a specific user program. Any printing or nonprinting character can be defined as a private delimiter:

- A letter or a number
- A function key, such as DELETE
- A control character, such as Control-Z
- A standard delimiter, such as LINE FEED

Private delimiters are useful on data entry terminals with a specialized keyboard where a large or conveniently located key can be defined as the delimiter key. They are also useful in keypad applications.

A private delimiter has basically the same characteristics as a standard delimiter. Like a standard delimiter, it

- Terminates a read operation.
- Cannot be deleted. The DELETE key and Control-U do not affect private delimiters in the type ahead buffer.
- Causes the system to awaken a sleeping job when typed at a terminal that the job has open or assigned. If the job cannot be awakened, the terminal service stores the delimiter character.

Unlike a standard delimiter, however, private delimiters change the way characters are processed in binary mode. If a terminal is open in binary mode and no private delimiters are in use, the terminal service terminates a read after every character. However, if one or more private delimiters are in use, terminal service terminates a read only when a private delimiter is typed.

Declaring a character as a private delimiter overrides the existing ASCII code for the character. Unlike a standard delimiter such as carriage return (CR) or line feed (LF), a private delimiter does not echo at the terminal. In addition, a special character used as a private delimiter no longer performs its normal function. For example, when the DELETE key is used as a private delimiter, it does not erase the last character typed.

All RSTS/E systems permit user programs to declare at least one private delimiter. An additional system generation option permits the use of multiple (up to 256(10)) private delimiters in programs written in MACRO. (BASIC-PLUS programs can use only one private delimiter.) Multiple private delimiters allow programs to do special character processing without having to do single character I/O.

Multiple delimiters are specified through means of a bit mask of up to 32(10) bytes. Each bit in the mask represents an ASCII character; setting a bit indicates that the associated character is to serve as a private delimiter. (See the *RSTS/E System Directives Manual* for a table showing the bit-to-ASCII correspondence.) When multiple private delimiters are in effect for a terminal, the bit mask is stored in a small buffer from the general pool, pointed to by a word in the variable portion of the terminal's device data block (see Section 13.1.2.2).

A single private delimiter is established using the "set terminal characteristics" system directive. Multiple delimiters are set using the special I/O function (.SPEC). Once set, private delimiters remain in effect for a terminal until cleared by the program, or until the job releases the terminal by deassigning it or closing the I/O channel on which the terminal is open. In addition, terminal service clears private delimiters when a dial-up line is hung up or when the job controlling the terminal is killed.

### 12.2.6 Binary Input

When a terminal is opened with binary input mode enabled, the terminal service passes typed data directly to the program without alteration or processing. Character echoing is suppressed and standard delimiters are neither required nor recognized. In this mode, each input request from the program normally causes the terminal service to transfer all data currently available from the keyboard, up to the specified user buffer size. However, if one or more private delimiters are set for the terminal, terminal service terminates a read only when a private delimiter is typed. If no data is available, the program is stalled.

Since terminal service accepts all data typed at the terminal as input, none of the standard control characters are processed, including XOFF, XON and Control-C. For this reason, binary input mode should be used with care. It is intended for use only by applications where it is essential that unprocessed data be returned to the program. Other features of this mode, such as suppression of echoing and elimination of the need for line delimiters, can be obtained through the use of the "special I/O" function or judicious use of other OPEN modes.

### 12.2.7 Conditional Input

If no keyboard input is available when a user program issues an input request, the system will normally stall the program until data is available.* However, it is sometimes undesirable for a program to be stalled in this way. When this is the case, the user program can issue a conditional request for input. Then, if data is not available, the system returns an error code to the program's file request queue block (FIRQB) but does not stall the job.

### 12.2.8 Gagging a Terminal

During normal timesharing operations, terminals can receive broadcast messages of concern to the job to which they are attached. For example, when new MAIL arrives for a user, a notification is typed on the console terminal of every job logged into that user account. Occasionally, however, this procedure can be undesirable, such as when the terminal is being used to print a document that would be corrupted by such output. When this is the case, the terminal can be "gagged," thereby suppressing any broadcasted messages.

A terminal can be gagged or ungagged under program control through use of the "set terminal characteristics" system directive. A user can also use the GAG/NOGAG commands of the TTYSET system utility program.

### 12.2.9 Suppression of Automatic Carriage Return

One of the values maintained within each terminal device data block is the width of the terminal print line. Normally, terminal service will automatically generate a carriage return/line feed sequence when that number of printing characters has been output and echoed since the last carriage return. In screen-oriented applications, however, this procedure can be inappropriate. Using direct cursor addressing, it is possible for a user program to generate an indefinite stream of output in which a carriage return is neither necessary nor desirable.

By opening the terminal for output using the special "suppress automatic carriage return" OPEN mode, a user program can suppress this normal processing.

### 12.2.10 Guarded Mode

Under normal system operation, there are two ways in which the execution of a user program can be interrupted due to "outside" circumstances.

* The availability of data is normally determined by the presence of a line delimiter in the keyboard input buffer. In binary mode or ODT submode, *any* characters in the buffer constitute available data.

- When a Control-C is encountered in the terminal input buffer, control transfers to the job's run-time system. The program has no way to recover.
- When a job becomes detached* and then attempts to do terminal I/O on a nonzero channel, it enters a state known as "hibernation." The job must wait until it is reattached (through some external process) before it can become runnable and recover.

There are times when either one or both of these procedures is inappropriate. Thus, to prevent interruption of program execution in either of these two circumstances, the user program can open the terminal using the special "guarded" OPEN mode. Then, if a Control-C is encountered in the input buffer, the terminal service cancels any pending output to the terminal and passes the Control-C character (as an ASCII value 3) to the program. If the terminal becomes detached, the terminal service loads an error code into the job's file request queue block (FIRQB). In both cases, the job is made runnable, at which point it can examine the input or error code and continue processing.

### 12.2.11 Processing of Incoming XOFF/XON

When a terminal opened in nonbinary mode is also opened with the special "enable incoming XOFF/XON" mode, no data is returned to the program. The terminal service does, however, process the XOFF and XON synchronization characters: an incoming XOFF suspends output to the terminal and an incoming XON resumes output to the terminal. This is useful for a terminal, such as an LA120 DECwriter, that is being used strictly as an output device. It is also an alternate way to keep a Control-C from interrupting a user program.

When the "incoming XOFF/XON" mode is combined with binary mode, incoming XOFF and XON characters are processed as stated above and all other input is passed to the user program without alteration or processing. This combination of OPEN modes should be used when a user program wishes to receive binary input from a terminal, such as a VT100, that depends upon the XOFF/XON synchronization protocol to prevent garbled output.

### 12.2.12 Terminal Special Function (.SPEC)

Under RSTS/E, a special I/O system directive (.SPEC) is reserved for use with a variety of devices. When issued for a terminal, this special function permits nine subfunctions. These subfunctions are described in the following subsections.

#### 12.2.12.1 Cancel Control-O-

When a Control-O is encountered in a terminal input stream, the terminal service automatically discards any further output sent to the terminal by the program. This continues until another Control-O is encountered in the input stream. By using the "cancel Control-O" subfunction of the terminal special I/O function, the program itself can cancel the effect of Control-O and resume physical output to the terminal.

* A job becomes detached when it executes the "detach" system directive or when it is running on a dial-up line that gets disconnected.

### 12.2.12.2 Set Tape Submode -

The special "tape" submode is available specifically for use with ASR33 terminals that have a low speed paper tape reader. When this mode is enabled through the "set tape mode" subfunction of the terminal special I/O function, echoing on the terminal is also disabled.

### 12.2.12.3 Disable Echo -

By using the "disable echo" subfunction of the terminal special I/O function, the program can disable the normal character echoing performed by the terminal service.

### 12.2.12.4 Enable Echo and Clear Tape Submode -

This subfunction is used to disable the special tape mode used to read the low speed paper tape reader of an ASR33 terminal. It also reenables echoing at the terminal.

### 12.2.12.5 Enable ODT Submode -

ODT submode allows input to be read from the terminal without the use of delimiters.

Normally, the terminal service waits to transfer input data to the user until a line is terminated by either a private delimiter or by one of the standard delimiters (carriage return, line feed, form feed, escape, or a Control-D). In ODT submode, however, terminal service does not wait for a delimiting character. Instead, any characters typed at the terminal are passed to the program immediately on the next keyboard input request. This input mode is called ODT submode because it is used in the system debugging programs.

### 12.2.12.6 Cancel All Type Ahead -

When a user enters input at the terminal keyboard faster than the system can process it, the terminal service stores the extra input to wait for processing. Sometimes, however, previously processed input can make such "type ahead" data obsolete. For example, an error in the previously accepted data might result in the program aborting an entire segment of processing. In this circumstance, the program can cancel all type ahead data and clear the input buffer by using the "cancel all type ahead" subfunction of the terminal special I/O function.

### 12.2.12.7 Force Input to Keyboard -

Using the "force input to keyboard" subfunction of the terminal special I/O function, a privileged MACRO program can force data to another job's terminal. The data specified is inserted in the pending input buffer and looks exactly like incoming keystrokes to the terminal's owner job. (This subfunction is not available to BASIC programs.)

### 12.2.12.8 Broadcast Output to Terminal -

Using the "broadcast output to terminal" subfunction of the terminal special I/O function, a privileged MACRO program can send an output message to another job's terminal. The data specified will interrupt and interleave with any output currently being sent to the display unless the receiving terminal is gagged. If the terminal is gagged, the broadcast data is discarded. (This subfunction is not available to BASIC programs.)

#### 12.2.12.9 Private Delimiter Subfunction -

A MACRO program can set, clear or read the current set of private delimiters by using the "private delimiter" subfunction of the terminal special I/O function. A BASIC program can use this subfunction to clear its single private delimiter.

### 12.2.13 FMS Support

The Forms Management System (FMS-11) is a DIGITAL software product that contains the tools for developing forms-oriented applications to run on VT100 scope terminals. It is sold as an optional product for RSTS/E and other operating systems. It is not part of the standard RSTS/E distribution kit.

FMS normally validates and echoes each character as it is typed. However, since FMS also runs as a resident library attached to the user program, this process can be quite "expensive" under RSTS/E. Performance analysis has shown that user echoing of input places an inordinate load on the overall system. Thus, to support FMS while maintaining reasonable system performance, many of the character and field validating functions are performed within the RSTS/E monitor. Thus, if a RSTS/E installation is to use FMS, the terminal service code to support it must be selected as part of the system generation process.

### 12.2.14 RSX Mode Terminal I/O

RSTS/E permits MACRO programs running under the RSX run-time system to use a subset of the RSX-11M system directives. Formerly, however, the RSTS/E emulation of the QIO$ and QIOW$ directives was not identical to the behavior of the same directives under RSX-11M, due to inherent differences between the terminal handlers of the two systems. The major difference was in how the handlers echo a carriage return keystroke typed at the keyboard. In RSTS/E, the handler echoed a carriage return/line feed combination. In RSX-11M, the handler echoes only a carriage return.

This difference presented a problem for programmers using QIO$ or QIOW$ in programs to be run on both systems. The problem arose for programs doing a read operation followed by a write operation using the null vertical format control character. Under RSX-11M, such write operations must include a leading line feed character. If this character is missing, the remainder of the data will overwrite the characters typed by the user. Under RSTS/E, however, the leading line feed is not necessary since the terminal service echoes one to the terminal following the input carriage return.

Code in the RSTS/E terminal service resolved this problem. When a QIO$ or a QIOW$ is executed, the RSX emulator tells the terminal service to enter "RSX mode" on that particular terminal. In this mode, terminal service echoes typed carriage returns as they are echoed under RSX-11M — without a following line feed. However, if the user types more characters before any programmed read request (type ahead), a line feed is echoed to the terminal ahead of the typed characters. Thus, the type ahead will not overprint the previously typed line. Furthermore, when the terminal service is in RSX mode, it prints a line feed whenever a read or write is requested on the terminal.

RSX mode remains in effect for the terminal until the program issues any non-RSX read or write request.

# CHAPTER 13
# Terminal Service Structure [V8.0]

Terminal service encompasses two separate device drivers: one for standard terminal devices (device type "KB") and one for pseudo-keyboards (device type "PK"). Each driver follows the standard device driver conventions and rules of structure as described in Chapter 6.

Both drivers contain three of the four possible program sections (PSECTs). The driver code PSECTs (KBDVR and PKDVR) are required and contain all processing code and noninterrupt level entry points. The interrupt dispatch PSECTs (KBDINT and PKDINT) are also required and contain the code necessary to map and dispatch to the interrupt processing routines in the driver code PSECT. The pseudo-keyboard driver does not service a physical interrupt-driven device and, therefore, does no real interrupt processing. Nevertheless, PKDINT is defined (as an empty PSECT) to conform to the standard driver conventions.

The read/write data PSECT (xxDCTL) is optional and is generally included when the driver must maintain general read/write data apart from the device-specific data contained in the device data blocks. Both the KB driver and the PK driver include this PSECT. Program section KBDCTL contains job counters that must be available to other system components (see Section 13.1.5). It also contains the queue pointers used by the Forms Management System (FMS-11) support code (if that option was included during system generation) and the error logging control table. Program section PKDCTL contains dummy control and status registers (CSRs) for compatibility with the terminal driver's common (single line) interface code.

The remainder of this chapter discusses the general structure of terminal service. Section 13.1 discusses the primary data structures used. Section 13.2 presents an overview of the processing of major terminal service functions, including those that are device-related as well as those subfunctions pertaining to job creation and maintenance.

## 13.1 Data Structures

The primary control structure for terminal devices is the device data block (DDB). DDBs are used by the terminal service to control the actual use of a device and to communicate with the common monitor support routines. There is one DDB for every individual terminal unit (including pseudo-keyboards) configured on the system. DDBs reside in the read/write portion of the monitor's permanently mapped root.

In addition to the device data blocks, appropriate entries are maintained in the monitor tables required to provide generalized function processing within the monitor above the driver level.

### 13.1.1 Keyboard Numbers

A RSTS/E system can handle a maximum of 128(10) terminals, including at least one pseudo-keyboard. An installation can have any combination of local and remote line interfaces as long as the total number of terminal lines and pseudo-keyboards does not exceed 128.

During system generation, each terminal is assigned a keyboard number in the range of 0 to 127(10). The console terminal is assigned keyboard number 0 and is referred to by the designator "KB0". The type of line interface that connects a terminal (other than the console terminal) to the system establishes its keyboard number. Keyboard numbers are assigned in the following order:

- The console terminal
- All DL11A and DL11B lines
- All DL11C and DL11D lines
- All DL11E (remote dial) lines
- All pseudo-keyboards
- All DJ11 lines
- All DH11 and DZ11 lines in increasing order of unit number and increasing order of configured lines on each unit
- All DHV11 and DHU11 lines in increasing order of unit number and increasing order of configured lines on each unit

### 13.1.2 Standard Terminal Structures

There is one device data block format for all standard terminal devices, regardless of type. The first 17(10) words of the DDB are fixed, while the remaining data items vary from system to system, depending on the types of terminals available and the optional features supported.

#### 13.1.2.1 Device Data Block (Fixed Portion) -

Figure 13-1 shows the format of the fixed portion of the terminal DDB. Note that the first nine words of this section conform to the standardized format used by all devices. (See Section 6.3 for the general details of this part of the DDB.) Terminal-specific information for each item is discussed below.

*Offset* *Contents*

DDIDX This byte contains the driver index for the device. The driver index for physical terminals is two.

DDSTS This byte contains a set of flags describing certain terminal characteristics of interest to the general purpose monitor routines. There are only two bits of concern for terminals:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | DDPRVO | If this bit is set, ownership of the terminal is reserved to privileged users. |
| 15 | DDSTAT | If this bit is set, a Control-O was typed at the keyboard, causing terminal output to be discarded. This bit is not actually used by the monitor routines but is cleared whenever the device is closed. |

DDJBNO This byte contains the job number (multiplied by two) of the user job that currently owns the terminal.

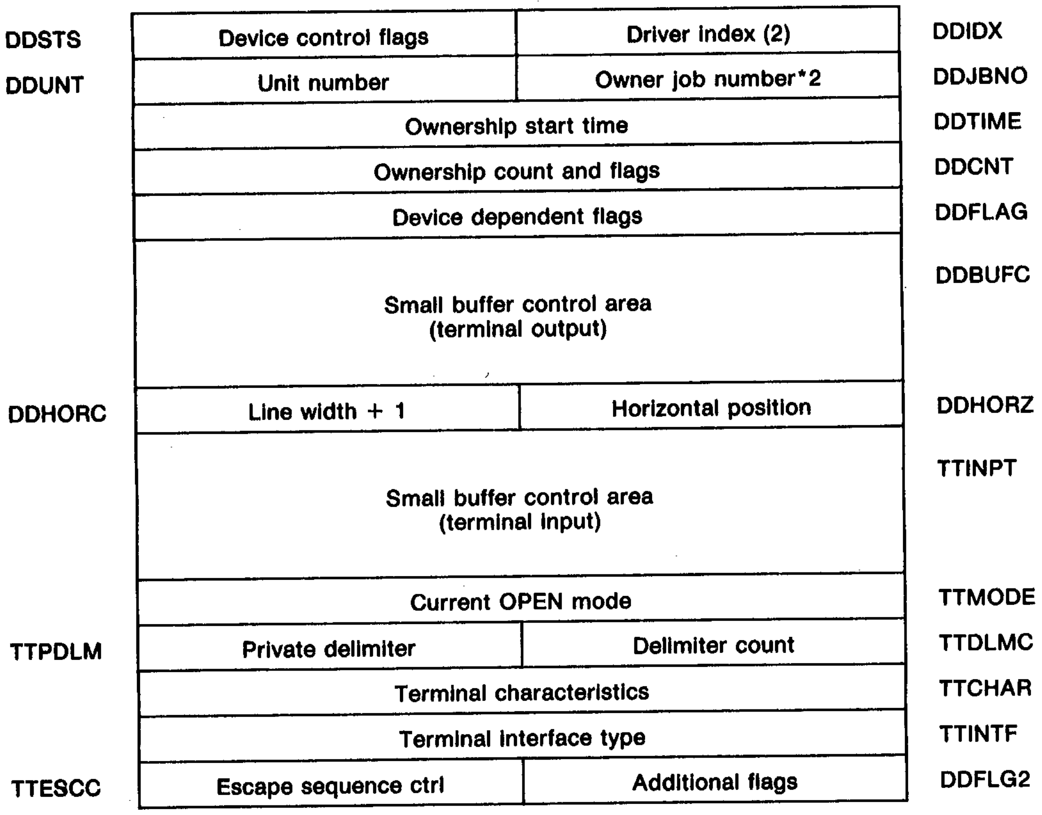


**Figure 13-1:** Fixed Portion of Terminal Device Data Block

DDUNT This byte contains the unit number of the terminal.

DDTIME This word contains the system time at which the terminal was assigned to the job indicated at DDJBNO.

DDCNT This word controls access to the device. If ownership of the device is implicit (through an OPEN), the low-order byte contains a count of the number of users that have the device open. (This is never greater than one for terminals.) If ownership of the device is explicit (through an ASSIGN), the low-order byte is zero, but one or more flags are set in the high-order byte. The setting of these flags is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8-12 | | (Unused) |
| 13 | DDCONS | This bit indicates that the device is the owning job's console terminal. It is used to differentiate a terminal owned by the job for general purpose I/O and the terminal on which the job was created. |
| 14 | DDUTIL | This bit indicates that the device is temporarily assigned to the file processor. It is never set for terminal devices. |

| | | |
|---|---|---|
| 15 | DDASN | This bit indicates that the device has been explicitly assigned to the job through use of the ASSIGN command. |

DDFLAG This word contains flags concerning the current state of the terminal. These flags are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | TTLFRC | This bit is on after a force/broadcast, off after a Control-C. |
| 1 | TTNBIN | If this bit is set, the next character is to be output in binary mode. |
| 2 | TTRSX1 | This bit is the RSX mode low-order bit. The high-order bit is bit 8 (TTRSX2). If either of these two bits is set, the terminal is in RSX mode (see Section 12.2.14). |
| 3 | TTDDT | If this bit is set, the terminal is currently in ODT submode (see Section 12.2.12.5). |
| 4 | TAPE | If this bit is set, the terminal is currently in ASR33 tape mode (see Section 12.2.12.2). |
| 5 | NOECHO | If this bit is set, the system is not echoing input from the terminal. |
| 6 | LCLCPY | If this bit is set, input is echoed locally at the terminal, not by the software. |
| 7 | | (unused) |
| 8 | TTRSX2 | This bit is the RSX mode high-order bit. The low-order bit is bit 2 (TTRSX1). When either of these bits is set, the terminal is in RSX mode (see Section 12.2.14). When this bit is set, terminal service will output a line feed before printing another character. |
| 9 | TTHUNG | If this bit is set, a terminal "hang" is pending. That is, the terminal has timed out, but the software is attempting to "unhang" it (see Section 13.2.1.4). |
| 10 | TTDFIL | If this bit is set, any required fill characters will be delayed until after the next output character. |
| 11 | TTMSG | If this bit is set, the software is processing an incoming message over a synchronous interface. |
| 12 | RUBOUT | If this bit is set, the software is processing a string of rubouts on a hardcopy terminal. |
| 13 | TTSXOF | If this bit is set, the software must send an XOFF to the terminal, as soon as possible, to suspend further input. |
| 14 | TTXOFF | If this bit is set, the software has sent an XOFF to the terminal. |
| 15 | TTSTOP | If this bit is set, output to the terminal has been temporarily suspended. |

DDBUFC This 3-word area is the small buffer control block required for use of the monitor routines that store and process characters in small buffers from the general pool. This buffer control area is for use with terminal output. (See Section 5.3.3 for a discussion of monitor buffer usage and the format of this 3-word buffer control area.)

DDHORZ This byte contains the terminal's current horizontal position.

DDHORC This byte contains the terminal's line width, plus 1.

TTMODE This word contains a set of bit flags, specifying the mode(s) in which the terminal is currently open. The setting of these flags is defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | TTBIN | Binary input is enabled from the terminal. |
| 1 | TTTECO | (Reserved for TECO) |
| 2 | TTCRLF | Carriage return/line feed is not automatically generated at the right margin. |
| 3 | TTECTL | Echo control is enabled. |
| 4 | TTGARD | The program is guarded against Control-C interruption and dial-up line hibernation. |
| 5 | TTPCOL | Special incoming XOFF/XON processing is enabled. |
| 6 | TTFMS | (Reserved for FMS) |
| 7 | TTTECS | Special scope RUBOUT processing is enabled. |
| 8 | TTESEQ | Escape sequence processing is enabled. |
| 9-13 | | (Unused) |
| 14 | TTCTLC | Transparent control character mode is enabled. |
| 15 | | (Unused) |

TTDLMC This byte contains a count of the number of input delimiters (both standard and private) that are currently pending in the input buffer. It is used to provide a quick check to determine whether a normal user input request can be satisfied.

TTPDLM This byte contains the user's first private delimiter (if any). If no delimiter is set, this byte contains a zero. If the system supports the optional feature of mulitple private delimiters, additional delimiters are specified using a bit mask pointed to by the word at offset TTDLMM (in the variable portion of the DDB).

TTCHAR This word contains a set of bit flags describing characteristics of the terminal. These bits are set or cleared through the use of the TTYSET system utility or the "set terminal characteristics" monitor directive. These flags are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | TTGAG | If this bit is set, broadcast messages are not output to the terminal. |
| 1 | TTBRK | If this bit is set, the BREAK keystroke is treated as a null; otherwise, BREAK is translated to and treated as a Control-C. |
| 2 | TTXANY | If this bit is set, the XOFF/XON processing will resume output whenever *any* character is typed after the XOFF is received; otherwise, output is resumed only after an XON or a Control-C are typed. |
| 3 | TTFUNC | If this bit is set, the special Control-R and Control-T processing are disabled. |
| 4 | TTESC | If this bit is set, the terminal has an ESC key that generates an actual escape character (ASCII value 27(10)); otherwise, the ALTMODE (}) and PREFIX (~) keystrokes are translated into (and treated as) escape characters. |
| 5 | TTSCOP | If this bit is set, the terminal is a scope, or CRT (cathode ray tube) display device, and has the following features:<br>• The terminal responds to the hardware XOFF/XON synchronization protocol.<br>• The system processes the DELETE keystroke by outputting a backspace, followed by a space, followed by another backspace. |
| 6 | TTESCI | If this bit is set, the system treats an incoming escape character (ASCII value 27(10)) and any following characters as a special control sequence; otherwise, an incoming escape is treated as a line terminator and is echoed as the $ character. |
| 7 | TTLCOU | If this bit is set, the terminal can receive and print lowercase characters; otherwise, the system translates lowercase characters (ASCII values 64 through 94(10)) into uppercase before output. |
| 8 | TTPODD | If this bit is set, the system generates an output parity bit in the odd parity format; otherwise, the parity bit conforms to the even parity format. (Note that this bit has meaning only when bit 9, TTPRTY, is set.) |
| 9 | TTPRTY | If this bit is set; the system generates and sends an output parity bit in the format specified by TTPODD (bit 8); otherwise, no parity bit is generated. |
| 10 | TTUPAR | If this bit is set, the system echoes control characters as the uparrow (^) character followed by the equivalent printable character. For example, the Control-D character is echoed as ^D. |

| | | |
|---|---|---|
| 11 | TTSYNC | If this bit is set, the system will stop sending output to the terminal if an XOFF character is received. |
| 12 | TTXON | If this bit is set, the terminal will stop sending input to the computer if the system outputs an XOFF character. |
| 13 | TTFORM | If this bit is set, the terminal has the hardware to process form feeds and vertical tabs; otherwise, the system performs these operations by outputting four line feed characters. |
| 14 | TTTAB | If this bit is set, the terminal has the hardware to process horizontal tabs; otherwise, the system simulates this feature by outputting spaces to align the tab positions every eight characters, beginning at position 1. |
| 15 | TTLCIN | If this bit is set, the terminal can transmit the full ASCII character set and lowercase characters (ASCII values 64 through 94(10)) are *not* translated to the their uppercase equivalents before being stored in the input buffer. |

TTINTF This word contains the specification for the interface being used for this terminal. The format of this word is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-7 | | This byte contains a code indicating the type of interface. If this keyboard unit is the KB side of a pseudo-keyboard, this byte contains the value TTPK11=10. |
| 8-11 | TTSUBL | These four bits specify the subline number if the terminal interface handles multiple lines. |
| 12 | TTRDRN | If this bit is set, the (single line) interface is ready to accept a character. |
| 13 | TTHRPT | If this bit is set, the interface hardware generates the necessary output parity bit; otherwise, the terminal service software is responsible for parity processing, as specified in the TTPRTY and TTPODD bits of TTCHAR. |
| 14 | TTMODM | If this bit is set, the communications line is controlled by a modem. |
| 15 | TTMUX | If this bit is set, the terminal interface handles multiple lines. |

DDFLG2 This byte contains additional flags used by the terminal service. These bits are defined as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | FMSFLD | If this bit is set, the current FMS field is active. |
| 1 | FMSRDY | If this bit is set, the current FMS field is ready for processing. |

2 TTESCO If this bit is set, the terminal service is in the process of scanning an escape sequence on output.

TTESCC This byte is used to store the sequence counter during escape sequence processing.

### 13.1.2.2 Device Data Block (Variable Portion) -

Figure 13-2 shows the format of the variable portion of the terminal device data block. The length and content of this portion vary depending on the types of terminals available and the optional features supported. However, within any given system, the format of this section is the same for all terminal devices, regardless of whether or not a particular data item is relevant to an individual terminal.

*Offset* *Contents*

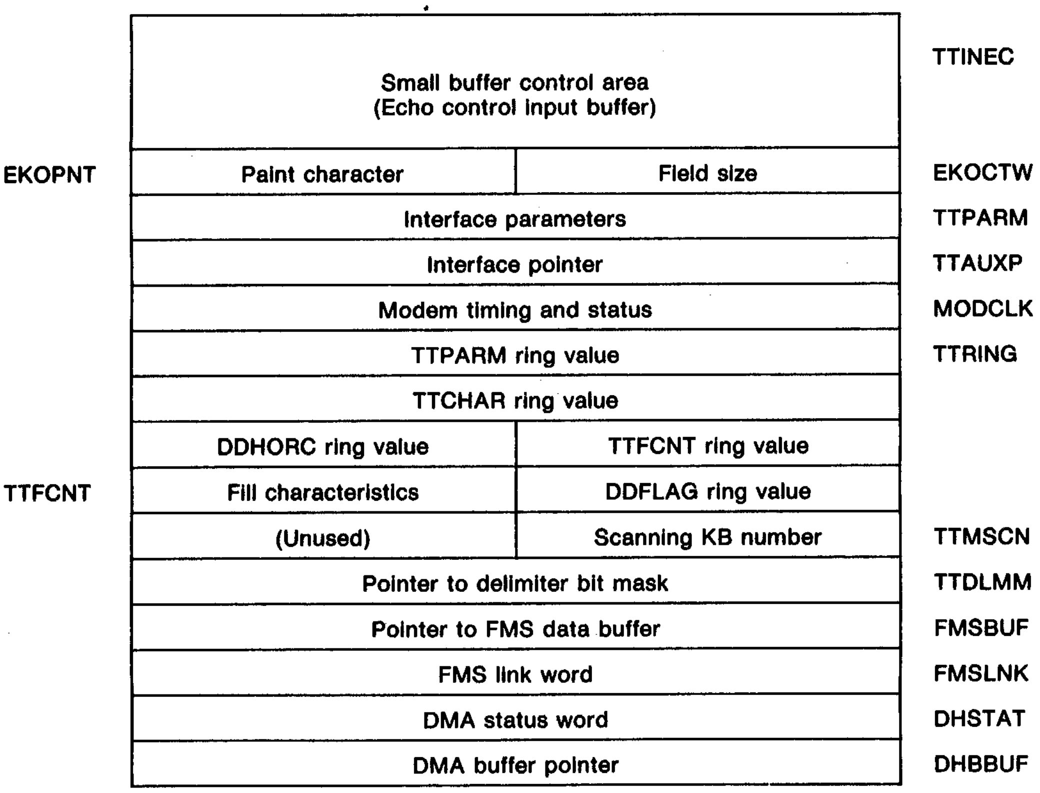
TTINEC If the system supports echo control or FMS processing, this 3-word area contains the small buffer control block for the auxiliary input buffer used to hold user type ahead. (Small buffer control blocks are required for use of the monitor routines that store characters in small buffers from the general pool. See Section 5.3.3 for a discussion of monitor buffer usage and the format of this 3-word control area.)

EKOCTW If the system supports echo control processing, this byte is used to store the current field size.

EKOPNT If the system supports echo control processing, this byte is used to store the 7-bit "paint" character. In addition, the sign bit of this byte is set to indicate that a field is currently active; if the sign bit is not set, no field is currently active.

TTPARM This word contains interface characteristics (such as speed, character length, and parity information), and its format is dependent on the particular type of interface used for this terminal. This word is configured into the DDB whenever the system supports any pseudo-keyboards or any physical terminals other than the console terminal. (Note, therefore, that since every system supports at least one pseudo-keyboard, this word is always present.)

TTAUXP If the system supports the DM11BB interface (and this terminal is attached to one of that type of interface), this word is used to store any auxiliary pointer that might be necessary for processing.



| | | | |
|---|---|---|---|
| | Small buffer control area<br>(Echo control input buffer) | | TTINEC |
| EKOPNT | Paint character | Field size | EKOCTW |
| | Interface parameters | | TTPARM |
| | Interface pointer | | TTAUXP |
| | Modem timing and status | | MODCLK |
| | TTPARM ring value | | TTRING |
| | TTCHAR ring value | | |
| | DDHORC ring value | TTFCNT ring value | |
| TTFCNT | Fill characteristics | DDFLAG ring value | |
| | (Unused) | Scanning KB number | TTMSCN |
| | Pointer to delimiter bit mask | | TTDLMM |
| | Pointer to FMS data buffer | | FMSBUF |
| | FMS link word | | FMSLNK |
| | DMA status word | | DHSTAT |
| | DMA buffer pointer | | DHBBUF |

**Figure 13-2:** Variable Portion of Terminal Device Data Block

MODCLK If the system supports modem control (and this terminal is attached to a line controlled by a modem), this word contains the modem timer and status bits. The format of this word is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-7 | | These eight bits (the low byte) are used to store the modem timer. |
| 8-13 | | (Unused) |
| 14 | TTWRC | If this bit is set, the system is waiting for carrier to be established over the line. |
| 15 | TTDSBL | If this bit is set, the line is disabled and a request for connection via dial-up will not be honored. |

TTRING If the system supports modem control (and this terminal is attached to a line controlled by a modem), this 9-byte area contains "ring values" for the terminal. Ring values are the default characteristics to be assigned to a terminal when a connection is initiated on a dial-up line (see Section 12.1.1.4). Default values are provided for TTPARM, TTCHAR, TTFCNT, DDHORC, and the low byte of DDFLAG.

TTFCNT This byte is always configured into the DDB. Its format is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-2 | | These three bits specify the fill factor for the terminal. The fill factor is the base number to be used in calculating the number of NULL characters that must be output to provide enough time for certain terminal operations to complete (see Section 12.1.2.7). |
| 3 | TT8BIT | If this bit is set, the terminal supports 8-bit characters. |
| 4-7 | | (Unused) |

TTMSCN If the system supports multiple terminals open on one I/O channel (and this terminal is the master terminal on a channel), this byte is used to store the number of the last terminal checked for pending input.

TTMSCN+1 This byte is configured into the DDB only when necessary to ensure that the following data items begin on a word boundary. It is unused.

TTDLMM If the system supports multiple private delimiters, this word contains a pointer to the bit mask of private delimiters currently in use on this terminal (see Section 12.2.5).

FMSBUF If the system provides support for the Forms Management System (FMS), this word contains a pointer to the FMS data buffer.

FMSLNK If the system provides support for the Forms Management System (FMS), this word contains a link to the next DDB in the FMS queue.

DHSTAT This word is used for internal diagnostics.

DHBBUF This word is used for internal diagnostics.

#### 13.1.2.3 Monitor-Level Specifications -

All standard physical terminals under RSTS/E are considered to be of one device type -- namely, keyboards. Thus, within the monitor support tables that specify generalized device characteristics, there is only one entry for all physical terminals. These support tables reside in the read/write portion of the monitor's permanently mapped root.

*Device Dependent Flags (FLGTBL)*

Each entry in the monitor table of device dependent flags (FLGTBL) contains the device driver index (in the low byte) and a set of characteristics flags (in the high byte) for the associated device. For keyboard devices, the driver index is two, and the following characteristics flags are set:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | DDNFS | The device is non-file-structured and does not require a file name for access. |
| 9-10 | | (unused) |
| 11 | FLGPOS | The device driver maintains the current horizontal position for the device. |

12 FLGMOD The device accepts modifiers on read and write requests.

13 FLGFRC The device is byte-oriented and does not require that a specific number of characters be transferred on each I/O request.

14 FLGKB The device is a keyboard.

*Line Width (SIZTBL)*

Each entry in the monitor line width table (SIZTBL) contains the maximum horizontal position of the associated device type. The entry for keyboard devices contains a zero, indicating that the line width is variable and is maintained by the driver in byte DDHORC of the device data block.

*I/O Buffer Size (BUFTBL)*

Each entry in the monitor table of device buffer sizes (BUFTBL) contains the default buffer size to be used when the associated device is opened with no specified buffer size. The default buffer size for keyboard devices is 128(10) bytes.

*JBWAIT/JBSTAT Status Bits*

When a job is stalled waiting for an I/O operation to complete, a bit is set in the job's JBWAIT entry, making the job unrunnable. When the operation completes, the corresponding bit is set in the job's JBSTAT entry, thereby making the job runnable. (See Section 3.2 for a complete discussion of job scheduling.)

When a job is stalled waiting for I/O from a keyboard device, the input and output functions are treated separately. A job waiting for input from any physical terminal will have bit 1 (JS.KB = 2) set in its JBWAIT entry; a job waiting for output will have bit 11 (JSTEL = 4000) set.

### 13.1.3 Pseudo-Keyboard Structures

Under RSTS/E, a pseudo-keyboard is treated as a separate device type from a physical terminal. A pseudo-keyboard has a different driver index and a different format device data block.

#### 13.1.3.1 Device Data Block -

The format of a pseudo-keyboard device data block is different from that of a standard physical terminal. The DDB of a pseudo-keyboard contains of only six words. The first five words are the standard DDB portion required of all nondisk devices, while the remaining word contains flags used strictly by the pseudo-keyboard handler.

Figure 13-3 shows the format of the pseudo-keyboard device data block. Following the figure is a description of each entry.

Note that while a pseudo-keyboard has an input and an output buffer (both of which come from the small buffer pool), there are no small buffer control areas configured into the DDB. This is because the input and output buffers of the pseudo-keyboard are, in reality, the output and input buffers, respectively, of the associated keyboard unit. Thus, the small buffer control areas for the buffers are contained in the device data block of the associated keyboard unit.

| | | | |
|---|---|---|---|
| DDSTS | Device control flags | Driver index (6) | DDIDX |
| DDUNT | Unit number | Owner job number*2 | DDJBNO |
| | Ownership start time | | DDTIME |
| | Ownership count and flags | | DDCNT |
| | Actual keyboard number * 2 | | DDFLAG |
| PKINWT | "Input wait" flag | OPEN mode | PKMODE |

**Figure 13-3:** Pseudo-Keyboard Device Data Block

*Offset* *Contents*

DDIDX This byte contains the driver index for the device. The driver index for pseudo-keyboards is six.

DDSTS This byte contains a set of flags describing certain terminal characteristics of interest to the general purpose monitor routines. There is only one bit of concern for pseudo-keyboards:

*Bit* *Symbol* *Meaning*

8 DDPRVO If this bit is set, ownership of the pseudo-keyboard is reserved to privileged users.

DDJBNO This byte contains the job number (multiplied by two) of the job that currently owns the pseudo-keyboard (the controlling job).

DDUNT This byte contains the unit number of the pseudo-keyboard.

DDTIME This word contains the system time at which the pseudo-keyboard was assigned to the job indicated at DDJBNO.

DDCNT This word controls access to the device. If ownership of the device is implicit (through an OPEN), the low-order byte contains a count of the number of users that have the device open. (This value is never greater than one for terminals.) If ownership of the device is explicit (through an ASSIGN), the low-order byte is zero, but one or more flags are set in the high-order byte. The setting of these flags is as follows:

*Bit* *Symbol* *Meaning*

8-12 (Unused)

13 DDCONS This bit indicates that the device is the owning job's console terminal. It is never set for pseudo-keyboards.

14 DDUTIL This bit indicates that the device is temporarily assigned to the file processor.

15 DDASN This bit indicates that the device has been explicitly assigned to the job through use of the ASSIGN command.

DDFLAG This word contains the unit number of the keyboard device associated with this pseudo-keyboard.

PKMODE This byte indicates the mode in which the pseudo-keyboard is OPEN. There are two possible modes:

| *Mode* | *Meaning* |
|---|---|
| 0 | If the pseudo-keyboard is opened in this mode, the system will kill the job running on the pseudo-keyboard (the controlled job) when the pseudo-keyboard is closed. |
| 1 | If the pseudo-keyboard is opened in this mode, the system will detach the controlled job when the pseudo-keyboard is closed. |

PKINWT This byte is used as a flag to indicate that the job running on the pseudo-keyboard (the controlled job) has entered an input wait state.

### 13.1.3.2 Monitor-Level Specifications -

There is one entry for all pseudo-keyboards within the monitor support tables that specify generalized device characteristics. These support tables reside in the read/write portion of the monitor's permanently mapped root.

*Device Dependent Flags (FLGTBL)*

Each entry in the monitor table of device dependent flags (FLGTBL) contains the device driver index (in the low byte) and a set of characteristics flags (in the high byte) for the associated device. For pseudo-keyboards, the driver index is six, and the following characteristics flags are set:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | DDNFS | The device is non-file-structured and does not require a file name for access. |
| 9-11 | | (unused) |
| 12 | FLGMOD | The device accepts modifiers on read and write requests. |
| 13 | FLGFRC | The device is byte-oriented and does not require that a specific number of characters be transferred on each I/O request. |
| 14 | FLGKB | The device is a keyboard. |

*Line Width (SIZTBL)*

Each entry in the monitor line width table (SIZTBL) contains the maximum horizontal position of the associated device type. The entry for pseudo-keyboards contains the value 5*14.+1, indicating that a line width does not apply.

*I/O Buffer Size (BUFTBL)*

Each entry in the monitor table of device buffer sizes (BUFTBL) contains the default buffer size to be used when the associated device is opened without a buffer size being specified. The default buffer size for pseudo-keyboards is 128(10) bytes.

*JBWAIT/JBSTAT Status Bits*

When a job is stalled waiting for an I/O operation to complete, a bit is set in the job's JBWAIT entry, making the job unrunnable. When the operation completes, the corresponding bit is set in the job's JBSTAT entry, thereby making the job runnable. (See Section 3.2 for a complete discussion of job scheduling.) When a job is stalled waiting for I/O from a pseudo-keyboard, bit JS.PK is set in its JBWAIT entry.

### 13.1.4 Device Data Block for Detached Jobs

As discussed in Section 3.1.4, the address of the device data block for each device a job has open is stored in the job's I/O block (IOB). If an entry in the IOB is zero, that particular channel is closed and does not have an associated device open on it. Channel 0 is a special case. It is always open and contains the address of the device data block for the job's console terminal.

When a job detaches from a terminal, the IOB entry is not cleared but the job number in the device data block is set to zero, disassociating the terminal and the job. Later, if the job attempts to do I/O on the terminal, it will be placed in a state known as "hibernation." At that point, the job must wait until it is reattached (through some external process) before it can become runnable and recover.

If, however, the terminal was opened (on a nonzero channel) in "guarded" mode (see Section 12.2.10), terminal service follows a slightly different procedure in order to flag the fact that the job should not be made to hibernate. In this circumstance, the job number in the terminal DDB is cleared and the IOB entry is loaded with the address of a dummy device data block. This address is then used as a flag that the terminal is in a "guarded" detached state.

Figure 13-4 shows the format of this dummy device data block. Note that the driver index is two (the index for keyboards) but all other entries are zero. There are no special flags and the unit number is zero. Also, since there is no owner job, the job number (multiplied by two) is zero, and there is no owner start time or count.

| | | |
|---|---|---|
| Device ctrl flags (0) | Driver index (2) | KBFDDB |
| Unit number (0) | Owner job number*2 (0) | |
| Ownership start time (0) | | |
| Ownership count and flags (0) | | |

**Figure 13-4:** Device Data Block for Detached Jobs

### 13.1.5 Additional Monitor-Level Data

Since terminal service is the monitor component responsible for job creation, it maintains the various job counts and limit values within its KBDCTL program section.

Location JOBCNT contains the number of currently active jobs on the system.

While JOBMAX (a system generation parameter) is the absolute maximum number of jobs configured on the system, the *actual* number of possible jobs is determined by several other parameters, discussed below.

Location LMTCNT contains the maximum number of simultaneously active jobs permitted during this timesharing session. It is set during system initialization and does not change throughout a single timesharing session.

Location MAXCNT contains the maximum number of jobs that can be swapped at one time. This is determined by the amount of space configured within the system swap files and the maximum storage space a job can occupy in memory and on disk (system parameter SWPMAX). Since every job must be swappable, the actual maximum number of jobs that can be active at once can never be larger than MAXCNT. Note that MAXCNT can change as swap files are added or removed from the system. However, it can never be greater than the value in location LMTCNT.

Location MAXJOB contains the maximum number of active jobs currently permitted at one time. This value can be changed dynamically through use of the "set logins" system directive. However, it can never be greater than the value in location MAXCNT.

The relationship of these values to one another is as follows:

```
0 <= JOBCNT <= MAXJOB <= MAXCNT <= LMTCNT <= JOBMAX
```

## 13.2 Processing Overview

Terminal service is primarily a device driver. As such, it processes all the standard functions required for user I/O to terminal-type devices, both standard terminals and pseudo-keyboards. These functions include opening and closing terminals, servicing user I/O requests, processing terminal interrupts, logging errors, and so on.

In addition, since a job must always be associated with a keyboard device (at least initially), terminal service is also responsible for various job-related functions. These include creating, detaching and terminating jobs as dictated by certain conditions detected on a terminal.

### 13.2.1 Terminal I/O Functions

The terminal service processing of general terminal I/O functions is basically straightforward. The software is loosely organized into separate routines with the appropriate entry points defined in accordance with the standard device driver conventions.

#### 13.2.1.1 Assign and Deassign Processing -

The same procedure is followed to process both the assign and deassign functions (ASN$KB and DEA$KB). All special flags and mode indicators are cleared from the device data block, along with any private delimiters that may be in effect. The input buffer is also cleared.

#### 13.2.1.2 Open Processing -

Open processing (OPN$KB) involves initializing the DDB: storing the open mode, initializing any special flags, and clearing the input and output buffer chains. If multiterminal handling has been enabled on the system and the open request is for a nonzero channel number, terminal service also scans the remaining keyboard units for other terminals that have been assigned to, but not opened by, the calling job. If any are found, they are established as slave terminals and their device data blocks are also initialized.

### 13.2.1.3 Close Processing -

The close function (CLS$KB) is handled in much the same way as the deassign function. Special flags and mode indicators are cleared from the device data block, along with any private delimiters. The input buffer, however, is not cleared unless the terminal was opened in binary input mode. Terminal service also checks to determine if the terminal was opened in echo control mode. If it was, and the terminal is the job's console, the echo control type-ahead buffer is transferred to the keyboard input buffer. If the terminal is not the job's console, the echo control type-ahead buffer is cleared.

### 13.2.1.4 Timeout Processing -

The majority of the processing necessary to handle a terminal timeout (TMO$KB) depends on the particular interface to which the terminal is connected. If the device is the keyboard unit associated with a pseudo-keyboard (that is, its interface code is equal to TTPK11=10), no processing is done at all. Otherwise, a number of processing paths are possible, depending on the specific interface type. In general, however, the software is concerned with only two basic timeout conditions.

First, if the interface is connected to a modem, terminal service tests for dataset hang up and dropped carrier, indicating a disconnected dialup line. If this is the case, and the terminal is either the job's console or has been opened in "guarded" mode (see Section 12.2.10), the terminal is detached from the job. Otherwise, the terminal is cleared -- all input and output buffers are cleared and the flags and status indicators in the DDB are reset.

The second possible condition indicated by a terminal timeout is that the terminal is "hung" or not responding to data. In this circumstance, the device data block is flagged "hang pending" but the terminal is not necessarily disabled. It is possible that there has simply been no user activity for a period of time. Thus, terminal service attempts to "unhang" the device by alternately disabling and reenabling interrupts and then sending a null character to the device. If the device is, in fact, functioning properly, it will generate an interrupt once the null character has been received, and the "hang pending" flag will be cleared.

### 13.2.1.5 User I/O Processing -

For the most part, processing user I/O requests (SER$KB) consists of simply transferring data between the user job's I/O buffer and the terminal input and output buffers.

However, before dispatching to process the read or write request, terminal service determines whether the calling job is actually attached to the terminal. If it is not, and the job is no longer logged in, the job is killed. If the job is detached but still logged in, the software checks to see if the detach is a "guarded" detach (see Section 13.1.4). If so, an error is returned to the caller, thus permitting the program to attempt to recover from the condition and continue executing. If the terminal is not in the "guarded" detach state, the job is made to "hibernate" and will not be eligible to run again until it is reattached to a physical terminal.

Once this initial checking is complete, processing a user read or write request is straightforward. To process a read request, terminal service simply transfers any available input to the user job's I/O buffer.* If the multiple terminal handling feature is enabled on the system and the read request is for input from any nonspecific slave terminal, terminal service performs a round-robin search for a terminal with pending input.

Processing a write request is more complex since a certain amount of data manipulation and translation (RSX mode processing, lower to uppercase translation, tab expansions, and so on) must be done before the user's output data can be stored in the terminal output buffer.

If an I/O request cannot be completed immediately (either because no input is available or there is no room in the terminal output buffer), terminal service determines whether or not it should stall the calling job until conditions permit completion of the I/O. If the request is a read from a specific keyboard in a multiterminal arrangement or if the calling job has specified the "no stall" option, an error is returned to the caller along with a count of the number of unprocessed bytes of data, and the job is made runnable. Otherwise, terminal service stalls the job by exiting through IOREDO. (See Section 6.1.2 for a discussion of the IOREDO mechanism.)

When terminal service is invoked to process a user job's I/O request, the job's I/O buffer is mapped into the monitor's address space. Note, however, that RSTS/E always maps memory on one K-word boundaries but user I/O buffers can begin anywhere. As a result, transferring data into and out of the buffer can occasionally result in a memory management violation. If this happens, terminal service attempts to recover by checking the address at which the violation occurred. If it occurred during the transfer of user data, the error handler remaps the next segment of user code and continues processing. If the violation was caused by some other sequence of instructions, the monitor crashes.

### 13.2.1.6 Interrupt-Level Processing -

Initial processing of terminal device interrupts is based on the type of interface to which the terminal is connected. In particular, if the terminal line is controlled by a modem, the software checks for and processes ring and carrier interrupts from the modem. Character interrupts, however, are processed the same for all terminals, irregardless of interface type.

Character input processing at the interrupt level involves a significant amount of checking and data manipulation. In addition to testing for transmission and data errors, the interrupt-level software checks for and processing the following features and functions:

- XOFF/XON synchronization protocol
- Binary input mode
- ODT submode
- Control characters (such as Control-C, Control-R, carriage return, and so on)
- Escape sequences
- Standard and private delimiters

* The availability of data is normally determined by the presence of a line delimiter in the keyboard input buffer. In binary mode or ODT submode, *any* characters in the buffer constitute available data.

- FMS or echo control input
- Rubout sequences and scope backspaces
- Character translations (such as break to Control-C, lower to uppercase, and alternate altmodes to escape)

Thus, by the time the incoming character is stored and echoed (if necessary), it is already in the form expected by the user job and all flags necessary for later processing are set.

On the other hand, character output processing at the interrupt level is simple. Checking bit TTSXOF in word DDFLAG of the device data block, the software first determines whether it must send the terminal an XOFF. If so, it transmits the character and exits. If no XOFF is required, the software then determines whether output has been suspended due to an incoming XOFF *from* the terminal. If this is the case, the software simply exits. Otherwise, it gets the next character from the output buffer chain, transmits the character, and the exits. No character manipulation is done at interrupt level. Any type of formatting or character translation is done at the user level before the data is stored in the output buffer.

All character interrupt processing is done with a processor priority level equal to the device interrupt priority. As such, this processing is interruptable by "faster" devices but not by other terminal devices.

#### 13.2.1.7 Special I/O Processing -

Special I/O function processing (SPC$KB) operates through a dispatch table, indexed by subfunction code. The majority of subfunctions are handled by a single routine that sets or clears the appropriate status bits in the DDB. The "force input to keyboard" and "broadcast output to terminal" subfunctions are essentially treated as special forms of general terminal I/O. And the "private delimiter" subfunction is processed in a very straightforward fashion.

#### 13.2.1.8 Error Logging and Sleep Checking -

In addition to the required service routines, terminal service includes two optional routines: error logging (ERL$KB) and sleep checking (SLP$KB). ERL$KB formats an error message to be processed by the system logger. SLP$KB checks for available input on the terminal and blocks any conditional sleep for the calling job is there is data pending.

### 13.2.2 Pseudo-Keyboard I/O Functions

For the most part, the pseudo-keyboard device driver is extremely simple. In fact, three of the eight required service routines (ASN$PK, DEA$PK, and TMO$PK) do no processing at all beyond that done by the general monitor support routines. Processing of the remaining functions is very straightforward.

#### 13.2.2.1 Open Processing -

In addition to initializing the pseudo-keyboard device data block, processing of the open function (OPN$PK) involves mapping to and initializing the DDB of the associated keyboard unit. The keyboard's input and output buffer chains are cleared and any defined ring values are loaded into the device data block. However, no job is created. To create the controlled job, the controlling job must output LOGIN information to the pseudo-keyboard or use the "create job" system directive.

#### 13.2.2.2 Close Processing -

Close processing (CLS$PK) involves checking the mode in which the pseudo-keyboard was opened and then either killing or detaching the controlled job according to that open mode. As mentioned in Section 13.2.3.3, killing a job is a 2-step process handled outside the terminal service. The pseudo-keyboard driver merely initiates this process by setting the JFKILL bit in the job's secondary flags word (JDFLG2) and making the job runnable.

#### 13.2.2.3 User I/O Processsing -

Requesting input from a pseudo-keyboard is the same as viewing the controlled job's display, and sending output to a pseudo-keyboard is the same as typing at the controlled job's terminal or forcing input to the controlled job's keyboard. Thus, processing user service requests for pseudo-keyboard I/O (SER$PK) involves simply transferring data to and from the input and output buffers of the associated keyboard unit.

When processing a request for input, the pseudo-keyboard driver never stalls the controlling job to wait for data but immediately returns the contents of the keyboard unit's output buffer. This data can constitute a single record, several records, or a record fragment. If no data is currently available, an "end of file" error is returned to the controlling job.

If the controlled job outputs data faster than the controlling job can read and process it, the keyboard unit's output buffer fills and the controlled job enters an output wait state. When this occurs, the controlling job is immediately made eligible to run so that it can receive and process the data.

Sending output to a pseudo-keyboard is the same as typing data on the controlled job's terminal. However, the data is transferred to the controlled job's input buffer at a much greater speed than if it were typed at a physical keyboard. Thus, if the controlled job cannot keep pace with the input it is receiving, its small buffer chain can fill up. For this reason, the controlling job should send only one output record at a time and retrieve the program or system responses individually. Furthermore, it is desirable for the controlling job to check the status of the controlled job (KB wait state, Control-C state, etc.) before initiating a transfer.

To facilitate this status checking, the pseudo-keyboard driver accepts and processes a modifier argument with output requests. The modifier is passed to the pseudo-keyboard driver at location XRMOD in the calling program's transfer request block (XRB). The low-order six bits of this word are used as individual flags and are processed as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0 | PO.NCK | If this bit is clear, the pseudo-keyboard driver begins processing by checking the status of the controlled job. Certain conditions (such as, no job associated with the keyboard unit or job not in KB wait state) result in the driver returning an error to the calling program. If the bit is set, the driver does no initial status checking. |
| 1 | PO.CKC | If this bit is set (and bit 0 is clear), the driver determines whether the controlled job is in Control-C state, as well as in KB wait state. If it is not, an error is returned to the calling job. |

2 PO.NTX If the two previously described status checks are successful, the driver then checks this bit. If it is set, the requested output function was merely to check the controlled job status. No data is transferred and control returns to the calling job. Otherwise, the driver proceeds to transfer the caller's output data to the controlled job's input buffer.

3 PO.STL If this bit is set, the pseudo-keyboard driver will stall the calling program if the data transfer cannot proceed due to a lack of small buffers.

4 PO.KIL If this bit is set, the calling program has requested that the controlled job be killed, either unconditionally or in conjunction with the initial status checking described above (bits 1 and/or 0). As mentioned before, the pseudo-keyboard driver does not actually kill the job. It merely initiates the process by setting the JFKILL bit in the job's primary flags word (JDFLG) and making the job runnable.

5 PO.FCR If this bit is set, the pseudo-keyboard driver resets the keyboard unit's horizontal position (DDHORZ) following the data transfer.

#### 13.2.2.4 Interrupt-Level Processing -

Since a pseudo-keyboard is not a physical interrupt-driven device, the pseudo-keyboard driver does no interrupt-level processing.

#### 13.2.2.5 Special I/O Processing -

Pseudo-keyboard processing of the special I/O function (SPC$PK) involves only the NOECHO bit in word DDFLAG of the associated keyboard unit's device data block. The bit is clear by default, thereby enabling echo (to the controlled job's output buffer) of data output to the pseudo-keyboard. Using the SPEC function, the controlling job can set the bit (disabling echo), clear the bit (reenabling echo) or simply read the current status of the bit.

#### 13.2.2.6 Sleep Checking -

Sleep checking (SLP$PK) is an optional service routine, entered whenever a job using a particular device requests a conditional sleep. The device driver can then determine whether or not the job should be permitted to suspend execution. The pseudo-keyboard driver includes this service routine to process conditional sleep requests from the controlling job. If the controlled job is stalled waiting for input or has sent output that has not yet been received and processed, the controlling job is not permitted to suspend execution. Otherwise, the conditional sleep succeeds.

### 13.2.3 Job-Related Functions

Terminal service is responsible for certain job related functions in addition to its standard driver functions. These include creating, detaching, and terminating (or killing) jobs according to certain terminal conditions. These functions are also accessible to user jobs through various system directives. When any such directive is invoked, the monitor EMT trap handler dispatches to the appropriate software within terminal service.

### 13.2.3.1 Creating a Job -

Terminal service creates a new job under two circumstances:

- When a delimiter is detected on an unassigned terminal, or
- When a job issues the "create job" system directive for another job.

Job creation involves assembling a skeletal job control structure. To do this, terminal service requires an empty slot in the master job table (JOBTBL) and three small buffers from the general pool. (See Chapter 5 for a complete description of the job control structure.) Once the necessary resources are acquired, terminal service initializes conditions for the job (job size, run burst, job status, and so on). It also initializes the terminal's device data block, making it the job's console. The job is then made runnable as a logged-out job.

Note that none of this processing requires any knowledge of user accounting information or any interpretation of the typed input. When the job is made runnable, the scheduler passes control to the 'new user' entry point in the primary run-time system (see Section 15.5). The software at this entry point then invokes the LOGIN procedure. If the login sequence is valid, control passes to the system default keyboard monitor, which waits for further input from the terminal. If the login sequence is invalid, the job remains logged out and is eventually terminated. (See Section 13.2.3.3.)

### 13.2.3.2 Detaching a Job -

Terminal service detaches a job under three circumstances:

- When a "detach job" system directive is issued for the job,
- When the dialup line over which the job is running becomes disconnected, or
- When the job is running on a keyboard unit associated with a pseudo-keyboard opened with a mode of 1 and the pseudo-keyboard is closed.

To detach a job from a terminal, terminal service merely clears the job number in the terminal device data block. Once the job is detached in this fashion, it can continue to execute but cannot do I/O on that particular device. If the job attempts to do I/O on the terminal, it enters a state known as "hibernation" and must then wait until it is reattached (through some external process) before it can become runnable.

However, if the terminal was opened in "guarded" mode, terminal service not only clears the job number in the DDB but also loads the address of the dummy DDB into the appropriate entry of the job's I/O block (IOB). This serves as a flag to prevent hibernation should the job issue further I/O requests for the terminal (see Section 13.1.4).

### 13.2.3.3 Terminating a Job -

Terminal service terminates a job under two circumstances:

- When the job is logged out and attempts to enter Control-C state, or
- When the job is running on a keyboard unit associated with a pseudo-keyboard opened with a mode of 0 and the pseudo-keyboard is closed.

Terminating a job is a 2-stage function. First the job must be logged out. Once it is logged out, the job can be deleted and its data structures can be returned to the general small buffer pool. Terminal service initiates this process by setting bit JFKILL in the job's secondary job status word (JDFLG2) and making the job runnable.

# CHAPTER 14
# The Human Interface [V8.0]

The RSTS/E operating system is designed to run jobs under the control of a run-time system and each job must have a run-time system associated with it. In general, a run-time system acts as an interface between the RSTS/E monitor and the system user — either a human user and/or user software. To act as an interface for the human user, however, a run-time system must support what is known as a "keyboard monitor."

This chapter discusses the general concepts of the RSTS/E human interface. Section 14.1 discusses keyboard monitors in general, together with the command environment that they provide the system user. Section 14.2 describes the Concise Command Language (CCL) facility, a special extension of the command environment supported by all standard RSTS/E keyboard monitors.

NOTE

> The general structure of run-time systems used to interface between the RSTS/E monitor and user software is discussed in Chapter 15.

## 14.1 Keyboard Monitors

The portion of a run-time system with which the user communicates is known as a "keyboard monitor." This is the part of the run-time system that analyzes (or "parses") user-typed commands and invokes either the appropriate component of the RSTS/E monitor or system utility to execute the user's request.

### 14.1.1 Default Keyboard Monitors

As discussed in Chapter 15, all RSTS/E systems have at least one permanently installed run-time system known as the "primary" or "default" run-time system. In fact, at system startup, the primary run-time system is the *only* run-time system available. Optionally, one or more auxiliary run-time systems can be added during the system startup process or at some later time during the timesharing session.

The system's primary run-time system also acts (at least initially) as the system's default keyboard monitor, controlling jobs that are not currently running programs. When a new job is created, the terminal service job creation code sets the new job's current and default run-time system and keyboard monitor to the system default. The user can, however, select a different run-time system to act as its *private* default run-time system and keyboard monitor.

When a user job runs a program, the monitor switches control of the job to the run-time system under which the program must run. When the program exits, control is switched back to the job's private default run-time system. (However, if the job's default run-time system is no longer installed when the program exits, control is switched to the system default.) Thus, a job can use one run-time system as its keyboard monitor while running programs under the control of other run-time systems.

At system startup, the system primary run-time system is designated as the system default keyboard monitor. Using the "declare default keyboard monitor" system directive, however, the system manager (or any privileged user) can declare a default keyboard monitor that is different from the keyboard monitor of the primary run-time system. Thus, for example, a system can have BASIC-PLUS or RSX as its primary run-time system but use DCL as its default keyboard monitor.

If a separate default keyboard monitor is defined, terminal service will use that keyboard monitor as a new job's default run-time system. Control, however, passes initially to the primary run-time system to handle the LOGIN sequence. When the job exits from LOGIN, control then passes to the default keyboard monitor.

After a separate default keyboard monitor is defined, the monitor returns control to that keyboard monitor whenever an action occurs that would otherwise return control to the primary run-time system (for example, when a user runs the SWITCH program without specifying a run-time system name, or when certain nonrecoverable errors occur in the job's private default keyboard monitor).

A separate default keyboard monitor remains in effect until one of the following events occurs:

- Another "declare default keyboard monitor" system directive is issued.
- The run-time system containing the default keyboard monitor is removed from the system.
- A nonrecoverable error (such as a fatal disk error during a swap, a SP stack overflow, or a memory parity fault) occurs in the run-time system containing the default keyboard monitor.
- The system is shut down.

### 14.1.2 Command Environments

The major function of a keyboard monitor is to provide a command environment in which the system user can access various RSTS/E monitor features. Thus, each keyboard monitor has its own set of monitor level commands, consistent with the general structure and operation of its associated run-time system. Certain commands, however, are common to all the standard RSTS/E keyboard monitors. These common commands are as follows:

ASSIGN
BYE
DEASSIGN
EXIT
HELLO
MOUNT
REASSIGN
RUN

In addition, each of the standard keyboard monitors supports the Concise Command Language (CCL) facility and recognizes all CCL commands defined for the system (see Section 14.2).

Four of the standard RSTS/E run-time systems include a keyboard monitor. Each has an identifying prompt that it displays to indicate that the terminal is currently at system command level. These prompts are as follows:

| | |
|---|---|
| $ | DIGITAL Command Language (DCL) |
| Ready | BASIC-PLUS |
| ] | RSX |
| | RT11 |

## 14.2 Concise Command Language

RSTS/E allows a user to run certain programs (typically the system library programs) by typing a brief system command called a Concise Command Language (CCL) command. A CCL command is a shorthand alternative to typing a longer "RUN" specification (for example, "PIP" instead of "RUN [1,2]PIP") and therefore permit the user to access frequently used programs in much the same manner as the available set of keyboard monitor commands. It is also possible to pass a line of input to the program when typing the CCL command.

CCL commands can be used to run user programs, in addition to those in the system library, as long as these programs are coded to recognize and process the fact that they have been invoked with a CCL command. CCL commands can also be invoked from within a user program using the "execute CCL command" system directive.

CCL commands do not have permanent definitions and must be installed either at system startup or during timesharing. They can be defined through the "add/delete CCL command" system directive or with the UTILTY system utility program.

### 14.2.1 Data Structures

When a Concise Command Language command is installed, its definition is stored in the monitor in a 16(10)-word block allocated from the FIP small buffer pool. When the command is later invoked, portions of the command block are copied into the low-core area of the user job's address space for use by the called program.

### 14.2.1.1 CCL Command Block -

The command blocks for all CCL commands installed on the system are kept in a linked list within the monitor. Location CCLLST is the root of this list and points to the first CCL command block in the list. The list is ordered according to the order in which the commands were installed, with new command blocks always being linked to the end of the list.

Figure 14-1 shows the format of a CCL command block. Following the figure is a description of each entry. Note that the offsets within the block correspond to offsets within the file request queue block (FIRQB) since that structure is used to specify the command at the time the command is installed.

| | |
|---|---|
| Link to next command block | FIRQB |
| Pointer to the start of the command text | FQJOB |
| Pointer to first optional character | FQFIL |
| Project-programmer number | FQPPN |
| Program file name (in RAD50) | FQNAM1 |
| Program file type (in RAD50) | FQEXT |
| Command text string | FQSIZ |
| Program device name | FQDEV |
| Program device unit number | FQDEVN |
| Pointer to first optional character | FQCLUS |
| Entry parameter | FQNENT |

**Figure 14-1:** CCL Command Control Block

*Offset* *Contents*

0 This word contains the address of the next CCL command block in the linked list. If this is the last block in the list, this word contains a zero.

FQJOB This word contains the address of the start of the CCL command text string. This always points to the word at offset FQSIZ.

FQFIL This word contains the address of the first optional character in the command text string. CCL commands can be abbreviated to some leading portion of their full length, and this word specifies the portion of the command that is required. For example, on some systems, the command "SYSTAT" might be abbreviated simply "S", whereas other systems might require "SY" to uniquely identify the command. In the first instance, the address in this word would point to the Y in the command string, and in the second instance, it would point to the second S.

FQPPN This word contains the PPN of the program to run in response to this command. The high byte contains the project (or group) number and the low byte contains the programmer (or user) number.

FQNAM1 These two words contain the filename (in RAD50 format) of the program to run in response to this command.

FQEXT This word contains the file type (in RAD50 format) of the program to run in response to this command. If the value here is -1, the automatic file type determination rules of the "RUN" processor will be used to locate the required program. (See Section 15.1.1 for a discussion of RUN processing.)

FQSIZ This 10(10)-word area contains the actual CCL command. The command is stored as a string of up to nine characters, terminated by a byte containing -1. If the command is only one character long, it must be either a letter or one of the special character: "@", "#", or "$". (See Section 14.2.2.) If the command is two or more characters, it must begin with a letter and can contain only letters and digits.

FQDEV This word contains the name of the device (in ASCII) on which the program is located. It must be a disk device. A value of zero specifies the public disk structure.

FQDEVN This word contains the unit number of the device on which the program is located.

NOTE

> The specifications at offsets FQDEV and FQDEVN follow the standard FIRQB form for device name and unit. All combinations that are legal in a FIRQB are legal here. The only exception is that the device must be a disk device.

FQCLUS This word contains a copy of the pointer stored in FQFIL.

FQNENT This word contains the initial parameter to be passed to the run-time system when the program is run. The monitor takes no action as a result of the contents of this word; any processing is left to the run-time system. (For example, BASIC-PLUS interprets this word as the starting line number of the program.)

#### 14.2.1.2 Core Common String (CORCMN) -

When a run-time system is entered in response to a Concise Command Language command, the CCL command and its arguments are stored in the user's "core common" string, a 128(10)-byte area beginning at location CORCMN in the user job's low core area. The first byte of the string contains the number of characters that follow. The data passed consists of the full CCL command string (even if the user åbbreviated the command), followed by any arguments supplied by the user.

### 14.2.2 System Processing

When a line of input is typed at a terminal waiting at system command level or is included with the "execute CCL command" system directive, the line is passed to the Concise Command Language parser in the monitor. The parser then processes the line to determine if it contains a valid CCL command.

To process the line, the parser first "translates" the input string. The following steps are involved in this translation:

- Any parity bits are stripped from the input characters.
- All control characters, line terminators, excess null and rubout characters, and leading and trailing spaces and tabs are discarded.
- All tabs are changed to spaces and all multiple spaces are reduced to single spaces.
- Lowercase characters are changed to uppercase.

- Any characters within quotes are left unaltered.

Once the input string is translated in this fashion, the parser examines the string's format to determine whether it could be a valid CCL command. The actual format of a CCL keyboard command is described in detail in the *RSTS/E Programming Manual*. Briefly, the format consists of a CCL command keyword, followed by one or more optional CCL switches (see below), followed by any additional input to be passed to the invoked program.

The command keyword must be from one to nine characters (as described in Section 14.2.1.1) and is ordinarily delimited from any additional input by one or more blank characters (spaces or tabs). The parser, however, recognizes and processes three special characters ("@", "#", and "$") as "self-delimiting" CCL commands. If any of these characters is defined as a CCL keyword, the parser treats an input string beginning with that character as if the character was followed by a space. Thus, for example, if the single character "$" is defined as a CCL command to run DCL, an input string of the form "$xxxx" will invoke DCL (from any keyboard monitor). The "xxxx" portion of the input string is then passed as additional input to DCL, which attempts to interpret that input as a standard DCL command.

When a possible CCL command keyword has been extracted, the parser compares it with each entry in the list of valid CCL commands (CCLLST). If the keyword matches a defined CCL command at or beyond its abbreviation point, the parser then writes the fully expanded CCL command (as specified in the CCL command block) to the job's core common area. If no match is found, the parser writes the translated command line to the core common area and returns control to the job's keyboard monitor.

When a CCL command match is found, the parser then checks the remainder of the translated command line for valid CCL switches. [Two switches are available: /SIZE (to specify the size required for the program) and /DETACH (to specify that the program is to run detached).] If any are found, they are stripped from the command line. The parser sets the corresponding status bits in the transfer request block (XRB) but takes no direct action on the switches. The run-time system under which the invoked program runs can optionally interpret and process these status bits. The remainder of the input string is then written to the core common area following the expanded CCL command.

Once the command line has been completely processed and the job's core common area is set up, the parser extracts the entry parameter from the CCL command block and loads it into the XRB. The program file is opened on channel 15 in the job's I/O block, and the program is set up to run. Control then passes to the run-time system associated with the invoked program, at the P.RUN entry point (see Section 15.5). When the program exits, control returns to the job's default keyboard monitor.

### 14.2.3 Rules of Precedence

With the exception of DCL, all standard RSTS/E keyboard monitors follow the same rules of precedence for Concise Command Language commands — CCL commands are processed before keyboard monitor commands. On the other hand, DCL processes keyboard monitor commands before CCL commands, unless the CCL prefix was specified. (See the *RSTS/E DCL User's Guide* for details on the CCL prefix.)

In addition, due to its unique set of permissible constructs at command level, BASIC-PLUS uses a somewhat expanded procedure for analyzing command level input. This procedure is as follows:

- If the line begins with a numeric character, it is passed to the BASIC-PLUS parser for processing and storage as intermediate code.
- If the line begins with a nonnumeric character, it is passed to the CCL command parser for validation.
- If the line does not contain a valid CCL command, it is passed to the BASIC-PLUS parser for immediate mode processing.
- If the line does not contain a valid BASIC-PLUS keyboard monitor command or immediate mode statement, BASIC-PLUS generates an error message.

Thus, BASIC-PLUS processes CCL commands before keyboard monitor commands and immediate mode statements, but after line-numbered statements.

Note, therefore, that caution must be used when defining Concise Command Language commands. Except when using DCL, a CCL command that duplicates a keyboard monitor command or a BASIC-PLUS immediate mode statement will override that command or statement, thus making certain keyboard monitor features unavailable.

# Part V
# SHARED COMMON CODE

In any operating system, it is advantageous to permit user jobs to share the use of data and program code that would otherwise be contained within each individual user program. Permitting code to be shareable reduces swapping and physical memory requirements and generally improves overall system performance.

Under RSTS/E there are two ways in which user jobs can share data and program code: run-time systems and resident libraries. However, while run-time systems and resident libraries are both shareable and are similar in many ways, they serve separate purposes and have distinct differences.

- In general, run-time systems are used to extend the functionality of the monitor. Each job must run under the control of a run-time system, which acts as an interface between the RSTS/E monitor and the job.
- Resident libraries, on the other hand, are typically used to extend the functionality of a program. They run under control of the user job, from which they are accessed only as needed.

This part of the *RSTS/E V8.0 Internals Manual presents a general overview of the features and operation of both run-time systems (Chapter 15) and resident libraries (Chapter 16). Note that the intent here is not to provide the details necessary to write either of these entities. Specific guidelines for doing so can be found in the RSTS/E System Directives Manual* and the *RSTS/E Task Builder Reference Manual.*

# CHAPTER 15
# Run-Time Systems [V8.0]

The RSTS/E operating system is designed to run jobs under the control of a run-time system and each job must have a run-time system associated with it.

This chapter presents a general overview of run-time systems. Section 15.1 discusses some general concepts. Section 15.2 presents the details of the monitor control structures involved in run-time system processing, while Section 15.3 covers those structures within the user space. Section 15.4 discusses the general processing of synchronous and asynchronous traps, and Section 15.5 describes the required entry points into a run-time system.

## 15.1 General Concepts

In general, a run-time system acts as an interface between the RSTS/E monitor and the system user — either a human user and/or user software.

NOTE

> This chapter is concerned with the concept of a run-time system as an interface for user software. To act as an interface for the human user, a run-time system must support what is known as a "keyboard monitor." Keyboard monitors are discussed in Chapter 14.

As an interface for user software, a run-time system loads programs from disk and initializes them for execution. It also returns control to the job's keyboard monitor once a user program has terminated execution. Other common run-time system functions include centralized error handling and nonstandard trap processing. In addition, a run-time system can be written to emulate the directives of other operating systems.

Some run-time systems act as a language interface for programs written in higher level languages — languages such as BASIC-PLUS that have no trap facility and, therefore, no direct way to call the monitor.

An important feature of run-time systems is that they generally represent a segment of common code that can be shared by many users. Normally they are implemented as "pure" code — that is, instructions and fixed data only. Thus, in addition to being shareable, they do not need to be "swapped" to and from disk as user programs are. Since they contain no variable data, run-time systems can be simply overwritten when they are not being used and reloaded when they are needed again.

In general, a program written for one run-time system cannot be executed under control of another. (This is particularly true if the program uses those features of the run-time system that emulate another operating system.) It is possible, however, for run-time systems to be compatible, where code written under one will execute properly under the other. For example, there are several run-time systems (such as BASIC2 and BP2COM) based on the RSX run-time system. Code assembled under the RSX run-time system can be run under these RSX derivatives.

### 15.1.1 Primary and Auxiliary Run-Time Systems

All RSTS/E systems have at least one permanently resident run-time system known as the "primary" run-time system. It is installed during the system generation process and cannot be removed once it is installed. It is also the first (and only) run-time system available at system startup. Optionally, an installation can have one or more "auxiliary" run-time systems, added during the system startup process or at some time after.

The system's primary run-time system is also known as the system default run-time system. In addition, every job has its own private default run-time system which acts as the job's keyboard monitor when it is not running any user program.* When a program is executed, the monitor switches control of the job to the run-time system under which the program must run. When the program exits, control is switched back to the job's private default. This process permits a job to use one run-time system as its keyboard monitor while running programs under the control of other run-time systems.

The job creation code initially sets both the new job's current and default run-time systems to the system default, but the default can be changed by the user through a monitor-level keyboard command. However, if a job's default run-time system has been removed when a user program exits, control of the job is switched to the system default.

Every run-time system installed on the system is described by a "run-time system descriptor block" (see Section 15.2.1), and all such descriptor blocks are linked together in a singly linked list. The first entry in the list is always the system default run-time system. When an auxiliary run-time system is installed, the system manager can optionally position the descriptor block within the system list. If the run-time system is to be used frequently, placing it toward the front of the list will reduce system overhead. If no position is specified, the descriptor block is added to the end of the list.

The position of a run-time system descriptor block in the system list also affects how the monitor treats a RUN request for a program when no file type is specified. On receiving such a RUN request, the monitor checks the indicated directory for all files with the specified name and an executable file type. If it finds more than one executable file of the proper name, the program that is run is the one controlled by the run-time system nearest the front of the list.

### 15.1.2 Run-Time System Emulators

As mentioned, run-time systems can be written to emulate the environment of other operating systems. In particular, RSTS/E supports two such run-time systems — one that emulates the RSX real-time system and one that emulates the RT-11 "single-job" monitor. These run-time systems are useful for developing programs to be run under both RSTS/E and the operating system being emulated.

* Note, therefore, that both the system default run-time system and a job's private default run-time system must support a keyboard monitor.

It should be noted, however, that when a run-time system is described as emulating a particular operating system environment, this statement is only partially true. The RSX run-time system, for example, does not provide an actual real-time multiprocessing environment to users of a RSTS/E timesharing system. While it does process directives that are identical in form and similar in objective to a subset of the RSX-11M executive directives, not all the RSX-11M directives are provided and the meanings of those that are emulated are fitted to the RSTS/E environment.

### 15.1.3 User-Written Run-Time Systems

Several standard run-time systems are provided under RSTS/E, each with its own set of features and characteristics. It is possible, however, that a user application can require features different from those supported by these standard run-time systems. In that event, a user-written run-time system can be implemented. Indeed, before RSTS/E V7.0, the only way to provide shareable code was with a run-time system.

There are two main drawbacks to implementing a run-time system. First, run-time systems must adhere to certain strict (and sometimes limiting) guidelines. Second, since they do not permit the use of either disk- or memory-resident overlays, a user job can find itself faced with critically restricted space requirements. Resident libraries, on the other hand, are easier to implement, and they can be written using memory-resident overlays. (Disk overlays are still not permitted.)

There are several factors to consider when deciding whether to implement shareable code through a run-time system or a resident library. In general, a run-time system should be used under the following circumstances:

- When the shareable code is to control the execution of the program, rather than the other way around.
- When the shareable code emulates another operating system or environment.
- When the user program uses any of the interrupt-generating processor instructions (EMT, TRAP, IOT, or BPT) in a nonstandard fashion.
- When the shareable code must serve as a keyboard monitor.

On the other hand, implement the shareable code through a resident library under the following circumstances:

- When the user program needs the features provided by an existing run-time system.*
- When the shareable code consists of support code (that is, subroutines and/or data) that will run under control of the program. (While a run-time system can also be used in this case, it will generally be more difficult to implement.)
- When the virtual address space used by the program must be optimized by using memory-resident overlays.

* Note, however, that programs using resident libraries must run under the RSX run-time system or one of its derivatives. RSTS/E provides no resident library support for programs written in BASIC-PLUS and there are currently no plans to do so.

## 15.2 Monitor Control Structures

Every run-time system installed on the system is described by a run-time system descriptor block (RTS). This block is allocated from the monitor's general small buffer pool when the run-time system is added (or "installed") and is returned to the buffer pool when the run-time system is removed.

The run-time system descriptor blocks of all installed run-time systems are linked together in a singly linked list, pointed to by monitor location RTSLST. The first entry in the list is always the primary (system default) run-time system.

The control structures for all active jobs contain pointers to the descriptor blocks of each job's current and default run-time systems. This allows quick access to the run-time systems used by any given job. In addition, the descriptor blocks of all installed run-time systems are linked together, permitting the monitor to access any given run-time system by name.

### 15.2.1 Run-Time System Descriptor Block (RTS)

Every installed run-time system has an associated run-time system descriptor block (RTS). At system startup, only the primary run-time system is installed so only one RTS exists. The auxiliary run-time systems must be installed and the descriptor blocks created during each timesharing session.

The job data blocks for each active job on the system contain pointers to the RTS of both the job's current and private default run-time systems. The address of the RTS of the current run-time system is found at offset JDRTS in the job data block (JDB); the address of the RTS of the job's default run-time system is found at offset J2DRTS in the secondary job data block (JDB2).

Each run-time system has certain predefined characteristics indicated by parameters and bit flags specified in its pseudo-vector region (see Section 15.3). When the run-time system is installed and its descriptor block is built, those parameters are copied into the RTS. However, using the optional switches provided with the installation process, the system manager can override some or all of these characteristics. (Note that changing the characteristics of a run-time system does not alter the characteristics specified in the disk file; only the definitions in the descriptor block are affected.)

Figure 15-1 shows the format of the run-time system descriptor block. Each item is discussed below.

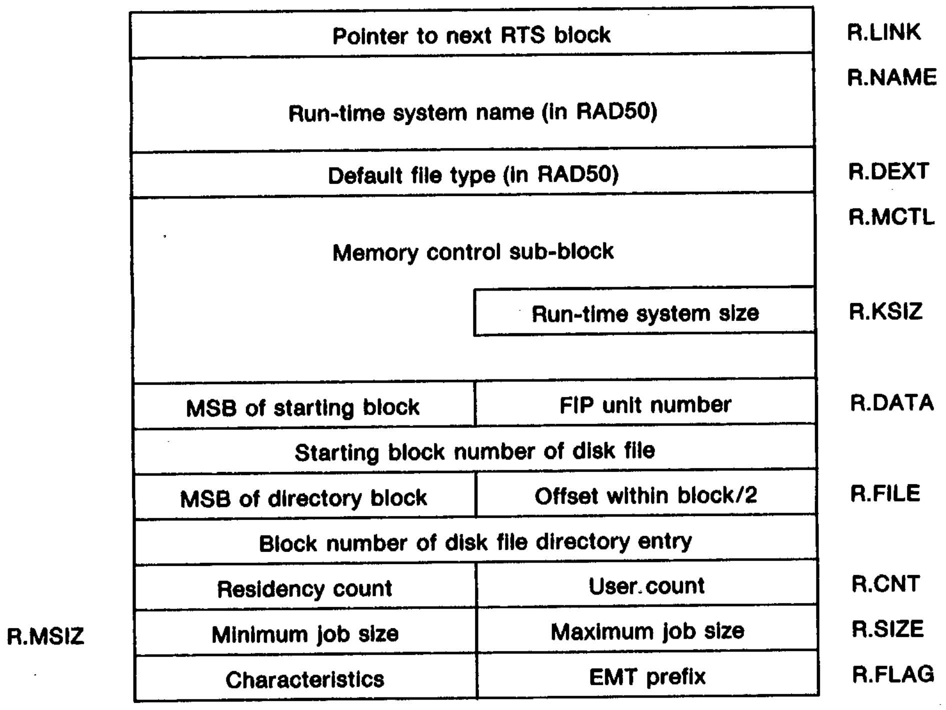


**Figure 15-1:** Run-Time System Descriptor Block

| *Offset* | *Content* |
|---|---|
| R.LINK | This word contains the address of the next run-time system descriptor block in the system list of run-time systems (RTSLST). If this entry is the last in the list, this word contains a value of zero. |
| R.NAME | These two words contain the name of the run-time system, in RAD50 format. The run-time system is stored on disk in account [0,1] using this file name, with a file type of "RTS". |
| R.DEXT | This word contains the default file type of an executable file for this run-time system, stored in RAD50 format. The RUN command processor uses this value to find the correct file to run when no file type was specified. If this word contains a zero, the run-time system has no default file type. |
| R.MCTL | This 5-word area is the memory control sub-block for the run-time system. (See Chapter 4 for a discussion of memory control and the memory control sub-block.) |
| R.KSIZ | This byte (within the memory control sub-block) contains the size of the run-time system, in K-words. |
| R.DATA | This byte contains the FIP unit number of the disk on which the run-time system file is stored. |

R.DATA+1 This 3-byte area contains the 24-bit FIP block number (FBN) of the first block of the run-time system's image on disk. The byte at R.DATA+1 is the most significant byte of this value.

NOTE

All run-time systems are stored on disk in contiguous files. Thus, given the starting block number and the size of the file (derived from R.KSIZ), the entire run-time system can generally be loaded with one disk access. If the file is longer than 31 K-words, however, the memory manager will break the file into 31 K-word segments for loading.

R.FILE This byte contains the offset, divided by two, into the directory block to the name entry of the run-time system's disk file. This value is used to close the file when the run-time system is removed from the system.

R.FILE+1 This 3-byte area contains the 24-bit FIP block number (FBN) of the directory block containing the name entry of the run-time system's disk file. The byte at R.FILE+1 is the most significant byte of this value.

R.CNT This word contains the access counter for the run-time system. The low byte is incremented every time a job begins using the run-time system and is decremented when a job stops using it. The high byte is incremented every time a job controlled by this run-time system becomes resident and is decremented every time an associated job is swapped out. If the entire word goes to zero, the run-time system is not being used and can be removed (unloaded) from memory.

If the run-time system is to remain in memory (even when there are no jobs using it), the sign bit (bit 15) is set. This ensures that the count will never be zero. (Note that the system default run-time system always has bit 15 of this word set.)

R.SIZE This byte contains the maximum job image size, in K-words, of a job using this run-time system.

R.MSIZ This byte contains the minimum job image size, in K-words, of a job using this run-time system. This value is always nonzero.

R.FLAG This word contains a set of eight flags describing the general characteristics of the run-time system, along with an 8-bit value used for the special EMT "prefix" feature. The format of this word, along with the setting of each of the flag bits, is identical to the characteristics word (P.FLAG) of the run-time system's pseudo-vector region (see Section 15.3).

### 15.2.2 Disappearing RSX Descriptor Block (NULRTS)

When a RSTS/E system is generated, the system manager has the option of installing the emulation code of the RSX run-time system as part of the monitor. When this is done, the portion of the run-time system code that processes monitor calls from the user software is permanently resident in memory. It is not permanently mapped, however, but is dynamically mapped (as needed) using kernel APR 5.

The entire RSX run-time system exists as a file that is loaded from disk when necessary, remains as long as required, and is removed when it is no longer needed. When an RSX program is to be run, for example, the RSX run-time system is required to load the program from disk and initialize it for execution. Once the program is loaded, however, the entire run-time system is no longer needed. Since the portion necessary for processing traps and RSX directives is permanently resident, control passes to the monitor and the run-time system "disappears" from the user job's high segment. At this point, the program can execute a directive to expand itself up to a full 32 K-words, making use of the space no longer used by the run-time system.

Under RSTS/E, every job must have a run-time system associated with it at all times. Therefore, every job data block must contain a (nonzero) pointer to a run-time system descriptor block. To satisfy this requirement for those jobs using the disappearing RSX run-time system, the monitor contains a null descriptor block, NULRTS. This null descriptor block is not linked into the system list of installed run-time systems (RTSLST), but is only used to provide a nonzero pointer in the job data block.

The format of the null run-time system descriptor block is the same as that of a regular RTS. All fields contain zeros except the following:

| *Offset* | *Content* |
|---|---|
| R.NAME | "...RSX" (in RAD50 format) |
| M.CTRL | LCK bit set (see Chapter 4 for details of memory control) |
| R.SIZE | System swap maximum |
| R.MSIZ | 1 |

## 15.3 Pseudo-Vector Region

Within RSTS/E, a run-time system acts as the interface between the user and the system. Thus, a run-time system communicates with both the user job and the monitor. Control of software execution passes back and forth between the three entities, and data is passed using established ranges of virtual addresses.

As mentioned in Chapter 2, user jobs can occupy up to 32 K-words of virtual address space, using the eight user APRs. The program being executed occupies the job's "low segment" (mapped with the low-order user APRs), while the run-time system is contained in the "high segment" (mapped with the high-order APRs). Thus, the high segment occupies a multiple of four K-words of the high portion of the job's virtual address space.

The monitor uses certain areas within the high and low segments to access information and return data to the job. The first 1000(8) locations of the low segment are used to exchange data with the user program. (The format of this area is discussed in detail in Section 5.1.2.)

The very top of the high segment contains what is known as the "pseudo-vector" region and is used to provide the monitor with information concerning the characteristics and operation of the run-time system. In general, this region contains the following:

- Flags and parameters defining the capabilities of the run-time system
- Addresses of locations within the run-time system to which the monitor should pass control when certain conditions occur. These addresses are of three types.

- Synchronous system trap (SST) service routines. SSTs cause control to be passed to the monitor because of something internal to the job itself. (Synchronous trap processing is discussed in Section 15.4.1.)
- Asynchronous system trap (AST) service routines. ASTs cause control to be passed to the monitor because of something external to the job. (Asynchronous trap processing is discussed in Section 15.4.2.)
- Standard run-time system entry points. (Run-time system entry points are discussed in Section 15.5.)

Figure 15-2 shows the format of the pseudo-vector region. Each item is discussed below.

*Address* *Symbol* *Content*

P.OFF — This mnemonic is simply used to define the first word of the pseudo-vector region. It is equivalent to 177732(8), the same as the mnemonic P.FLAG.

177732 P.FLAG — This word contains a set of eight bit flags that define the characteristics of the run-time system, along with a 8-bit value used if the run-time system is using the special EMT "prefix" feature (see Section 15.4.1.1). When the run-time system is installed, these bit flags are copied into the run-time system descriptor block at offset R.FLAG. However, as mentioned in Section 15.2.1, the UTILTY program provides optional switches that can be used to override these flags during the installation process.

The format of this flag word and the setting of each of the flag bits is as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-7 | | This byte contains the code to be used with the special EMT "prefix" feature (see Section 15.4.1.1). It has no meaning unless bit 15 (PF.EMT) is set. |
| 8 | PF.KBM | The run-time system can serve as a keyboard monitor. |
| 9 | PF.1US | The run-time system is nonshareable and can only control one user. |
| 10 | PF.RW | The run-time system should be mapped read/write instead of read-only. |
| 11 | PF.NER | Errors occurring within the run-time system should not be logged to the system error log. |
| 12 | PF.REM | The run-time system should be removed from memory immediately when the access count at offset R.CNT in the run-time descriptor block goes to zero. |

13 PF.CSZ The initial job image size (in K-words) of a program running under the run-time system should be computed from the size of the program's file on disk. If this bit is clear, the monitor preallocates space for the job image based on the value found at location P.MSIZ.

14 PF.SLA The run-time system should be loaded at the address specified in the memory control subblock within the run-time descriptor block.

15 PF.EMT The run-time system is using the special EMT "prefix" feature. The associated prefix code is found in bits 0 to 7. (See Section 15.4.1.1.)

Note that bits PF.RW, PF.1US, and PF.NER are typically set only while the run-time system is being tested. The usual procedure is to specify the bits as nonzero here in the pseudo-vector region and override them when the run-time system is installed for testing.

177734 P.DEXT This word contains the run-time system's default file type for executable files — three characters in RAD50 format. If the run-time system has no standard default executable file type, this word will contain a zero.

177736 P.ISIZ This word is no longer used. It must contain a zero.

| | | |
|---|---|---|
| 177732 | Characteristics flags | P.FLAG |
| 177734 | Default executable file type | P.DEXT |
| 177736 | (former use now obsolete — reserved) | P.ISIZ |
| 177740 | Minimum size of user job image | P.MSIZ |
| 177742 | Trap address for FIS hardware floating point | P.FIS |
| 177744 | Crash entry point (primary run-time system) | P.CRAS |
| 177746 | Startup entry point (primary run-time system) | P.STRT |
| 177750 | Entry point for new user | P.NEW |
| 177752 | Entry point for new program | P.RUN |
| 177754 | Trap address for "bad" errors | P.BAD |
| 177756 | Trap address for BPT and T-bit traps | P.BPT |
| 177760 | Trap address for IOT instructions | P.IOT |
| 177762 | Trap address for non-monitor EMT instructions | P.EMT |
| 177764 | Trap address for TRAP instructions | P.TRAP |
| 177766 | Trap address for FPP floating point unit | P.FPP |
| 177770 | Trap address for one Control-C | P.CC |
| 177772 | Trap address for two Control-Cs | P.2CC |
| 177774 | Maximum size of user job image | P.SIZE |
| 177776 | (reserved for future use) | |

P.OFF = P.FLAG

**Figure 15-2:** Run-Time System Pseudo-Vector Region

177740 P.MSIZ This word contains the minimum job image size, in K-words, for a user of this run-time system. It is used to check the validity of a user's request to change its size. The value must be an integer between 1 and 28(10).

177742 P.FIS This word contains the trap processing address for the hardware floating-point instruction set available on certain PDP-11 processor models. (See Section 15.4.1.)

177744 P.CRAS This word contains the address of the entry point used by the primary run-time system to restart the system after a crash. (See Section 15.5.)

177746 P.STRT This word contains the address of the entry point used by the primary run-time system for normal system startup. (See Section 15.5.)

177750 P.NEW This word contains the address of the run-time system's "new user" entry point. It is typically used to process a switch back to the job's private default run-time system. (See Section 15.5.)

177752 P.RUN This word contains the address of the run-time system's entry point used to process the loading and execution of a new user program. (See Section 15.5.)

177754 P.BAD This word contains the trap processing address for various "bad" errors. "Bad" errors are generally those hardware and software errors that are considered nonrecoverable. (See Sections 15.4.1 and 15.4.2.)

177756 P.BPT This word contains the trap processing address for breakpoint trap (the BPT instruction) and for T-bit traps. (See Section 15.4.1.)

177760 P.IOT This word contains the trap processing address for the IOT instruction. (See Section 15.4.1.)

177762 P.EMT This word contains the trap processing address for all non-monitor EMT calls. If the run-time system is using the special EMT "prefix" feature, all EMT calls are processed at this address, except those with the special prefix. (See Sections 15.4.1 and 15.4.1.1.)

177764 P.TRAP This word contains the trap processing address for all TRAP instructions. (See Section 15.4.1.)

177766 P.FPP This word contains the trap processing address for all exception traps from the FPP (or FPU) hardware floating point unit. (See Section 15.4.2.)

177770 P.CC This word contains the processing address used when a Control-C character is typed at any terminal accepting input for the job. (See Section 15.4.2.)

177772 P.2CC This word contains the processing address used when two Control-C characters are typed in quick succession at any terminal accepting input for the job. (See Section 15.4.2.)

177774 P.SIZE This word contains the maximum job image size, in K-words, for a user of this run-time system. It is used to check the validity of a user's request to change its size. The value must be an integer between 1 and 28(10). The effective upper limit is 32, minus the size of the run-time system rounded up to a multiple of four. Thus, a run-time system that requires five K-words of space could set an upper limit here of 24 (32-8). (The value here could be smaller, however.)

The RSX run-time system is an exception to this rule. When the emulator is installed as part of the monitor, an executing program can expand to a full 32 K-words (see Section 15.2.2). Thus, in this case, the value in this word can be 32.

177776 This word is reserved for future use.

## 15.4 Trap Processing

The term "pseudo-vector," as applied to the uppermost locations of a job's high segment, arises from the relationship of the (one-word) trap addresses within the region to some of the (two-word) trap vectors in low memory.

As dictated by the PDP-11 hardware architecture, every device (along with several machine language instructions) capable of generating a processor interrupt has at least one interrupt "vector": a unique pair of locations (two words) reserved in the low end of physical memory. The first word contains the location of the kernel mode service routine; the second word contains the processor status word that is to be used by the service routine. When the monitor receives control as a result of a trap through certain of these low-memory trap vectors, it passes control on to the run-time system at the corresponding address specified in the pseudo-vector region.

There are two types of traps: synchronous system traps (SSTs) and asynchronous system traps (ASTs).

### 15.4.1 Synchronous System Traps (SSTs)

A synchronous system trap (SST) is always the direct result of some action taken by the user job — usually the execution of one of the interrupt generating machine language instructions.

When the monitor passes control to the run-time system as the result of an SST, the top two words on the user stack contain the location at which the trap occurred (the user job's program counter) and the value of the user processor status word (PSW).

In general, the run-time system can process an SST in any way it wishes. If execution of the user job is to continue after completion of trap processing, a "return from interrupt" (RTI) instruction terminates the service routine and returns control to the point at which execution was suspended when the trap occurred.

There are six synchronous system traps that can be processed by a run-time system.

- *Hardware floating-point instruction set traps.* When an instruction from this set is executed that causes a trap, the monitor passes control to the run-time system at the location specified in P.FIS. The run-time system can process the trap in any way appropriate.
- *Breakpoint instruction and T-bit traps.* When the job issues a breakpoint instruction (BPT) or a T-bit trap occurs, the monitor passes control to the run-time system at the location specified in P.BPT. Breakpoint and T-bit traps are typically used to implement debugging programs.
- *IOT instruction traps.* When the job issues an IOT instruction, the monitor passes control to the run-time system at the location specified in P.IOT. The run-time system can process the trap in any way appropriate.
- *Emulator traps.* When the job issues an emulator trap instruction (EMT) requesting a non-monitor service, the monitor passes control to the run-time system at the location specified in P.EMT. If the PF.EMT bit is set in location P.FLAG, control is transferred here for all EMT instructions except those preceded by the special EMT "prefix" found in the low byte of P.FLAG. (See Section 15.4.1.1 for a discussion of the EMT prefix feature.) Processing of each EMT depends on the functional specification of those services supported by the individual run-time system.
- *TRAP instruction traps.* When the job issues a TRAP instruction, the monitor passes control to the run-time system at the location specified in P.TRAP. The run-time system can process the trap in any way appropriate.

- *Synchronous error traps.* The monitor passes control to the run-time system at the location specified in P.BAD whenever one of the following error traps occurs:
    - Reserved instruction trap. The job attempted to execute an instruction reserved for kernel mode.
    - Odd address trap. The job attempted to reference a word using an odd address.
    - Memory management unit exception trap. The job attempted to access a location outside its virtual address space.

  An error code is returned in the job's file request queue block (FIRQB) for each of these traps, permitting the run-time system to determine which error caused the trap. These errors are generally considered nonrecoverable and are processed accordingly.

  Note that certain asynchronous error traps are also processed at the location specified at location P.BAD (see Section 15.4.2).

### 15.4.1.1 Special EMT Prefix Feature -

Under RSTS/E, user calls for system services are made using the emulator trap instruction (EMT).* Normally, EMT instructions fall into two classes: direct monitor calls and non-monitor calls. The code to process a direct monitor call resides in the monitor itself, whereas non-monitor calls are processed by the run-time system.

Direct monitor calls are all issued using an EMT instruction where the low byte is an even number in the range of 0 to 76(8). When such an instruction is executed, control transfers to the monitor, the call is processed, and control returns to the instruction following the EMT. An EMT instruction with an odd value in the range 1 to 75(8) in the low byte, or any value in the range 77 to 377(8), also transfers control to the monitor. However, the monitor immediately transfers control back to the run-time system for processing at the location specified at word P.EMT of the pseudo-vector region.

In some circumstances, the run-time system itself can process or emulate certain direct monitor calls. In this case, the run-time system indicates that it wishes to process EMTs that are normally handled by the monitor by setting a bit in the flag word (P.FLAG) of its pseudo-vector region. However, if there are then certain EMTs that the run-time system should *not* process, the run-time system must use what is known as the special EMT "prefix" feature to force the monitor to process those calls.

To use this feature, the run-time system defines a special "prefix" value in the low byte of P.FLAG in the pseudo-vector region. Then any monitor calls that are preceded by an EMT using this prefix value are not returned to the run-time system but are processed by the monitor in the usual fashion. When the run-time system is using this feature, EMT processing proceeds as follows:

- All EMTs with a low byte not equal to the prefix value are returned to the run-time system for processing.

* Within assembly language programs, user calls for system services are generally invoked through the use of macros that expand to a series of instructions including the appropriate EMT. Higher level languages (which do not support macros and/or trap facilities) generally use subroutine calls to the run-time system and the EMT instruction is executed from there.

- If the low byte of the EMT is equal to the special prefix, the monitor examines the next instruction. If is also an EMT, it is processed in the normal fashion — a monitor EMT is processed by the monitor and a non-monitor EMT is returned to the run-time system.
- If the next instruction is *not* an EMT, control is passed to the run-time system.

Note that this feature is not based completely on the value of the prefix but depends instead on two *consecutive* EMT instructions, with the first one having a low byte equal to the special prefix. The run-time system does not "lose" the use of an EMT by using this feature. To issue the EMT normally associated with the prefix, the software should issue a single EMT if the call is to be processed by the run-time system or two consecutive EMTs if the call is to be processed by the monitor.

### 15.4.2 Asynchronous System Traps (ASTs)

An asynchronous system trap (AST) is the result of one of two types of conditions: some event external to the job itself (such as a user typing Control-C at the terminal) or some internal but asynchronous process (such as an error in the hardware floating-point unit whose execution overlaps that of the central processor).

As with an SST, when the monitor passes control to the run-time system as the result of an asynchronous system trap, the top two words of the user stack contain a program counter (PC) and a processor status word (PSW). In this case, however, the PC and PSW do not refer to the instruction that caused the trap but to the instruction that was executing when the trap occurred.

In general, the run-time system can process an AST in any way it wishes, subject to any standard processing already done by the monitor. As with the processing of SSTs, returning to the user can be accomplished with the "return from interrupt" (RTI) instruction.

There are four asynchronous system traps that can be processed by a run-time system.

- *Hardware floating-point unit exception traps.* When the FPP (or FPU) hardware floating-point unit takes an exception trap, the monitor passes control to the run-time system at the location specified in P.FPP. Since the floating-point exception code (FEC) and the floating-point exception address (FEA) of the unit are not otherwise accessible, the monitor pushes these two data items onto the user job's stack along with the program counter and processor status word. The run-time system can process this trap in any way appropriate. The FEC and FEA must be removed from the stack before executing the RTI instruction.
- *Control-C terminal input.* When a Control-C is encountered in any terminal input being accepted by the job, the monitor cancels all pending character output for the job and passes control to the run-time system at the location specified in P.CC. The run-time system can processs the Control-C in any way it wishes. Typically, the job is aborted unless the user program has indicated that it wishes to process Control-C traps itself. (See the *RSTS/E System Directives Manual.*)
- *Double Control-C terminal input.* When a second Control-C is encountered in the terminal input before the run-time system has been able to respond to the first, the monitor cancels all pending character output for the job and passes control to the run-time system at the location specified in P.2CC. The run-time system can process this trap in any way it wishes.

- *Asynchronous error traps.* The monitor passes control to the run-time system at the location specified in P.BAD whenever one of the following error traps occurs:
  - Stack overflow. The user job's stack overflowed.
  - Disk error. A fatal disk error occurred when the user's job image was being swapped. The original contents of the job image are lost.
  - Parity error. A memory parity fault occurred in the user's job image. The original contents of the job image are lost.

  Before passing control to the run-time system, the monitor refreshes the job's keyword and resets the stack pointer (SP) to the value USRSP (the default stack location). An error code is also returned in the job's file request queue block (FIRQB), permitting the run-time system to determine which error caused the trap. These errors are generally considered fatal and no recovery is really possible.

  Note that certain synchronous error traps are also processed at the location specified in P.BAD (see Section 15.4.1).

## 15.5 Entry Points

The monitor transfers control to the run-time system whenever certain major transition points are reached for the job (such as when the user types a "RUN" command to begin execution of a new program). The entry point addresses of the processing routines for these transitions are specified in the pseudo-vector region.

There are four standard entry points included within a run-time system.

- *System startup entry point.* When the system is started up, the monitor transfers control to the system default run-time system at the location specified in P.STRT within the pseudo-vector region. Processing is completely up to the run-time system.
- *System crash restart entry point.* When the entire system is to be restarted after a crash, the monitor transfers control to the system default run-time system at the location specified in P.CRAS within the pseudo-vector region. Processing is completely up to the run-time system.
- *New user entry point.* Whenever a job switches from one run-time system to another, the monitor passes control to the new run-time system at the location specified in P.NEW. This entry point is most commonly used when a user program terminates execution and control of the job is switched back to the job's default run-time system (keyboard monitor). It is also entered when a user program issues a monitor directive to change controlling run-time systems.

  Unless specifically instructed not to, the monitor does some general "housekeeping" before transferring control to the new run-time system. Specifically, this includes resetting the user's stack pointer to its default location (USRSP), refreshing the job's keyword, and loading the job's job number, multiplied by two, into the file request queue block (FIRQB). The transfer request block (XRB) is also loaded with additional information concerning the switch of control. (See the *RSTS/E System Directives Manual* for details concerning entry conditions of the P.NEW entry point.)

- *New program entry point.* When an executable program is to be run as the result of a "RUN" command issued by the user, the monitor transfers control to the run-time system at the location specified in P.RUN.

  Before passing control, the monitor refreshes the job's keyword and then sets any appropriate privilege bits (see Section 5.1.1). It also opens the file to be run but does not read it. It is the responsibility of the run-time system to load the program, initialize it and execute it. Any additional information (such as entry conditions, accounting data, file size, and so on) is passed in the job's XRB and FIRQB. (See the *RSTS/E System Directives Manual* for details concerning entry conditions of the P.RUN entry point.)

# CHAPTER 16
# Resident Libraries [V8.0]

Under RSTS/E, a resident library is a collection of shareable subroutines or data areas that have been linked together into one disk file and converted into a save-image library, or SIL. When in use, a resident library occupies an assigned "region" (or contiguous portion) of physical memory that is then available to be mapped and accessed as part of a user job's virtual address space.

This chapter presents a general overview of the resident library feature of RSTS/E. Section 16.1 discusses some general concepts. Section 16.2 gives an overview of the program logical address space (PLAS) directives used in the process of accessing resident libraries by user jobs, and Section 16.3 presents the details of the associated monitor control structures.

## 16.1 General Concepts

Support of the resident library feature under RSTS/E is a system generation option..

A resident library can contain shareable code, shareable data, or both. It can be designated read-only or read/write and can also have access requirements that must be satisfied before access rights can be granted. (These requirements are basically the same as the requirements to access the resident library disk file based on file protection code.) Unlike a run-time system, a resident library contains no job control or monitor-related data structures.

### 16.1.1 Library Mapping Within the User Job

Within the user job's address space, the user program (or low segment) occupies the lowest virtual addresses, starting with 0. The run-time system (or high segment) occupies the highest virtual addresses, ending at 177774(8). The virtual address space between the end of the user program and the beginning of the run-time system (the middle segment) is then available for use by resident libraries attached to the job. Note, however, that since the memory management hardware allocates virtual memory in four K-word segments, the actual available size of this middle segment can be significantly less than the remaining virtual memory. Any virtual memory between the end of the user program and the next four K-word boundary is unavailable. The same is true for the virtual memory between the beginning of a four K-word boundary and the beginning of the run-time system.

Since resident libraries are also mapped into the job's address space using the memory management hardware, they also occupy virtual memory in four K-word increments. Any address space between the end of the resident library and the next four K-word boundary is unused.

### 16.1.2 Installing a Library

Before a resident library can be accessed by a user job it must be installed within the system and added to the monitor's list of available resident libraries. This list is empty at system startup, and all libraries must be added at the start of each timesharing session.

The RSTS/E memory manager (see Section 4.2.1) is not set up to determine load addresses for anything except user programs and run-time systems. Thus, when installing a resident library, a specific load address must be given. The UTILTY program (used for this installation process) will not decide where the library should reside. It will, however, verify that the address specified is valid and can be used without fragmenting the library. (Remember that resident libraries must occupy a *contiguous* portion of physical memory.)

A resident library can be defined as permanently resident, or it can be defined so that it is loaded from disk when a job attaches to it and remains in memory only as long as at least one job is using it. Note, however, that a nonpermanent resident library can be "swapped out" temporarily if all the jobs currently attached to it are also swapped out and the memory is needed for something else. Libraries are not "swapped out" in the standard sense, however. Like run-time systems, they are simply overwritten and then reloaded from disk.

### 16.1.3 Program Access

To access a resident library a user job must attach itself to the region of physical memory in which the library resides. This process can be built into the user program by the task builder utility (TKB) when the program's executable disk file is compiled. Or, using the progam logical address space (PLAS) directives provided by the RSX run-time system, the user program can be coded to dynamically attach to and detach from one or more resident libraries at run time. Using these directives, the user program can also attach to an entire resident library or can selectively access an individual section.

A user job can be attached to up to five different resident libraries at once. However, if this is not done dynamically, attention must be paid to the total memory requirements. The task builder will not link a job to more libraries than will fit simultaneously in the job's virtual address space.* Thus, if a program attempts to access two libraries that together span more than the available address space, the task builder will not permit it.

However, using the PLAS directives to selectively access portions of individual resident libraries, a job can manually attach to and access multiple libraries whose combined size requires more space than is available. A job can also alternate access between different libraries using the same virtual address range, or an individual library can do so itself. Thus, a user job or library can use memory resident overlays in much the same way that phases are used to extend the effective address space of the monitor. (See Chapter 2.)

(See Section 16.2 for a more complete discussion of using the PLAS directives to dynamically access resident libraries.)

### 16.1.4 Position Independent Code

Resident libraries can be built to run at a specific virtual address, or they can be position-independent.

Libraries written to run at a specific address are generally easier to code, but they tend to restrict their own flexibility since they cannot be used in conjunction with other libraries that require the same address range. While more difficult to write, position-independent libraries are much more flexible in their application.

* Note, however, that the task builder does support the creation of "cluster" libraries that are compacted so that they share address space within the user's virtual memory.

The construction of position-independent code requires the proper use of the PDP-11's addressing modes. (See the *PDP-11 Processor Handbook* for a complete discussion of addressing modes.)

## 16.2 Program Logical Address Space (PLAS) Directives

For a job to access a resident library, the job must first attach itself to the region of physical memory that contains the library. In addition to setting up the necessary data within the job's control structures (see Section 16.3.2), attaching provides a means of verifying the user job's access rights to the library. Attaching also ensures that the resident library will not be permanently removed from memory while users are still accessing it. (The library need not be physically in memory when not being used, but it cannot be removed from the monitor's list of resident libraries until all jobs which have attached to it have subsequently detached.)

After the user job has attached to a resident library, it must create an address "window" that describes the portion of the job's virtual address space that it will use to access the code or data within the resident library. This window enables the job to address all or part of the resident library existing outside its own physical address space. The created window cannot overlap virtual addresses reserved by another window or by the user's own job image.

Finally, the user job must request that tfe monitor map all or part of the created window into all or part of the resident library. Once the user job has completed these three operations, it can directly access code or data within the resident library.

Under RSTS/E, support for these operations is provided by a subset of the RSX-11M program logical address space (PLAS) directives, emulated by the RSX run-time system. The use of optional features of the task builder utility (TKB) will automatically set up these calls for a user program. However, as mentioned in Section 16.1.3, there are certain restrictions inherent in this procedure. By executing these directives explicitly, instead of relying on the automatic features of the task builder, a MACRO program can control the mapping of windows within the job more flexibly and more completely.

Six of the RSX-11M PLAS directives are emulated. These are as follows:

- *Attach region.* The "attach region" directive attaches a user job to a resident library. The resident library must have been previously installed on the system. The attach marks the resident library in use and prevents its removal while the job is running. However, as long as the job currently executing is not actually mapped to the library, the resident library can be temporarily "swapped out" to make room for other jobs. (Resident libraries are not swapped out in the standard sense but are merely overwritten and then reloaded when necessary.)
  A user job can attach to a maximum of five resident libraries at one time.
- *Detach region.* The "detach region" directive detaches the job from a previously attached resident library. Any of the job's address windows that reference the detached library are automatically unmapped and eliminated.
- *Create address window.* The "create address window" directive creates a virtual address window by allocating an address window sub-block in the job's window descriptor block (see Section 16.3.2). An address window must be created in order to map virtual address space in the job's address space to all or part of an installed

resident library. Any existing windows that overlap the specified range of virtual addresses are unmapped (if necessary) and then eliminated.

An address window can be created and mapped at the same time or simply created, to be mapped later.

A user job can create a maximum of seven address windows.

- *Eliminate address window.* The "eliminate address window" directive deletes an existing address window, unmapping it first if necessary. Further references to the window or to the virtual addresses described by the window are impossible (and will generate an error) without recreating the window and remapping it.
- *Map address window.* The "map address window" directive maps all or part of an existing virtual address window to all or part of an attached resident library. The mapping begins at a specified offset from the start of the library. If the window is already mapped elsewhere, the window is unmapped before any remapping is done.

  A user job can map a maximum of seven address windows.
- *Unmap address window.* The "unmap address window" directive unmaps a specified address window, eliminating any previous resident library mapping context but retaining the window itself. The window must be remapped before access to the described virtual address space is possible.

(See the *RSTS/E System Directives Manual* for a more complete discussion of how to use these PLAS directives.)

## 16.3 Monitor Control Structures

Every resident library installed on the system is described by a resident library descriptor block (LIB). This block is allocated from the monitor's general small buffer pool when the resident library is installed and is returned to the buffer pool when the library is removed from the system. The library descriptor blocks of all installed resident libraries are linked together in a singly linked list, pointed to by moniitor location LIBLST.

When an active job attaches to a resident library, a window descriptor block (WDB) is created. This block is pointed to by the job's secondary job data block (JDB2) and contains information concerning the job's current resident library mapping. It is also allocated from the monitor's general small buffer pool and is returned when the job terminates.

### 16.3.1 Resident Library Descriptor Block (LIB)

As mentioned, every installed resident library has an associated resident library descriptor block (LIB). This block has essentially the same format as the run-time system descriptor blobk (RTS) (see Section 15.2.1), with a few minor exceptions. Note that many of the offsets are even defined with the same mnemonics.

Figure 16-1 shows the format of the resident library descriptor block. Following the figure is a description of each entry.

| *Offset* | *Contents* |
|---|---|
| R.LINK | This word contains the address of the next resident library descriptor block in the system list of resident libraries (LIBLST). If this entry is the last in the list, this word contains a value of zero. |
| R.NAME | These two words contain the name of the resident library, in RAD50 format. The library is stored on disk using this file name with a file type of "LIB". |

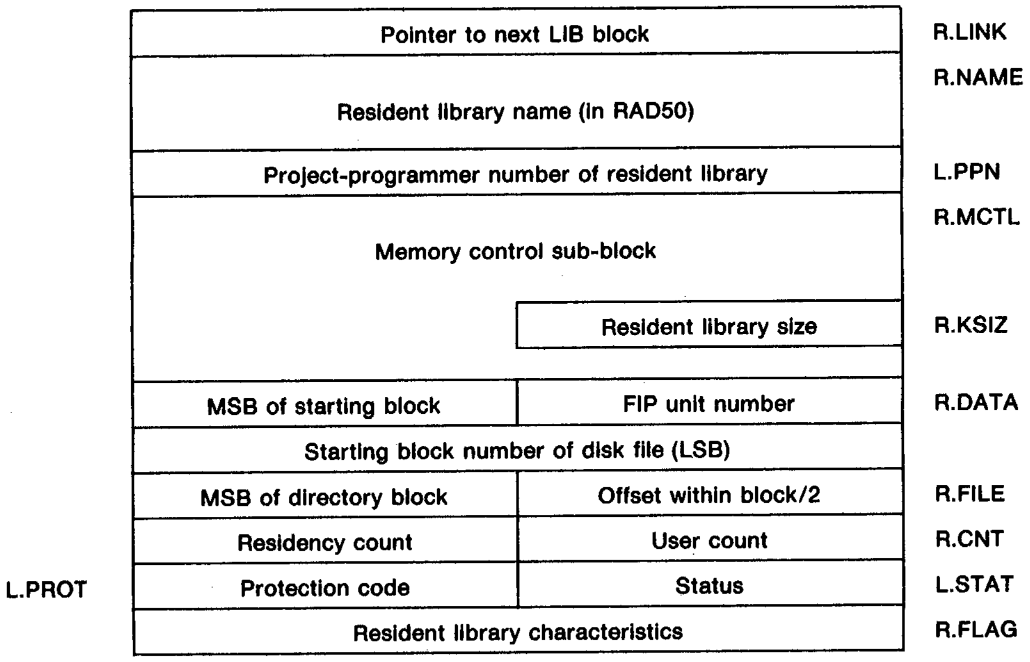


**Figure 16-1:** Resident Library Descriptor Block

| | |
|---|---|
| L.PPN | This word contains the project-programmer number (PPN) under which the resident library's disk file is stored. The high byte contains the project number and the low byte contains the programmer number. |
| R.MCTL | This 5-word area is the memory control sub-block for the resident library. (See Chapter 4 for a discussion of memory control and the memory control sub-block.) |
| R.KSIZ | This byte (within the memory control sub-block) contains the size of the resident library, in K-words. |
| R.DATA | This byte contains the FIP unit number of the disk on which the resident library is stored. |
| R.DATA+1 | This 3-byte area contains the 24-bit FIP block number (FBN) of the first block of the resident library's image on disk. The byte at R.DATA+1 is the most significant byte of this value. |

NOTE

All resident libraries are stored on disk in contiguous files. Thus, given the starting block number and the size of the file (derived from R.KSIZ), the entire library can generally be loaded with one disk access. If the library is longer than 31 K-words, however, the memory manager will break the file into 31 K-word segments for loading.

R.FILE This byte contains the offset, divided by two, into the directory block to the name entry of the resident library's disk file. This value is used to close the file when the resident library is removed from the system.

R.FILE+1 This 3-byte area contains the 24-bit FIP block number (FBN) of the directory block containing the name entry of the resident library's disk file. The byte at R.FILE+1 is the most significant byte of this value.

R.CNT This word contains the access counter for the resident library. The high byte is used as a map counter. It is incremented every time a user maps a portion of the library and is decremented every time a user unmaps a portion of the library. If this byte contains a zero, the resident library is not being used and can be temporarily removed from memory. If the library is to remain in memory even when there are no jobs using it, the sign bit (bit 15) is set, thereby ensuring that the count will never be zero.

The low byte is used as an attach counter. It is incremented every time a user attaches to the library and is decremented every time a user detaches from the library. A resident library cannot be removed from the monitor's list of installed libraries unless this byte is zero.

L.STAT This byte contains one status bit (bit 7). Its primary purpose is to permit a library descriptor block to be differentiated from a run-time system descriptor block. This bit is always set in a library descriptor block. The corresponding byte in an RTS block contains the minimum size of a job executing under control of the run-time system. Since this value can never exceed 32, this bit is never set in an RTS block.

L.PROT This byte contains the protection code for the resident library and is used to determine what access rights a user can have to the library. This code follows the same conventions as the RSTS/E disk file protection codes (see Section 8.5.2). If no protection code is specified when the library is installed, the default code is 42 — public read only.

R.FLAG This word contains a set of bit flags that define the characteristics of the resident library. These bits are essentially a subset of those used for run-time systems. The setting of each of these bits within the LIB are as follows:

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-8 | | (Unused) |
| 9 | PF.1US | The resident library is nonshareable and can only be accessed by one user. |
| 10 | PF.RW | The resident library can be mapped read/write if requested by a user with the appropriate privileges. |
| 11 | | (Unused) |

12 PF.REM The resident library should be removed from memory immediately when the access count at offset R.CNT goes to zero.

13 (Unused)

14 PF.SLA The resident library should be loaded at the address specified in the memory control sub-block. This bit is always set for a resident library since all libraries are loaded into a unique, predetermined location in memory. Although this is not a necessary requirement for the proper implementation of the resident library feature, it is done to limit the amount of work the memory manager must perform in order to make a user job resident and runnable.

15 (Unused)

### 16.3.2 Window Descriptor Block (WDB)

When a job is attached to a resident library, its secondary job data block points to what is known as a window descriptor block (WDB). This WDB contains information concerning the libraries to which the job is (or ultimately will be) attached, along with information concerning any mapping windows that exist for the job.

The WDB is created and information is stored in it in conjunction with the execution of various program logical address space (PLAS) directives. When the job first attaches to a resident library, the window descriptor block is allocated from the monitor's general small buffer pool and a pointer to the library descriptor block (LIB) of the attached resident library is loaded into the WDB. When an mapping window is created, the corresponding address window sub-block is set up with the appropriate mapping information (virtual address and size). When the window is ultimately mapped to all or part of a resident library, a pointer to the corresponding LIB block is loaded into the corresponding address window sub-block.

The first window descriptor block created for a job can contain up to two address window sub-blocks. If more windows are created, additional WDBs are created to hold the extra window sub-blocks. A job can have up to seven address windows.

Figure 16-2 shows the format of the first window descriptor block for a job. Each item is discussed below.

*Offset* *Contents*

W.LINK This word contains a pointer to the second window descriptor block (if any) for this job.

W.ALIB This 5-word area contains pointers to the descriptor blocks of the resident libraries to which this job is currently attached. A job can attach to a maximum of five libraries at one time. Any unused words within this area contain a value of zero.

W.WIN1 This 5-word area contains the job's first address window sub-block.

W.WIN2 This 5-word area contains the job's second address window sub-block.

| | |
|---|---|
| Pointer to the next WDB for the job | W.LINK |
| Pointer to LIB #1 | W.ALIB |
| Pointer to LIB #2 | |
| Pointer to LIB #3 | |
| Pointer to LIB #4 | |
| Pointer to LIB #5 | |
| Address window sub-block #1 | W.WIN1 |
| Address window sub-block #2 | W.WIN2 |

**Figure 16-2:** First Window Descriptor Block

Figure 16-3 shows the format of an address window sub-block. Each item is discussed below.

| | | | |
|---|---|---|---|
| W$NSTS | Window status | Base APR | W$NAPR |
| | Window size, divided by 64. | | W$NSIZ |
| | Pointer to library descriptor pointer | | W$NLIB |
| | Offset into library, divided by 64. | | W$NOFF |
| | Mapped length | | W$NBYT |

**Figure 16-3:** Address Window Sub-Block

| *Offset* | *Contents* |
|---|---|
| W$NAPR | This byte contains the number of the active page register (APR) to be used for the base of this window. Values can range from 1 to 7. |
| W$NSTS | This byte contains flag bits indicating the status of the address window. These bits are as follows: |

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 8 | WS$WRT | If this bit is set, the job has requested write access to the address window. |
| 9-14 | | (Reserved for future use) |

15 WS$MAP If this bit is set, the address window is currently mapped.

W$NSIZ This word contains the size of the address window, divided by 64(10).

W$NLIB This word contains a pointer into the area at offset W.ALIB (see Figure 16-2). That is, it points to the address of the LIB of the resident library to which this window is mapped. If the window is not currently mapped, this word contains a zero. (Note that this is a pointer to a pointer. It does *not* contain the address of the LIB itself.)

W$NOFF This word contains an offset (divided by 64(10)) into the resident library to the start of the mapped portion. If the window is not currently mapped, this word contains a zero.

W$NBYT This word contains the length of the mapped portion of the resident library, in bytes. If the window is not currently mapped, this word contains a zero.

If a job creates more than two address windows, an additional window descriptor block is allocated and linked to the first. If more than five windows are created, a third descriptor block is allocated and linked to the second.

Figures 16-4 and 16-5 show the formats of the second and third window descriptor blocks (respectively). The format of each address window sub-block within these descriptor blocks is identical to those in the first window descriptor block.

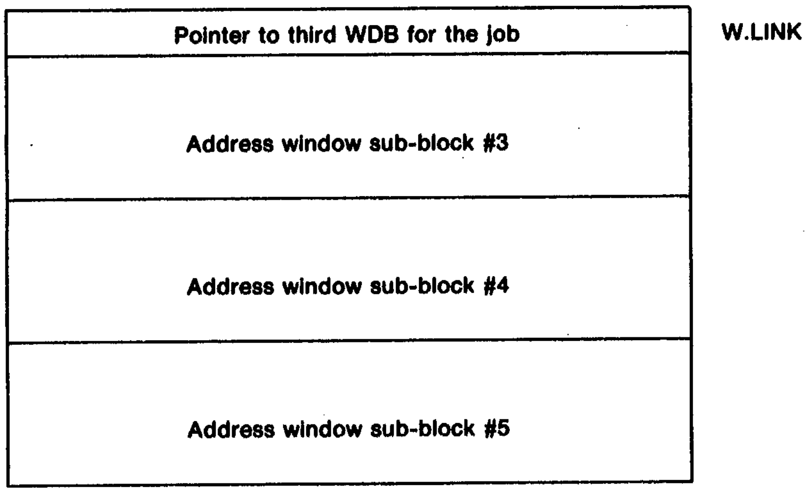

Pointer to third WDB for the job
W.LINK
Address window sub-block #3
Address window sub-block #4
Address window sub-block #5

**Figure 16-4:** Second Window Descriptor Block

| | |
|---|---|
| Always 0 | W.LINK |
| Address window sub-block #6 | |
| Address window sub-block #7 | |
| Unused | |

**Figure 16-5:** Third Window Descriptor Block

# Part VI
# SYSTEM GENERATION AND INITIALIZATION

Before a RSTS/E operating system can be used for timesharing, it must be built to fit the hardware configuration and software requirements of the particular environment in which it is to run. This process, known as "system generation" (or SYSGEN), involves several steps that create and tailor the monitor, the default run-time system, and the system utility library.

System generation itself is designed and implemented as a series of application programs running within the RSTS/E environment. Thus, to generate a RSTS/E system requires an *existing* RSTS/E system. This apparent conflict is resolved by using a special-purpose monitor (SYSGEN.SIL), configured for a minimal set of hardware and tailored specifically to provide support for the system generation process.

As with all RSTS/E monitors, SYSGEN.SIL must be installed before it can be used. This is done with the system initialization program (INIT.SYS), a stand-alone program used to configure and initialize the overall RSTS/E environment.

This part of the *RSTS/E V8.0 Internals Manual* presents a general overview of the system generation procedure (Chapter 17) and of the system initialization program (Chapter 18). Note that the intent here is not to present detailed instructions on how to actually execute a system generation or use INIT.SYS. Detailed instructions for doing so can be found in the *RSTS/E System Generation Manual.*

# CHAPTER 17
# System Generation [V8.0]

The system generation process involves four basic steps. First, using the intialization code, the SYSGEN system is tailored for the particular installation. Once the SYSGEN monitor has been tailored, installed and started, the next step is to generate the target system by running the various pieces of the actual SYSGEN application. After generating the target system, the SYSGEN monitor is shut down and the initialization code is used again to tailor the target system. The final step is to use the BUILD program to generate the system library.

The remainder of this chapter presents an overview of these steps.

## 17.1 Tailoring the SYSGEN System

The first step in the system generation process is to use the hardware bootstrap loader to read the first record of the RSTS/E distribution medium (either disk or tape) into memory at location 0. This record (known as the "secondary bootstrap") contains software that is executed immediately after it is loaded. On RSTS/E systems, the secondary bootstrap is designed to load and execute the system initialization code (INIT.SYS). (See Section 8.6.1 for a detailed discussion of the function and structure of the secondary bootstrap.)

INIT.SYS is a stand-alone utility program used to create a functional RSTS/E environment by creating the RSTS/E file structure, system files and startup conditions necessary for timesharing. During the system generation process, INIT.SYS is used to tailor the SYSGEN system for the particular installation. The following steps are performed using INIT.SYS functions:

- The SYSGEN system disk is initialized.
- System files are copied from the distribution medium to the SYSGEN system disk.
- Corrections (that is, patches) are made to the initialization code, the SYSGEN monitor, and the RT11 run-time system, if necessary.
- Device controllers are disabled, if necessary, and bus addresses are specified for any nonstandard devices.
- The file SYSGEN.SIL is installed as the RSTS/E monitor for the duration of the system generation process.
- Swapping files, as well as other required system files, are allocated on the SYSGEN system disk.
- Defaults are established for the SYSGEN monitor. These include the job and swap maxima, the primary run-time system, the error message file, the installation name, the allocation of memory space, the crash dump facility, and the labelling format for magnetic tapes.
- Device characteristics are specified and device units are disabled, if necessary.
- Timesharing is started on the tailored SYSGEN system.

NOTE

These functions are discussed in more detail in Chapter 18.

## 17.2 Generating the Target System

After tailoring and starting the SYSGEN system, the next step in the system generation process is to generate the target system. This involves copying the required files from the distribution medium to the SYSGEN disk, determining the target system configuration, and assembling and linking the components of the RSTS/E target monitor.

### 17.2.1 Copying the Required Files

The RSTS/E distribution medium contains a program (CREATE.SAV) that generates a minimal system disk from which the system generation application can be run. This program first enables logins for the SYSGEN sytem. It then copies the LOGIN.SAV program from the distribution medium to the SYSGEN disk and logs in on a pseudo-keyboard under account [1,2]. Once it is logged in, CREATE.SAV uses a batch stream to run PIP and copy the following files to the SYSGEN disk:

- LOGIN, LOGOUT, and UTILTY (special SYSGEN versions that run under RT11).
- PIP.
- MACRO — used assembles the terminal driver (TTDINT and TTDVR) and the monitor tables (TBL); all other system modules are preassembled before shipment.
- CREF — creates cross-reference listings for the modules assembled with MACRO.
- LINK — links modules of both the RSTS/E monitor and the BASIC-PLUS run-time system.
- SILUS — creates save-image libraries (SILs).
- HOOK — links the initialization code object file into the secondary bootstrap of the target disk, making it possible to bootstrap the target system.
- ERR.STB — defines the error symbols for linking the monitor and the BASIC-PLUS run-time system.
- ONLPAT — installs and verifies patches to the system code.
- SYSGEN.SAV — asks the target system configuration questions and builds the batch file necessary for actual system generation.
- SYSBAT.SAV — executes the commands in the SYSGEN batch file (created by SYSGEN.SAV) and generates the actual monitor and BASIC-PLUS run-time system.

After all the necessary files have been copied to the SYSGEN disk, CREATE.SAV logically dismounts the distribution medium and chains to the dialog program, SYSGEN.SAV.

### 17.2.2 Determining the Target System Configuration

When CREATE.SAV has copied all the required files to the SYSGEN disk, it chains to SYSGEN.SAV, the system generation dialog program, which asks the configuration questions. Answering the configuration questions involves declaring permanent settings in the RSTS/E monitor code and specifying the primary (default) run-time system. These settings establish the hardware devices and optional software elements that the target system will support.

Note that configuring a system is different from tailoring a system. Tailoring a system involves specifying *variable* factors for the RSTS/E monitor. These factors can be changed (within their configured limits) at any time that the initialization code is running. Configuring a system, on the other hand, establishes what software features are to be included or excluded from the monitor, as well as the size of various internal structures and the resulting memory allocation. Thus, these factors are permanent and cannot be changed without regenerating the total monitor.

As SYSGEN.SAV receives answers to the configuration questions, it builds a configuration file (CONFIG.MAC) and a batch control file (SYSGEN.CTL) to be used in the assembly and linkage step. When it has asked and received answers for all configuration questions, SYSGEN.SAV prints a message instructing the user to edit the files (if necessary) and then begin the batch process by running program SYSBAT.SAV.

### 17.2.3 Assembling and Linking the Monitor Components

Program SYSBAT.SAV is a batch process that executes commands in the batch control file (SYSGEN.CTL) generated by SYSGEN.SAV. These commands generate the monitor save-image library (SIL) and (optionally) the BASIC-PLUS run-time system.

The first step in the batch process consists of invoking MACRO to assemble the monitor tables and the terminal service. The monitor tables (TBL) are assembled using the configuration file created by SYSGEN.SAV (CONFIG.MAC) as a prefix file. Thus, the various monitor tables are tailored to the specific installation, resulting in the optimum use of space within the monitor's permanently mapped root. This ensures that as much memory as possible is available for the general small buffer pool.

The terminal service driver code (TTDVR) and data module (TTDINT) are also assembled at this time. Because of the large number of optionally supported hardware and software features available with terminal service, these modules are distributed in source form and then assembled using CONFIG.MAC as a prefix file. As with TBL, this ensures that the resulting terminal service software is tailored to the specific installation and includes no unnecessary processing code or data structures.

These three modules (TBL, TTDVR, and TTDINT) are the only monitor components that are distributed in source form. All other components are distributed preassembled. The various optional features are included or excluded simply by changing the way in which the monitor is linked.

Once all the necessary assemblies are done, SYSBAT.SAV invokes LINK to link individual object modules into the separate phases and SILUS to link the phases into one save-image library. During this process, the monitor's permanently mapped root (that is, the "RSTS" phase) is linked first. The symbol table generated from this first link is used as an input file for subsequent links, allowing the overlaid phases to access locations within the root directly.

The output from SILUS is the monitor save-image library.

When SYSBAT.SAV completes processing the batch control file, it prints an informative message and returns to monitor level. The SYSGEN system is then shutdown by running the UTILTY program. The shutdown process automatically bootstraps the system initialization code (INIT.SYS) from the SYSGEN disk, permitting the tailoring of the generated target system.

### 17.2.4 System Generation During Normal Timesharing

Additional RSTS/E monitors can be generated during timesharing using the RT11 run-time system. The SYSGEN monitor is not required. After ensuring that all the system generation programs are on the system disk, the SYSGEN.SAV program is run to ask the configuration questions and build the configuration file and system generation batch file. SYSGEN.SAV allows generation of an entire system, only a monitor, or only the BASIC-PLUS run-time system.

When SYSGEN.SAV completes, the batch file is processed by running SYSBAT.SAV. This generates the monitor SIL and/or BASIC-PLUS run-time system.

When SYSBAT.SAV terminates, the new monitor save-image library can be tailored at some later time, after the current timesharing session has ended.

## 17.3 Tailoring the Target System

After the target system is configured, it must be tailored. This is done by bootstrapping the SYSGEN disk and then using the system initialization code (INIT.SYS) to tailor the system. This is the same process used to tailor the SYSGEN system.

Using INIT.SYS, the following operations are performed:

- The target system disk is initialized.
- System files are copied from the SYSGEN disk to the target system disk.
- Corrections (that is, patches) are made to the system code, if necessary.
- Device controllers are disabled, if necessary, and bus addresses are specified for any nonstandard devices.
- The target monitor save-image library (SIL) is installed as the RSTS/E monitor.
- Swapping files, as well as other required system files, are allocated on the target system disk.
- Defaults are established. These include the job and swap maxima, the primary run-time system, the error message file, the installation name, the allocation of memory space, the power-fail delay and crash dump facilities, and the labelling format for magnetic tapes.
- Device characteristics are specified and device units are disabled, if necessary.

NOTE

These functions are discussed in detail in Chapter 18.

## 17.4 Building the System Library

Once the target system is created and tailored, the RSTS/E system utility library can be built and any auxiliary run-time systems or optional software can be installed.

The RSTS/E utility programs are included on the distribution medium along with a special batch process (BUILD) and several BUILD control files. The control files contain commands to patch (if necessary) and compile the utility programs. After retrieving and compiling the programs, BUILD places them in the system library account [1,2] on the target system disk. These programs constitute the system library.

As an alternative to compiling the set of system utilities, users can select precompiled utilities supplied in .TSK format on the distribution kit. If this option is chosen, the BUILD program generates the commands to copy the precompiled tasks from the distribution medium to account [1,2] of the target system disk.

The BUILD program can also be used to construct the utility programs for the system default run-time system if RSX or BASIC-PLUS-2 is to be used as the system default. In this case, these utility programs must be built prior to creating the system library.

Once the system library is built, any desired optional software can be installed. Additional files can be created containing user information, as well as system startup commands. User accounts are also created at this time.

The final steps in the system generation process are to start timesharing, verify the system startup control files, and store the distribution and recovery media. RSTS/E is then available for general use.

# CHAPTER 18
# System Initialization [V8.0]

The RSTS/E system initialization code (INIT.SYS) is a collection of routines that create the file structure, system files, and startup conditions required for the normal operation of RSTS/E timesharing. To perform system generation, INIT is used to load and start the SYSGEN monitor. After the system generation process is complete and the SYSGEN monitor has been shut down, INIT is used again to install the generated system (or load it onto a new disk) and bring up normal timesharing.

To bring up a RSTS/E system, several INIT features (known as "options") must be used to create the proper disk environment, install the monitor and default run-time system, and set system defaults. Beyond that, as the needs of an installation change, INIT can be used to alter system files and parameters, select a new monitor or default run-time system, and change system defaults.

INIT does extensive checking on the integrity of a system and provides options which allow the system to function when noncritical hardware is inoperative. INIT also provides a facility for patching the system software, changing device characteristics, and loading stand-alone utility programs from the disk. Finally, INIT is responsible for loading the monitor into memory and beginning timesharing.

The remainder of this chapter presents an overview of the internal structure of INIT.SYS. Section 18.1 discusses the general structure. Section 18.2 discusses the function and format of major data structures. Section 18.3 covers the function and general processing of each of INIT's primary modules, and Section 18.4 covers the processing involved in each of INIT's options.

NOTE

> Throughout this chapter, it is assumed that the reader is familiar with the RSTS/E on-disk file structure and the general concepts of RSTS/E disk I/O operations. See Part III.

## 18.1 General Structure

INIT.SYS is a large stand-alone program with many functions. Although it does not use the RT11 monitor, INIT is structured as a standard RT11 overlaid program, with only one overlay region.* It functions as its own monitor, using an interpreter to emulate the one RT11 system directive (requesting a disk read) issued by the RT11 resident overlay code.

Figure 18-1 shows the way in which INIT uses physical memory.

The root segment (up to a maximum of twenty K-words) contains the INIT mainline code, the BOOT option processor, commonly used subroutines (including the terminal service and a mini-FIP for accessing the RSTS/E file structure), and several required buffers. The root is permanently mapped using APRs 0 to 4.

* Details of the RT11 overlay structure can be found in the *RT11 System Reference Manual* and the *RT11 Software Support Manual.*

The RT11 disk-resident overlay region occupies the next eight K-words (from locations 120000 to 157776(8)). With the exception of the BOOT option (which resides in the root), all option processing routines reside in disk-resident overlays, with some overlays containing more than one option.

The memory region above location 160000(8) is used for memory-resident overlays containing code such as the disk accessing routines and the MSCP-class driver code. Any large buffers also reside here.

All INIT memory-resident overlays are restricted in size to eight K-words. They are dynamically mapped as needed using APRs 5 and 6. All disk-resident overlays are at least eight K-words but can possibly be larger. The size of the disk overlay region depends on the size of the root when INIT is built. The overlay region begins immediately after the root code, and thus, if the root is less than 120000(8) bytes, any extra memory in that segment is available for use as part of the disk overlay region.

APR 7 is used to map the I/O page.

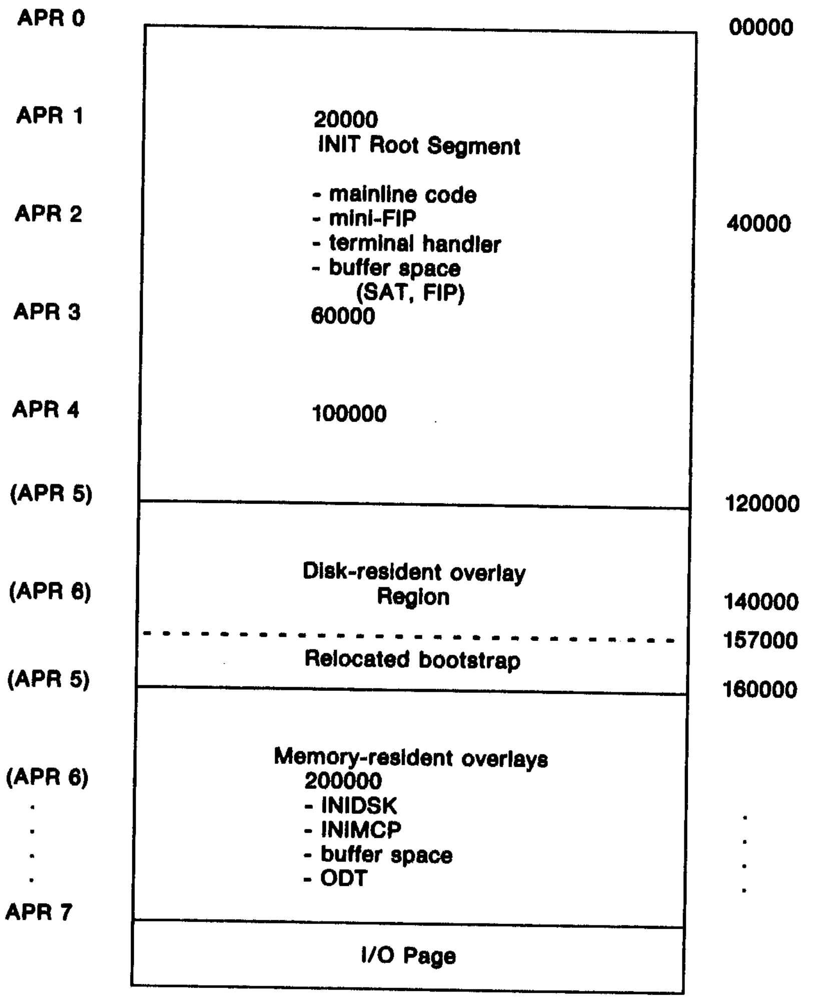


**Figure 18-1:** INIT Memory Layout

## 18.2 Data Structures

Data structures within INIT are used for several purposes: describing all devices supported under RSTS/E and their standard characteristics, specifying user-selected device characteristics (such as nonstandard bus and vector addresses), and so on. In addition, mini-FIP (see Section 18.3.5) uses several structures for controlling logical disk I/O.

### 18.2.1 Device Index Numbers

Within INIT, an individual disk or magnetic tape drive is described by two parameters: a controller index number and a unit number. Index numbers are fixed throughout the entire RSTS/E environment, with each supported controller having a unique index number. Index numbers for magtape controllers are always negative, thereby permitting any common device code to differentiate between the two types of devices.

Symbolically, index numbers have the form "IDX$xy", where "x" is either "D" for disks or "M" for magtapes, and "y" is a single character denoting the particular controller type.

The following list shows some examples of various controller index numbers. Note, however, that all these devices may not actually be supported by RSTS/E. As new controllers are introduced and obsolete ones are phased out, index numbers are added and deleted. There is generally some overlap between the time that a device is no longer supported and the time that INIT ceases to "know" about the device. Thus, this list should be viewed as illustrative only. A current comprehensive list can best be obtained by checking the INIT source listings.

| *Symbol* | *Controller Type* |
|---|---|
| IDX$MT | TU/TE16 magtape controller. |
| IDX$MM | TM11 magtape controller. |
| IDX$MS | TS11/TSV05/TU80/TK25 magtape controllers. |
| IDX$MU | All TMSCP-class magtape controllers. |
| IDX$DC | RC11/RS64 disk controller with one to four platters. The disk is treated as a single unit with all platters contiguous. |
| IDX$DF | RF11/RS11 disk controller with one to eight platters. The disk is treated as a single unit with all platters contiguous. |
| IDX$DS | RS03/RS04 disk controller with one to eight disk drives. |
| IDX$DK | RK11 disk controller. |
| IDX$DM | RK06/RM611 disk controller. |
| IDX$DP | RP11C disk controller. |
| IDX$DB | RH11 (or RH70) disk controller. |
| IDX$DL | RL11/RLV12 removable disk controller. |
| IDX$DU | All MSCP-class disk controllers. |

### 18.2.2 Parameter Buffers

As it scans the hardware and accepts user-specified settings and defaults, INIT stores autosizing information, user hardware options, and installed monitor parameters in two buffers kept in the root segment: INSBUF and DEFBUF. As parameters and settings change, these buffers are continually updated and written back to the disk.

The first buffer, INSBUF, is two K-words (eight disk blocks) long and is used to store information concerning the existing hardware configuration, as well as the name of the currently installed monitor SIL. In particular, it includes the following items:

- Installed monitor parameters, including the name of the current monitor SIL.
- Startup parameters for autorestart.
- Hardware configuration tables, including CSRTBL, VECTBL, TYPTBL, SETTBL, UMIDST, and HWTABL (see Section 18.2.3).
- VECMAP, a bit map of all vector addresses in use.
- Other pieces of device information, such as disk and tape media types and sizes.

INSBUF is written back to disk (within the file INIT.SYS) after any execution of the INSTAL or HARDWR options.

The second buffer, DEFBUF, is .5 K-words long (two disk blocks) and is an image copy of the DEFALT phase of the monitor SIL. It contains information used to set up the required system parameters when timesharing is started, along with the monitor load block used for allocating nonphase memory to permanent memory regions such as the primary run-time system and XBUF. DEFBUF is written to disk (within the monitor SIL) whenever any of the information is changed — after execution of the INSTAL or DEFALT options. Thus, defaults set for a particular monitor SIL are preserved even if the SIL is removed and then reinstalled at a later time.

### 18.2.3 Hardware Configuration Tables

When INIT is started, it determines the configuration of the PDP-11 on which it is running. To do this, it first assumes a standard configuration and then attempts to locate all devices which can be supported under RSTS/E.

#### 18.2.3.1 Device Scan Table (DEVTBL) -

The device scan table (DEVTBL) is the main table used in scanning the hardware configuration. It contains one 6-word entry (known throughout the INIT environment as a "device packet") for each possible device type supported under RSTS/E, and is terminated by a single word containing a zero. It is a read-only table.

Figure 18-2 shows the format of each 6-word entry. Following the figure is a description of each entry.

| | | | |
|---|---|---|---|
| | Maximum number of devices | | PK.MXU |
| | Offset into other tables | | PK.CSO |
| PK.NUN | Number of units | Size on I/O page | PK.SIZ |
| PK.PRI | Priority and flags | Offset to major CSR | PK.OFF |
| | Poke routine address | | PK.POK |
| | Device name | | PK.NAM |

**Figure 18-2:** Device Scan Table Entry

| *Offset* | *Contents* |
|---|---|
| PK.MXU | This word contains the maximum number of device controllers of this type that RSTS/E is able to support. Note that this is the number of *controllers* possible and not the actual number of units. |
| PK.CSO | For each device type described in DEVTBL, there are corresponding entries in several other tables used by INIT. These other tables, however, contain information on individual controllers and therefore contain multiple (PK.MXU) entries for each device type. This word contains the offset into the other tables of the entry corresponding to the first controller (controller 0) of this device type. |
| PK.SIZ | This byte contains the size of the register set for this device type — that is, the number of bytes on the I/O page allocated to each controller of this type. |
| PK.NUN | This byte contains the maximum number of units that can be connected to a controller of this type. |
| PK.OFF | This byte contains the offset from the lowest register of the device register set to the most commonly used register for this device type. For example, the lowest register of an RK11 controller is at 177400, but the most frequently referenced register is the control and status register at 177404. Thus, for RK11 type devices, this byte contains a value of 4. |
| PK.PRI | This byte contains the device priority level, along with other flags concerning this controller. The format of this byte is as follows: |

| *Bit* | *Symbol* | *Meaning* |
|---|---|---|
| 0-2 | | These three bits specify the number of UNIBUS windows required by the controller, minus one. These bits have no meaning if bit 3 (PS.UWN) is clear. |
| 3 | PS.UWN | If this bit is set, this controller requires UNIBUS windows. |
| 4 | PS.BAE | If this bit is set, this controller has a bus address extension (BAE) register. |

5-6 PS.PRI These two bits represent the processor priority level at which a controller of this type interrupts. Assuming that all devices interrupt above priority level three, INIT adds four to the value in these two bits to derive the true interrupt priority level.

NOTE

RSTS/E does not support any priority level seven devices.

7 PS.XBF If this bit is set, this controller requires mapped XBUF (see Section 5.3.1).

PK.POK This word contains the address of the device-specific routine in the INIT root that can force a device of this type to interrupt. This routine (known as the "poke" routine) is called during the vector checking operation of the INIT autosizing code.

PK.NAM This word contains the 2-character ASCII device name used by INIT to designate the controller type. To avoid conflicts within INIT itself, this name is not necessarily the same as the common RSTS/E name for the device. For example, INIT uses "RL" to designate RL11/RLV12 controllers, rather than "DL", as used by RSTS/E timesharing. Within INIT, "DL" designates the DL11-E communication device.

#### 18.2.3.2 Standard CSR Locations (CSRDEC) -

Table CSRDEC contains one 1-word entry for each possible controller specified by DEVTBL. Each entry contains the bus address of the register set assigned to the corresponding device controller. A device's register set is the set of locations (on the I/O page) assigned to a single device and which respond with slave synch when accessed. If the device's register set does not have a predetermined bus address or has a floating bus address (see Section 18.2.3.3), the corresponding entry in CSRDEC contains a zero.

CSRDEC is used to determine the system configuration, assuming no nonstandard bus address assignments are in effect. It is a read-only table.

#### 18.2.3.3 Floating Bus Address Table (FLTBA) -

Table FLTBA contains one 3-word entry for each device type with floating bus address assignments, in order of their assignment. A device with a floating address does not have a predefined bus address. Instead, when the system is configured, the device's assigned address "floats" with the presence or absence of other floating address devices. All floating devices are included in the table even though they may not be supported by RSTS/E. It is a read-only table.

Each FLTBA entry contains the size (in bytes) of the register set for the device, followed by the offset to the CSRDEC, CSRTBL, and SETTBL entries for this device type, followed by the absolute address of the entry in DEVTBL for this device type. If the device is not supported by RSTS/E, the second word of the entry contains the value -1 and the third word contains a zero.

#### 18.2.3.4 Nonstandard CSR Addresses (SETTBL) -

Table SETTBL contains nonstandard control and status register (CSR) bus addresses for devices in the hardware configuration. It is a parallel table to table CSRDEC, containing one 1-word entry for each possible device controller. For devices with a nonstandard CSR bus address (as specified by the CSR suboption of the HARDWR option), the entry contains that address. For devices with standard CSR bus addresses, or for nonexistent devices, the corresponding entry contains a zero.

SETTBL is built by the HARDWR option, and is used, in conjunction with tables CSRDEC (Section 18.2.3.2) and FLTBA (Section 18.2.3.3) to build table CSRTBL (Section 18.2.3.5).

#### 18.2.3.5 Actual CSR Addresses (CSRTBL) -

Table CSRTBL contains control and status register (CSR) bus addresses of all devices in the actual hardware configuration. It contains one 1-word entry for each possible device controller. For devices that exist, the entry contains the lowest address of the register set. For devices that do not exist, the entry contains a zero.

Since bus addresses always fall on word boundaries (and are, therefore, always even) bit 0 of each entry is used as a flag indicating the status of the controller at system startup. If the bit is clear, the device is to be enabled; if it is set, the device should be disabled.

CSRTBL is built by the INIT autosizing code from entries in CSRDEC and FLTBA. It is updated with any user-specified nonstandard bus addresses (as found in SETTBL) whenever a rescan of the hardware is executed (that is, following execution of the INSTAL or HARDWR options).

#### 18.2.3.6 Vector Addresses (VECTBL) -

Table VECTBL contains vector addresses of all devices in the actual hardware configuration. It contains one 2-byte entry for each possible device controller. For devices that exist, the low byte of each entry contains the address of the device's interrupt vector in low memory (as determined by the autosizing code). The high byte contains any nonstandard vector address (as specified by the VECTOR suboption of the HARDWR option). If the high byte contains a nonzero value, the autosizing code accepts that location as the vector address and does not attempt to make the device interrupt (see Section 18.3.2).

If a device requires two vectors, the entry contains the lower address. For devices that do not exist, the entry contains a zero.

NOTE

> RSTS/E does not require a device to interrupt at any standard vector location. A device can be supported under RSTS/E as long as it interrupts within a predefined processor priority range and vectors to a valid memory location (between 100 and 376(8)).

VECTBL is built by the INIT autosizing code and is modified by the VECTOR suboption of the HARDWR option.

#### 18.2.3.7 Device Characteristics (TYPTBL) -

Table TYPTBL contains additional device-specific information for devices in the actual hardware configuration. It contains one 1-word entry for every possible device controller.

The exact format of each TYPTBL entry is device-dependent, but generally, for each device that exists, each entry contains bit flags indicating the presence or absence of individual units, along with flags denoting the subtype (if appropriate) of each unit. For example, if the device is a disk with a disk index number of "IDX$DS", the TYPTBL entry contains eight bit flags in the low byte indicating existing disk units on the controller -- a set bit indicates that the corresponding unit exists. The high byte of the entry contains eight bit flags indicating whether the corresponding disk unit is an RS03 or an RS04 unit.

For devices that do not exist or do not require additional information, the TYPTBL entry contains a zero.

TYPTBL is built by the INIT startup code and can be modified by the HARDWR option.

### 18.2.3.8 Unit Media Overrides (UMIDST) -

Table UMIDST contains override values for disk media types. It consists of one 1-word entry for every possible disk unit in the hardware configuration. Each entry contains either a media type code of the corresponding unit or a zero.

During the autosizing code, INIT attempts to determine the type of media on each disk unit. If it is unable to determine the type (because the drive is either powered down or disconnected), INIT normally disables the unit. However, a nonzero value in the unit's UMIDST entry will override this processing and leave the unit enabled.

UMIDST is built by the UNITS suboption of the HARDWR option.

### 18.2.3.9 Miscellaneous Hardware (HWTABL) -

Table HWTABL contains a set of flag words and bytes indicating the presence or absence of other optional or required hardware. It is built by the INIT autosizing code and cannot be modified.

### 18.2.3.10 Device Status Table (DSTABL) -

The device status table (DSTABL) contains device-specific parameters for all device controllers configured into the installed monitor save-image library. The table consists of one 5-word entry for each device controller. Figure 18-3 shows the format of a DSTABL entry. Following the figure is a description of each entry.

DSTABL is created by the INSTAL option when a monitor save-image library is installed as the timesharing executive. The table is stored in a buffer in overlay DSTBLO (within file INIT.SYS) and is used later during system startup to modify locations within the RSTS/E monitor.

| | | | |
|---|---|---|---|
| DS.UNT | Unit number | DEVNAM table offset | DS.DVN |
| DS.NCS | Number of CSRs | Number of units | DS.NUN |
| DS.CTL | Controller number | Packet number | DS.PKT |
| | Pointer to CSR base | | DS.CSR |
| | Address of interrupt routine | | DS.DCT |

**Figure 18-3:** Device Status Table Entry

| *Offset* | *Contents* |
|---|---|
| DS.DVN | This byte contains the offset into the device type name table (DEVNAM) for the controller. DEVNAM contains the valid 2-character ASCII names for all devices on the system (see Section 6.2.1). |
| DS.UNT | This byte contains the RSTS/E internal unit number of the first unit on the controller. |
| DS.NUN | This byte contains the number of units that are connected to the device controller. |
| DS.NCS | This byte contains the number of control and status registers (CSRs) allocated to the controller. |
| DS.PKT | This byte contains the INIT internal packet number for the controller. It provides an index into the device scan table (DEVTBL, see Section 18.2.3.1) for the controller. |
| DS.CTL | This byte contains the controller number of the device. |
| DS.CSR | This word contains the address (within the monitor) to load with the actual CSR base address of the first unit connected to the controller. |
| DS.DCT | This word contains the address (within the monitor) of the interrupt routine for the controller. |

### 18.2.4 Mini-FIP Tables

Mini-FIP uses tables which describe the current hardware disk configuration and the FIP device to be used for file operations. To control file-structured disk operations within INIT, mini-FIP uses control blocks similar to those used during normal timesharing.

#### 18.2.4.1 Disk Characteristics Tables -

Mini-FIP uses several tables to describe the various disks on the system. These tables are organized in one of two ways: by device type or by unit number. Tables organized by device type contain one entry for each possible disk controller and are accessed by device index number. Tables organized by unit number contain one subtable for each disk type. Each subtable is pointed to by the corresponding entry in a master address table arranged by device type. Individual subtables are indexed by unit number times two, up to the maximum unit number of the device (normally seven, but MSCP-class disks can range up to fifteen).

*Disk Unit Size (DSKSZL and DSKSZM)*

Two master tables are used to describe the size (in blocks) of each disk unit. For each unit, the low-order (least significant) word of the 2-word integer block size is contained in subtable Dx$SZL (where "x" is the disk type letter), pointed to by master table DSKSZL. The high-order (most significant) word of the block size is contained in subtable Dx$SZM, pointed to by master table DSKSZM. Master tables DSKSZL and DSKSZM are arranged by device type.

For those disk types with predetermined sizes, the subtable entries are filled in at INIT assembly time. Other entries are filled in by the autosizing code when the hardware configuration is determined and updated whenever a rescan of the hardware is executed. If the autosizing code determines that a particular disk unit does not exist,* the corresponding entries within the subtables are zeroed. If a particular controller cannot be located, both associated tables are zeroed.

*Device Cluster Size (DSKDCS)*

Master table DSKDCS (arranged by device type) points to a series of subtables used to store the cluster size of each disk unit. For each individual unit, the corresponding entry in subtable Dx$DCS (where "x" is the disk type letter), pointed to by master table DSKDCS, contains the device cluster size of the unit. For those disks with a predetermined cluster size, the subtable entries are filled in at assembly time. Other entries are filled in by the autosizing code when the hardware configuration is determined and updated whenever a rescan of the hardware is executed.

*Status Register Addresses (DSKCSR)*

Table DSKCSR (arranged by device type) is used to store the 1-word address of the major status register on the I/O page of each disk controller. Entries in this table are filled in by the INIT autosizing code when the hardware configuration is determined and updated whenever a rescan of the hardware is executed. If a controller does not exist, the corresponding entry contains a zero.

This table also contains entries for the supported magtape controllers. Since magtape index numbers are always negative, these additional entries are actually located immediately before the beginning of the table.

*Disk Names (DSKNAM)*

Each 1-word entry in table DSKNAM (arranged by disk type) contains the 2-character ASCII device name of the corresponding disk controller. The table is terminated by a word containing a zero.

### 18.2.4.2 Boot Memory -

This 5-word table contains critical information concerning the system disk — the disk that was bootstrapped to load INIT. It is created as a read-only table by the INIT autosizing code and can be examined by the option processing routines but cannot be changed. When mini-FIP is initialized, these values are copied to other FIP control structures for working use.

* Note that this determination is not possible with all disk controllers since some controllers cannot distinguish between a disk drive that is powered down and one that is nonexistent.

Figure 18-4 shows the format of the boot memory area. Each item is discussed below.

*Symbol* *Contents*

SYNXSV This word contains the disk index number of the system disk. If INIT was booted from magtape, this word is negative.

SYSUSV This word contains the unit number of the system disk.

RCSLSV If SYNXSV contains "IDX$DC" (that is, the system disk is an RC11/RS64 fixed head disk), this word contains the size of the disk (in blocks).

RFSLSV If SYNXSV contains "IDX$DF" (that is, the system disk is an RF11/RS11 fixed head disk), this word contains the size of the disk (in blocks).

SYSNAM This word contains the 2-character ASCII name of the system device — either disk or tape.

| | |
|---|---|
| System disk index number | SYNXSV |
| System disk unit number | SYSUSV |
| RC disk size | RCSLSV |
| RF disk size | RFSLSV |
| System device name | SYSNAM |

**Figure 18-4:** Boot Memory

### 18.2.4.3 Mini-FIP Control Area -

The mini-FIP control area is used to describe the disk currently being accessed — the "FIP device." When mini-FIP is initialized, it is set up to describe the system disk. Figure 18-5 shows the format of the mini-FIP control area. Each item is discussed below.

*Symbol* *Contents*

SYSNDX This word contains the disk index number of the FIP device.

SYSUNT This word contains the unit number of the FIP device.

DCS This word contains the device cluster size of the FIP device.

PCS This word contains the pack cluster size of the FIP device.

CLURAT This word contains the cluster ratio (that is, PCS/DCS) of the FIP device.

FIBPHS This 2-word area contains the low- and high-order disk address of the block of the FIP device currently in memory.

I,J,K These three words are used as a scratch area for packing and unpacking directory link words.

FIBSTA This byte contains a flag indicating whether the disk block in memory has been modified. The flag is set to force the FIP directory read routine to rewrite this block to disk before reading another block.

| | | | |
|---|---|---|---|
| | Disk index number | | SYSNDX |
| | Unit number | | SYSUNT |
| | Device cluster size | | DCS |
| | Pack cluster size | | PCS |
| | Cluster ratio | | CLURAT |
| | Disk address of block currently in memroy | | FIBPHS |
| | (scratch) | | I |
| | (scratch) | | J |
| | (scratch) | | K |
| FIBSTA | Memory status flag | (unused) | |
| | Unit number | Unit number * 2 | SYSPHU |
| DSIFLG | DSKINT flag | Retry count | NRETRY |
| IOERR | Error code | Error logging flag | NOERR |

**Figure 18-5:** Mini-FIP Control Area

SYSPHU This 2-byte area contains the FIP device unit number, times 2, in the first byte and the unit number itself in the second byte.

NRETRY This byte contains the negative number of retries to be made by the disk drivers before deciding that a disk block is bad. It is normally set to -8 or -9 (depending on the disk controller), but is set to -2 during the DSKINT option in order to be more sensitive to potential bad blocks on the disk.

DSIFLG This byte is used as a flag to indicate that the DSKINT option is running. When this byte contains a zero, DSKINT is running and all disk I/O errors are returned to DSKINT in the DSQ used for the I/O operation. When this byte is nonzero, all disk errors are considered fatal and control of execution transfers to the INIT trap handler (see Section 18.3.3) and ultimately back to the INIT main program segment.

NOERR This byte contains a flag used by the backup utility SAVRES when it is running under INIT. When set, it indicates that no errors are to be logged. Any disk errors are to be recorded in byte IOERR.

IOERR This byte contains the most recent disk or tape error code.

## 18.3 Processing Overview

INIT processing can be divided into six main functional areas:

- The secondary bootstrap, responsible for loading the INIT root segment and the memory-resident overlays into memory.
- The one-shot autosizing code that determines the hardware configuration of the booted system.
- The main program segment, containing the INIT operating system environment.
- The physical I/O routines, responsible for I/O to the console terminal, as well as to the disks and magnetic tapes.
- The logical I/O routines (mini-FIP) that allow INIT to access the on-disk structures.
- The individual options that permit tailoring of the RSTS/E environment.

The first five of these functional areas are discussed in the following subsections. Option processing is discussed in Section 18.4.

### 18.3.1 Secondary Bootstrap

INIT is started when a user "bootstraps" any RSTS/E bootable device. A bootable device contains a valid RSTS/E file structure with a 1-block module (known as a "secondary bootstrap") in its first logical block (LBN 0). The hardware bootstrap loader reads this block into CPU memory location 0 and begins program execution at that location. (See Section 8.6.1 for a complete discussion of the structure and function of a secondary bootstrap.)

The secondary bootstrap begins with a standard header containing the load address of the initialization code, the number of words to load, and the address to which control should be transferred when the load is complete. The header also contains information which can be retrieved later by INIT to determine the index number of the system disk (that is, the disk that was bootstrapped).

When the secondary bootstrap is started, it first determines the unit number of the booted device by examining the hardware controller registers. This value is saved within the bootstrap header for future use. The bootstrap then relocates itself to locations 157000 to 157776(8) — the highest memory location addressable without memory management. After this, it loads the root segment of INIT and the ODT overlay containing the debugging code.

Once its root is loaded, INIT constructs its operating environment as shown in Figure 18-1. First it relocates the ODT overlay into higher memory. It then uses the bootstrap code to load the INIDSK overlay (containing the disk and magtape processing code and physical drivers) and the INIMCP overlay (containing the MSCP-class disk processing code and drivers). It then relocates these overlays into higher memory. From this point on, these three pieces of code remain as memory-resident overlays, mapped when necessary using APRs 5 and 6.

After loading the memory-resident overlays, INIT reads INIONE into the overlay region. This overlay contains the one-shot autosizing code.

### 18.3.2 Autosizing Code*

After the root segment and memory resident overlays have been loaded into memory, the autosizing code must create the hardware configuration data structures, reflecting the current state of the machine.

If the bootstrap indicates that the booted device contains no valid HARDWR or INSTAL information (that is, the word at location GOXFER within the bootstrap header contains a JMP instruction instead of a MOV instruction, see Section 8.6.1), buffers INSBUF and DEFBUF are zeroed. (This is a precaution to prevent the invalid propagation of parameters as INIT is copied from system to system.) In addition, the hardware scan discussed below is executed for only disk and magtape controllers and for the system console terminal.

The autosizing code then scans the hardware. The system hardware configuration table (CSRTBL), containing the bus addresses of all RSTS/E standard hardware, is built from the read-only table CSRDEC. Additional CSRTBL entries are made for devices with "floating" bus address assignments. During this process, INIT attempts to access a register from each device's register set. If a device is not found, causing a trap to memory location 4, the entry in CSRTBL is zeroed to indicate that no such device is present. (Note, however, that no checking is done to determine whether any address that responds actually belongs to the proper device controller.)

Next, any nonstandard CSR bus addresses (as specified by the HARDWR option) are applied to CSRTBL. If a nonstandard address conflicts with the register set of some other device, the other device is disabled without warning. However, if the nonstandard address does not respond, the device is disabled and a warning message is printed.

At this point, unless a nonstandard vector location is indicated in the high byte of the device's entry in VECTBL, each device specified and enabled within CSRTBL is checked for vectoring. Using a device-specific routine, each device is made to interrupt and the location to which it vectors is determined. Vector locations are then entered in VECTBL. If the device cannot be made to interrupt, it is assumed to be malfunctioning. It is disabled and a warning message is printed. If the device vector location is already in use, either by some other device or as a system communication location, the device is disabled and a warning message is printed. If a device requires two vectors (for example, DH11 multiplexers), both vectors are checked for conflicts and the device is disabled if either vector is in use.

Once the hardware configuration is determined, the autosizing code retrieves the system device type and unit number from the bootstrap and sets up the FIP control area to select the system device.

Other machine hardware is also checked and its existence recorded in table HWTABL. The existence of at least one clock, EIS, and memory management is verified. A fatal error is printed if any of these items is missing. Each disk controller is checked to determine which drives are present and what types they are (unless this function is overridden by an entry in table UMIDST). Each RH-controlled tape drive is checked to determine to which formatter it is connected. Also, if any two tape drives have the same unit number, a warning message is printed.

Next, INIT determines the amount of memory on the system and checks memory integrity. This is done by writing into memory and processing any "nonexistent memory" errors,

* The term "autosizing" derives from the fact that this portion of the INIT code actually determines the size of existing memory, as well as the processor type and the number and configuration of the peripheral devices.

UNIBUS timeout errors, or memory parity errors. A nondestructive check is also done of the memory space in which INIT itself is located. Memory is scanned up to the maximum amount that can optionally be configured on the CPU: two M-words on 22-bit processors or on 18-bit processors with UNIBUS mapping registers; 124 K-words on 18-bit processors without UNIBUS mapping registers. If any errors are detected, the one K-word portion of memory containing the error is marked "nonexistent" and a corresponding error message is printed.

Finally, if the booted device was a disk, INIT locates three critical system files in account [0,1]. Two of these (SATT.SYS and BADB.SYS) are required for a minimal disk structure. The third file (INIT.SYS) contains the INIT code itself. Without these three files, INIT cannot properly manipulate the system disk or load its overlay code.

When all processing is complete, control transfers to the main INIT program segment.

Note that this process of scanning the hardware -- that is, the autosizing code — is executed at various stages throughout INIT's operation. In particular, a complete hardware scan (of all devices in DEVTBL) is done following the INSTAL and HARDWR options. Since the autosizing code is also known as the INIT "one-shot" code (implying that it is executed only once), this can lead to some confusion. What, in fact, happens is that control transfers to the BOOT module within the root, the system device is rebooted, INIT is restarted from the beginning, and the (one-shot) autosizing code is reexecuted.

### 18.3.3 Main Program Segment

INIT's main program segment is responsible for essentially three functions. First, it handles the user interface, requesting the name of the desired option and dispatching to the appropriate processing module.

Second, it processes program overlays. As mentioned earlier, INIT is overlaid using the standard RT11 overlay structure. When the overlay handler issues an EMT instruction to read in a new overlay segment,* the EMT is intercepted by a process within INIT known as the "overlayer." The overlayer first verifies that it has been called from the RT11 overlay handler and translates the request to a call to INIT's file-structured read handler. Along with this read request, the overlayer passes the file control block (FCB) for the file INIT.SYS[0,1]. Once the overlay has been loaded, control returns to the overlay handler which then transfers control to the loaded module.

The third area of the INIT's main program segment is the trap handler, used to report errors. When a module detects a nonrecoverable error, it issues a TRAP instruction, followed by a word containing the address of the error message text to be generated. The trap handler then resets the system, ensures the integrity of the vector region, prints the error text on the console terminal, and transfers control to the main program segment restart code. With the exception of the EMT instruction (mentioned above), Q-bus clock interrupts, and memory parity traps, all other traps and interrupts** are regarded as "unexpected" and cause INIT to reset the vector region and return to the "Option" prompt.

* All other EMTs are invalid.

** INIT uses programmed wait loops to determine when an I/O request to a device has completed.

### 18.3.4 Physical I/O Routines

The memory-resident overlay INIDSK contains a complete set of disk and magtape drivers. These are the same drivers used in general timesharing with the exception that they cannot process I/O requests on more than one unit simultaneously.

Disk I/O is done within INIT using the DSQ control block, just as it is within the RSTS/E monitor (see Section 10.2.2). The module requesting the transfer must set up a DSQ to contain the FIP block number of the disk block at which the transfer is to begin, the memory address of the transfer, and the function code of the function to be performed (either read or write). It then calls routine DISK. DISK determines which disk is to be used, inserts the disk index into the DSQ, and dispatches to the appropriate driver. If the driver detects an error, the operation is retried a number of times (depending on the driver). A nonrecoverable error causes an error trap and a diagnostic message to be printed. Disk errors are not normally returned to the calling module.

The only other general I/O routines within INIT are those that make up the terminal driver, located in the INIT root segment. These support only the console terminal, which must be connected to a DL11-type interface. The input routine is simple. When called, it reads a complete line from the terminal and returns it to the caller in a fixed buffer. It echoes the typed characters and handles RUBOUT characters. A line must consist of a single line feed or a string of characters ending with a carriage return. If a Control-C is typed, the operation is aborted and control transfers to the main program segment restart code.

The basic terminal output module prints a single character. It performs horizontal tab to space conversion and adds fill characters as appropriate for the terminal. This routine also watches the terminal input, allowing one character of type-ahead whenever output is in progress. It recognizes and processes the special synchronization characters, Control-S and Control-Q, as well as Control-O.

### 18.3.5 Logical I/O Routines (Mini-FIP)

The system initialization code uses a file processor which is similar to the RSTS/E general timesharing file processor.

The function of INIT's FIP (mini-FIP) is to allow INIT option routines to access the RSTS/E on-disk file structure. Unlike the timesharing file processor, mini-FIP does not provide overlay capabilities, "large file" processing, or file extension capabilities. Also, in the single-user INIT environment, there is no need to provide synchronization of file operations. Futhermore, mini-FIP provides no access to any device other than disk. (File-structured tape is accessed only by the COPY and SAVRES options. Magtape processing routines reside in the INIDSK memory-resident overlay.)

Mini-FIP is capable of performing file-structured operations on only one disk at a time. This disk is known as the "FIP device" and can be any unit of any valid disk type. When mini-FIP is initialized, the FIP device is set to the system device (that is, the device which was originally bootstrapped). When a routine wishes to operate on a disk other than the system device, it must set up the FIP control area to designate the new device as the FIP device.

FIP does not provide a multiple disk public structure. A disk must be explicitly specified by setting up the FIP control area before mini-FIP is called.

## 18.4 Option Processing

In addition to the general integrity checking done on the system hardware, INIT provides several features (known as "options") that can be used to create the proper RSTS/E disk environment, install the desired monitor and/or default run-time system, and set device characteristics and system default parameters. Additional options provide the facilities for patching the system software, loading the monitor, and beginning general timesharing.

Options are invoked through a user dialog controlled by INIT's main program segment. With the exception of the BOOT option (which resides in the root segment), all options reside in disk-resident overlays, with some overlays containing more than one option.

### 18.4.1 Initializing a RSTS/E Disk (DSKINT)

The DSKINT option initializes RSTS/E disks. All disks to be used with RSTS/E (except non-file-structured foreign volumes) must be initialized before they are used. DSKINT writes the minimal RSTS/E file structure on the disk and (optionally) creates the disk's library account [1,2]. In addition, DSKINT can format certain disks and do pattern checking to locate bad blocks.

The DSKINT option creates a minimal file structure for a RSTS/E disk consisting of a bootstrap block, the disk pack label, the master file directory (MFD), the group file directory (GFD) for group 0, and the system file account [0,1]. The user file directory (UFD) for account [0,1] contains entries for the storage allocation file (SATT.SYS) and the bad block file (BADB.SYS). When a system disk is initialized, the GFD for group 1 and the system library account [1,2] are also also included in the minimal file structure.

Note that only the file *structure* is created by the DSKINT option. No system files are created other than those necessary to maintain the minimal structure. Required files are added to the system account [0,1]by the COPY option, and the system library account [1,2] is built by the BUILD step of the system generation process.

In addition to creating the minimal file structure, the DSKINT option provides the mechanisms for formatting disks* and for doing pattern checking to locate bad blocks. Formatting involves writing the necessary timing and sense marks onto the disk and erasing any extraneous information. Pattern checking involves alternately writing and then reading a specified bit pattern to the disk to detect any blocks on which data cannot be reliably recorded. All clusters containing bad blocks are allocated to the bad block file (BADB.SYS) in account [0,1].

While non-file-structured swapping disks need not be initialized, it is usually of value to have the DSKINT option perform pattern checking on these disks to detect bad blocks. The file structure that is created in this process is useless and is overwritten during general timesharing operations.

DSKINT uses standard mini-FIP routines to build the disk and allocate blocks on it. Thus, it marks the FIP control area as being in use before beginning the user dialog. Then, as the user answers the dialog questions, supplying the required disk parameters, DSKINT fills in the control area and prototype blocks for the MFD/GFD structures and the [0,1] UFD, along with other control data.

* Formatting applies only to removable disk types; fixed head disks cannot be formatted.

A prototype storage allocation table is built (within the root segment), indicating that all pack clusters are available. If the disk has factory-written bad block information, that information is used to allocate the corresponding pack clusters to the bad block file. Thus, even if no pattern checking is done, bad blocks found by the factory will appear in the bad block file and will be unavailable for use.

If the disk is to be formatted, a flag is set to disable processing of any Control-C typed during the format phase. The appropriate disk formatter is then called.* If the disk goes off line for some reason or a nonrecoverable disk error is detected, an error message is printed and the user can choose to retry the procedure or to abort the entire DSKINT option. No bad block information is gathered during the formatting phase.

After formatting the disk, DSKINT does pattern checking to detect bad blocks. Three of the eight standard system patterns are done first (if specified), followed by up to three user-specified patterns, followed by the system pattern that zero-fills the disk. The procedure for each pattern is as follows:

- The disk controller is reset.
- The pattern is loaded into a 1-block buffer and an informational message naming the pattern is printed.
- The pattern is written to block 0 of the disk, and a write-check operation is done to verify its accuracy. If an error is detected, a fatal error message is printed and DSKINT terminates processing since block 0 is required for the bootstrap.
- The pattern is written and then write-checked to the remainder of the disk in larger, 16-block transfers. During this procedure, only two error retries are permitted. If more than two retries are required for a correct transfer, it is assumed that the disk block will soon be completely bad.
- If an error is detected during a multiblock transfer, each block within the 16-block area is written and checked individually to isolate the precise location of the error.

  If the pack cluster containing the error was detected as bad on a previous pattern (or from the factory error records), no message is printed. If the cluster was not previously detected as bad, a message is printed identifying the bad block and pack cluster, and the cluster is marked as bad by setting the appropriate bit in the prototype storage allocation table.

  After each error is detected, the disk controller is reset and 1-block testing continues. When all sixteen blocks of the multiblock region have been tested, multiblock transfers are resumed. If there was an error on the 16-block transfer, but none was detected on individual block tests, an informative message is printed and testing continues.

  If an error is detected in block 1, a fatal error message is printed and DSKINT terminates processing since block 1 is required for the pack label.

When all patterns have been exercised, the storage allocation table has bits set for each bad pack cluster discovered. If the number of bad clusters exceeds a fixed maximum, a fatal message is printed and DSKINT terminates processing.

After pattern checking, DSKINT builds the minimal file structure. The procedure for this is as follows:

* The standard disk drivers are not used during formatting.

- Prototypes for the MFD, the group 0 GFD, and the [0,1] system account UFD are set up in memory.
- The bad block file (BADB.SYS) is generated by scanning the storage allocation table for nonzero bits and entering the appropriate pack cluster numbers in the file. Space for BADB.SYS is allocated on the disk and the prototype [0,1] UFD is filled in with the corresponding retrieval pointers.
- The storage allocation table (SATT.SYS) itself is allocated at the user's specified location on the disk, and the prototype [0,1] UFD is filled in with the retrieval pointers for each pack cluster allocated to SATT.SYS.
- Clusters are allocated for the MFD, the group 0 GFD, and the [0,1] UFD. The device cluster number of the GFD is recorded in the prototype MFD, and the [0,1] retrieval pointer is recorded in the GFD. Cluster maps are written to all MFD and GFD blocks (except blocks 1 and 2, as dictated by the directory structure).
- At this point, all allocation on disk is done and all DCNs and retrieval pointers have been filled in. SATT.SYS and all directory structures are written to their allocated disk clusters and write-checked.
- A dummy bootstrap routine is then written to block 0 of the disk and the pack label is written to DCN 1.

At this point, DSKINT processing is complete.

### 18.4.2 Copying System Files to Disk (COPY)

The COPY option is used to build a minimal RSTS/E system on a previously initialized system disk. It is, in effect, a limited file transfer facility allowing tape-to-disk or disk-to-disk image copying of a standard set of files from account [0,1] on the booted device. This set consists of the following files:

- The system initialization code (INIT.SYS).
- The first file in account [0,1] with a file type of "SIL" — the RSTS/E monitor save-image library (SYSGEN.SIL on the distribution medium).
- The first file (or, optionally, *all* files) in account [0,1]with a file type of "RTS" — the primary run-time system save-image library (RT11.RTS on the distribution medium).
- The first file in account [0,1] with a file type of "ERR" — the system error message text file (ERR.ERR on the distribution medium).
- All files in account [0,1]] with a file type of "SAV" — any stand-alone programs that can be run outside the normal timesharing environment.*

The output files are created as contiguous files in account [0,1] on the output disk. The output disk can contain other files but must have enough free contiguous space to permit creation of the four system files.

After performing the user dialog, COPY sets up the output disk as the FIP device and checks the file structure. The disk must contain a valid RSTS/E file structure, including accounts [0,1] and [1,2], and having both a bad block file (BADB.SYS) and a storage allocation table (SATT.SYS) in account [0,1]. If the disk does not contain a valid file structure, a fatal error message is printed and COPY terminates execution.

* DIGITAL no longer supplies any stand-alone programs.

Once the file structure is determined to be valid, COPY performs the following procedure for each file to be transferred:

- If the system device is a magtape, the tape is rewound.
- The [0,1] directory of the system device is searched for the first file with the appropriate file type. The length of that file is determined, in blocks. This is done as follows:
- All files with types "SIL" and "RTS", as well as file INIT.SYS, are built as save-image libraries and have a SIL header (known as a SIL "index block"). The total size of the file (in blocks) is contained in the SIL header.
- All files with type "ERR" are assumed to be 16(10) blocks long.
- All files with type "SAV" have their length (in bytes) stored in the first block of the file. This length is rounded up to a multiple of 512(10) bytes and then converted to blocks.

  COPY does an additional consistency check on all SIL headers by computing the checksum of the header and comparing it with a value stored within the header itself.
- The output disk is selected as the FIP device and the [0,1] directory is searched for any file with the same name as the input file. If one is found, a message is printed asking the user if the conflicting file is to be superseded or to be saved. If the file is to be superseded, it is deleted and the directory search is repeated (in the event that there are somehow two files of the same name in [0,1] of the output disk). If the file on the output disk is to be saved, the remainder of the COPY procedure is skipped for that particular file.
- The storage allocation table for the output disk is read into memory. A contiguous file is created in [0,1] with the same length as the input file. The file cluster size is set equal to the output disk pack cluster size and the file is marked "protected." If there is no room for the output file, a fatal error message is printed and COPY terminates execution.
- Mini-FIP is used to lookup the file just created. This creates a valid file control block (FCB) for the output file.
- The input file is copied to the output file, in 1-block transfers. Since the output file is contiguous and requires no window turns, no FIP operations are necessary for the output disk.

If the file transferred was INIT.SYS, a device-specific bootstrap block is created for the output disk, hooking the file to the bootstrap. This block is written to block 0 of the output disk. In addition, the blocks that correspond to the installed SIL and SIL defaults (buffers INSBUF and DEFBUF, see Section 18.2.2) are overwritten with zeros in the output file. Thus, the copy of INIT.SYS on the output disk will have no installed SIL.

Once this procedure has completed for all files, the output disk is bootstrapped and INIT is restarted from the output disk.

### 18.4.3 Patching the System Software (PATCH)

The PATCH option provides a means of altering the RSTS/E system code as errors are detected and corrections are published. It can be used to make permanent changes to any file in account [0,1] on the system disk. The files that can be patched include the initialization code itself (INIT.SYS), any save-image library (including the RSTS/E monitor and runtime systems), and any other file in [0,1].

PATCH makes permanent changes on the system disk but has no effect on any code currently in memory. Thus, if corrections are to be made to the initialization code, the system disk must be rebooted and INIT restarted before the corrections take effect.

The PATCH user dialog requests the name of the file to be patched, along with the internal module name (if the file is a save-image library). The base address within the file or module and an offset from the base of the first location to patch are also requested. The file name is looked up on the system disk using a general-purpose file request queue block (FIRQB). If the file is a SIL, block 0 (the SIL index) is read into memory and verified. Address limits are calculated from the index and stored in memory. The base and offset addresses are converted to block number and offset within the file, biased (if necessary) by the module start within the SIL.

PATCH then prints a column header and the current contents of the specified location within the file or module. The line ends with a question mark, prompting the user to enter the new contents of that location, if desired. The user can step forward or backward through the locations of the file, examining ("opening") and altering the contents as desired.

For each location opened, the corresponding block of the file is read from disk (if it is not already in memory) and the contents of the location are printed. If the user alters the location's contents, the new contents are recorded in memory and the block is immediately rewritten to the disk.

When the user enters a Control-Z, PATCH leaves the file open and returns to the user dialog, requesting a new module name and/or base and offset address. Thus, the user can continue patching the same file at a different location.

When the user enters a Control-C, the terminal service routines return control to the main INIT program segment, terminating PATCH execution.

### 18.4.4 Installing a Save-Image Library (INSTAL)

The INSTAL option is used to designate the save-image library on the system disk that is to be used as the RSTS/E timesharing monitor. This permits multiple monitor SILs to be stored on the system disk. The SIL installed with this option becomes the RSTS/E system executive file for all subsequent operations until superseded by another installed SIL. A monitor SIL must be installed before using the DEFALT and START options.

The INSTAL user dialog requests the name of the desired monitor SIL. The file specified must exist on the system disk in account [0,1]. The file is then opened using a general-purpose file request queue block (FIRQB).

The index block of the SIL is read into memory and scanned for required modules: RSTS, EMT, FIP, OVR and DEFALT. Each module within the SIL is checked for valid length, load address, position within the SIL, transfer address and symbol table. If any required modules are missing or any modules are not correctly configured, a fatal error message is printed and INSTAL terminates execution.

If a monitor SIL is currently installed, it is removed. This is done by clearing the "protected" bit in the directory entry of the SIL, the default run-time system file, and the system error message file.

The directory entry of the new SIL is marked "protected" and the SIL name is recorded in buffer INSBUF. This information is also written to disk (within file INIT.SYS) so that the newly installed SIL will be remembered the next time the system disk is booted and INIT is started.

The DEFALT phase of the new SIL is read into buffer DEFBUF. If the primary run-time system and system error message files have already been specified for the SIL, they are opened and their directory entries are marked "protected." If either file cannot be found in account [0,1], a message is printed and the name of the file is zeroed within DEFBUF. This change is immediately written back to the disk, preventing the system from starting until a valid set of defaults is specified. (Note that defaults previously specified for the SIL may now be inconsistent with the current hardware or with the current file structure. Any such errors are not detected by INSTAL but will be detected by the START option.)

During system startup, several memory locations within the installed SIL are modified according to the SIL's device support and default parameter settings. To minimize disk I/O during system startup, INSTAL preprocesses these locations by extracting them from the SIL's symbol table and storing them in a buffer in the INIT root. INSTAL also creates the device status table (DSTABL, see Section 18.2.3.10) containing information regarding each device controller configured into the installed SIL. DSTABL is created from device-related symbols found in the SIL symbol table and is stored in a buffer in overlay DSTBLO (within file INIT.SYS). Later, during system startup, this buffer is used to modify device-related locations within the RSTS/E monitor (see Section 18.4.9).

### 18.4.5 Changing System Files (REFRSH)

The REFRSH option operates on files in the system account ([0,1]) and is used to list file status, change system file allocations, create and delete files, and examine and update the bad block file. REFRSH also includes a facility for "cleaning" a disk — rebuilding the storage allocation table and checking the consistency of directories. REFRSH operates only on disks that have a valid RSTS/E file structure.

The REFRSH user dialog begins by requesting the name and unit number of the disk to be refreshed. If the on-disk directory structure indicates that the disk pack was not logically dismounted the last time the pack was used, REFRSH automatically cleans the disk before continuing. If the directory structure indicates that the disk *was* logically dismounted, REFRSH asks the user if the disk should be cleaned. The cleaning operation consists of the following steps:

- A new storage allocation table (SATT.SYS) is built reflecting all files on the disk.
- All directories are scanned and all files marked for deletion or with a file type of "TMP" are deleted.
- Doubly-allocated blocks are located and the user is asked to specify their proper allocation.
- Invalid directories are located and deleted.
- Files containing bad blocks are located and the user is given the option of deleting them.

- Any disk blocks deallocated during the cleaning process are overwritten with zeros.

Once the disk has been cleaned (if necessary or requested), REFRSH enters its suboption loop. There are four valid REFRSH suboptions:

| | |
|---|---|
| LIST | List the file status table. |
| CHANGE | Change the system file allocation. |
| FILE | Change the characteristics of a system file. |
| BADS | Examine and/or change the bad block file. |

### *The LIST Suboption*

The LIST suboption prints a table showing the current status of all files in account [0,1]. This status consists of such things as the file name, current size, minimum size, starting logical block number, and flags indicating whether the file is contiguous, protected, and/or required for starting the monitor.

### *The CHANGE Suboption*

The CHANGE suboption is used to change the system file allocation by creating, deleting, or relocating files in account [0,1].

Normal timesharing operation requires certain files on the system disk in account [0,1]]. Two of these files (SATT.SYS and BADB.SYS) are created when the disk is initialized (by DSKINT). Four more, required for a minimal system, are created by the COPY option or during system generation. The remaining required files — most notably the swapping files — can be created by REFRSH with the CHANGE suboption.

CHANGE prompts the user with the name of each of the standard RSTS/E system support files and then gives the user the option of creating or changing the file's allocation. The user can also create or delete other nonstandard files in account [0,1]. All files created or relocated are contiguous files.

To locate a file on the disk, the user specifies a base logical block number (LBN). This block number is considered a preferred location — not an absolute requirement. The specified block number is used as the place to begin searching for contiguous space.

Note that while these files can be created during normal timesharing, CHANGE allows the user to control the location of the files, thereby increasing system speed and efficiency. Also, since CHANGE performs simultaneous allocations, usually while the disk is less full, chances are increased that the required contiguous space will be found.

### *The FILE Suboption*

The FILE suboption is used to create and delete files in account [0,1] and to set or clear the "protected" bit in a file's directory entry.

FILE prompts the user for the file name. The only files that cannot be manipulated are SATT.SYS, BADB.SYS, and INIT.SYS. If the file exists, FILE asks if it should be deleted. A file can be deleted at this point regardless of the status of the "protected" directory bit.

If the file does not exist, FILE asks the user for the size and desired location of the new file. The file is then positioned and created as a contiguous file, using the same procedure as in the CHANGE suboption.

Finally, FILE asks the user to specify whether the "protected" bit should be set in the file's directory entry. If this bit is set, the file cannot be deleted or renamed during normal timesharing.

*The BADS Suboption*

The BADS suboption is used to examine and/or add to the bad block file (BADB.SYS).

If the user asks to examine the bad block file, BADS lists the starting logical block number of each pack cluster containing one or more bad blocks. If the user asks to add to the bad block file, BADS prompts the user for the logical block number of the bad block. The pack cluster containing that block is calculated and added to BADB.SYS, unless it is already allocated to the file.

Blocks allocated to BADB.SYS cannot be deallocated from the file.

### 18.4.6 Specifying the Hardware Configuration (HARDWR)

The HARDWR option is used to list the hardware configuration available on the processor being used, to set nonstandard bus addresses for I/O control and status registers, to set nonstandard vector addresses, and to disable and reenable devices. A monitor save-image library need not be installed when the HARDWR option is run because the settings affected are recorded in buffer DEFBUF and are rewritten to disk within INIT.SYS, not the monitor SIL. Thus, when a new monitor is installed, the hardware configuration (as reflected on the system disk) remains valid as long as no devices have been added or removed.

HARDWR is structured as a collection of suboptions, some of which are device-specific. The general suboptions available are as follows:

| | |
|---|---|
| LIST | List the current hardware configuration. |
| DISABLE | Disable a device controller. |
| ENABLE | Enable a device controller. |
| CSR | Declare a nonstandard controller address. |
| VECTOR | Declare a nonstandard vector assignment. |
| UNITS | Associate a disk type with a particular controller. |
| RESET | Reset all vectors and CSR addresses, enable all devices, and restart INIT. |
| EXIT | Exit from the HARDWR option. |

*The LIST Suboption*

The LIST suboption prints a table showing all controllers found by the autosizing code, along with their CSR bus addresses and vector locations. It also lists all disk media and tape formatter types, as well as all miscellaneous hardware options present, such as the clock, floating point instruction set, I&D space memory management option, and the type of bus.

### *The ENABLE/DISABLE Suboptions*

ENABLE and DISABLE are complementary suboptions used to specify whether a device controller should be enabled or disabled at the start of timesharing. A controller will be enabled at system startup if the low-order bit (bit 0) of its CSRTBL entry is zero; it will be disabled if bit 0 is set to one. Thus, the ENABLE and DISABLE suboptions merely clear and set this bit.

Note that these suboptions are different from the ENABLE and DISABLE suboptions of the SET option in that here the entire controller is enabled or disabled. SET, on the other hand, permits the disabling of a single unit without affecting other units on the same controller (see Section 18.4.8).

### *The CSR Suboption*

The CSR suboption is used to specify a nonstandard control and status register (CSR) bus address for a particular device controller. Addresses specified with this suboption are recorded in table SETTBL (see Section 18.2.3.4) and used to build table CSRTBL (see Section 18.2.3.5) when the hardware is rescanned.

### *The VECTOR Suboption*

The VECTOR suboption is used to specify the address of a nonstandard interrupt vector for a particular device controller. Addresses specified with this suboption are recorded in the high-order byte of the controller's VECTBL entry (see Section 18.2.3.6). When the hardware is rescanned, the autosizing code accepts this address as the vector location and does not attempt to make the controller interrupt (see Section 18.3.2).

RSTS/E does not require a device to interrupt at any standard vector location. A device can be supported under RSTS/E as long as it interrupts within a predefined processor priority range and vectors to a valid memory location (between 100 and 376(8)).

### *The UNITS Suboption*

The UNITS suboption is used to associate a disk media type with a particular unit of a controller. Many disk controllers can run a number of different types of disk drives. If the drive is powered up at the time that the autosizing code executes, INIT can determine which type of drive is present. However, if the drive is powered down or disconnected from the controller, there is generally no way for INIT to make this determination. Under this circumstance, INIT will normally disable the unit.

Using the UNITS suboption, however, the user can specify the media type for a particular unit, thereby permitting INIT to calculate the parameters necessary during timesharing (such as device cluster size). Thus, even if the unit is powered down or disconnected during the autosizing process, INIT can leave the unit enabled.

Media types specified with the UNITS suboption are stored in the unit media overrides table (UMIDST, see Section 18.2.3.8).

*The RESET Suboption*

The RESET suboption is used to purge all previous HARDWR settings. SETTBL is zeroed, clearing all nonstandard bus addresses; bit 0 is cleared for all CSRTBL entries, enabling all devices; and the high byte of all VECTBL entries is zeroed, clearing all nonstandard interrupt vectors. HARDWR processing then exits.

### 18.4.7 Setting System Default Parameters (DEFALT)

The DEFALT option is used to set or change default startup conditions for the currently installed monitor SIL. Since these definitions are set for a particular monitor rather than for the system device or environment, the option must be used whenever a newly generated monitor SIL is installed. However, any defaults specified are recorded in the DEFALT phase of the monitor SIL, so a SIL that has been removed and then reinstalled does not require the DEFALT option to be run a second time.

Some of the default parameters can also be changed at system startup time when the START option is used. However, any changes made at startup are temporary. They apply only to the current timesharing session (and any autorestart after a system crash). After the timesharing session shuts down, permanent defaults are again in effect.

DEFALT requires the system disk to contain a minimal RSTS/E system, including those files created by the COPY option. A monitor SIL must also be currently installed. If any required files are missing or a SIL is not installed, a fatal error message is printed and DEFALT terminates execution.

DEFALT steps through the various parameters that can be set, asking for initial settings and/or changes. It first prompts the user for the maximum number of jobs permitted during timesharing (system parameter LMTCNT*) and the maximum storage space a job can occupy in memory and in the swap files (system parameter SWPMAX). It also prints the current settings of these two parameters. If either value is zero (such as when no defaults have been set for the current monitor), the absolute maximum number of jobs configured during system generation (system parameter JOBMAX) is assumed for LMTCNT and SWPMAX defaults to 28 K-words.

Since RSTS/E requires that there be enough room allocated to the swap files to contain all possible running jobs, certain combinations of LMTCNT and SWPMAX may be inconsistent with the size of the existing swap files. In this case, DEFALT prints a warning message, informing the user of the problem. The user will not be allowed to start the system until the values agree; he must either change the maximum values or increase the size of the swapping file.

LMTCNT and SWPMAX can be changed at system startup time with the START option. The processing involved is identical to that performed by DEFALT.

The DEFALT dialog then prompts the user to specify the name of the run-time system to be used as the system default. If there is currently a run-time system installed, its name is printed. The file specified by the user must be contiguous and must exist on the system disk in account [0,1] with a file type of "RTS". It must be a valid save-image library.

* Note that this is different from the maximum number of jobs for which the monitor was configured during system generation.

DEFALT opens the named file and reads and checks the SIL index block. The directory of account [0,1] is scanned to ensure that the file is contiguous. If the file is a valid run-time system image, its name is saved in buffer DEFBUF. The size of the run-time system is computed and saved, and its default memory allocation is cleared, forcing it to be reallocated later during the memory allocation portion of DEFALT processing. The run-time system file is not marked as being in use until the DEFALT option completes processing.

After this, the user is prompted to specify the name of the file that contains the RSTS/E error message text. If an error message file is installed, its name is printed. The file specified must exist on the system disk in account [0,1] with a file type of "ERR". It must be exactly 8 or 16(10) blocks long.

DEFALT opens the named file and checks its length. The file name is saved in DEFBUF.

Next, the DEFALT dialog prompts the user for the installation name. If a name is currently specified, it is printed. The name must be a string of 1 to 15 printable characters. Once set, the name is used for INIT's identifying message and becomes part of the installed error message file when the system is started.

At this point, DEFALT begins the memory allocation portion of its processing. Memory for the run-time system is allocated (if necessary) and the user is asked to specify any memory that should be locked out or assigned to the extended buffer pool. The user can also increase the number of small buffers allocated to the FIP buffer pool.

If defaults have never been set for this monitor SIL, DEFALT automatically sets up the memory allocation table. If defaults have been set, the existing memory allocation table is checked against the actual memory layout of the central processor. If any existing portions of memory are flagged as nonexistent or if any nonexistent portions of memory are flagged as in use, an informative message is printed and DEFALT resets the entire table.

If the table as a whole is valid but a new primary run-time system has been installed, DEFALT scans the memory allocation table for a region in which to put the run-time system. If the current memory allocation contains no contiguous region large enough for the new run-time system, an informative message is printed and DEFALT resets the entire table.

The memory allocation table is then printed and the user is asked if there are any changes to be made in the allocation. If the user indicates that there are changes, DEFALT enters a suboption loop, offering the following suboptions:

| | |
|---|---|
| LIST | List the current memory allocation table. |
| PARITY | List the parity memory configuration. |
| RESET | Reset the default memory allocation. |
| LOCK | Lock out a portion of memory. |
| UNLOCK | Unlock a portion of memory. |
| RTS | Move the primary run-time system to a specific location. |
| XBUF | Allocate space for the extended buffer pool. |
| EXIT | Exit from the suboption. |

### The LIST Suboption

The LIST suboption lists the memory allocation table, showing the use of each region of memory. Each line of the listing contains the address range of the region, the starting location expressed in K-words, the length of the region in K-words, and a mnemonic indicating the use of the region. The mnemonic can be one of the following:

| *Mnemonic* | *Meaning* |
|---|---|
| USER | This region of memory is unassigned and can be used during normal timesharing for user jobs and resident libraries or alternate run-time systems. All existing memory not otherwise in use is USER memory. |
| EXEC | This region of memory is occupied by the RSTS/E monitor. It is always first, since the monitor must occupy the first physical memory on the machine. The region also contains all dynamic regions created for FIP (FIPOOL, OVRBUF, FIP resident oiverlays, and so on), instruction-space code (on I&D processors), and data structures required by MSCP-class device drivers. There is only one EXEC region of memory. It is required. |
| RTS | This region of memory is occupied by the primary run-time system. This region can be anywhere in existent memory as long as it is completely within the first 124 K-words of memory. There is only one RTS region of memory. It is required. |
| XBUF | This region of memory is in use by the extended buffer pool. There can be only one XBUF region of memory but there need not be any. |
| LOCK | This region of memory is not available to RSTS/E and will not be used during timesharing operations. There can be any number of locked regions. |
| NXM | This region of memory does not exist. There is normally one NXM region, beginning where physical memory ends and continuing to the highest legal address (that is, the region of addressing space beyond the actual memory of the machine). However, INIT performs checking on every bank of memory on the machine. If one is missing or malfunctioning, its region is marked as nonexistent. Thus, the appearance of an NXM region other than the expected one may indicate a hardware malfunction. |
| ODT | This region of memory contains the monitor ODT. The ODT region is set by the ODT suboption. Use of the ODT utility is restricted to debugging during RSTS/E development so the ODT suboption is hidden from the user. |

### The PARITY Suboption

The PARITY suboption is a diagnostic tool that prints the various types of parity on the system. Like the memory allocation table, each line of the PARITY listing contains an address range, starting location expressed in K-words, a length in K-words, and a code indicating the type of parity memory for the defined region. The code can be in one of the following forms:

| *Code* | *Meaning* |
|---|---|
| NO | The region consists of nonparity memory. |
| NXM | The region consists of nonexistent memory. |
| nn/mm | The region consists of interleaved core memory. |
| nn | The region consists of parity memory with address information. |
| nn(NA) | The region consists of parity memory with no address information. |
| nn(ECC) | The region consists of memory with error correcting code. |

The values "nn" and "mm" are the last two digits of the address of the parity register controlling that region of memory. Up to sixteen parity registers can be placed in the address range 772100 to 772136(8). When a parity error is detected, the parity register responsible for the region of memory containing the error usually contains information on the location of the error. The symbol "NA" indicates that the parity register contains no address information.

When core memory is interleaved, two parity registers are used to control a region. The first is responsible for even-word addresses and the second is responsible for odd-word addresses.

### *The RESET Suboption*

The RESET suboption is used to set up the memory allocation table according to the following rules:

- The primary run-time system is located in low memory immediately following the monitor.
- No memory is reserved for the extended buffer pool. If any memory was previously allocated to XBUF, it is released.
- No memory is locked. All memory not in use by the monitor or the primary run-time system is freed for user space.
- Any new memory on the system is found and is added to user space.

The first time the DEFALT option is invoked for a monitor save-image library, an implicit RESET is done. In general, RESET is only used when additional physical memory is added to the system.

If the monitor and/or the primary run-time system is too large for the existing memory, a fatal error message is printed and DEFALT terminates execution. The only way to recover from this is to install a smaller monitor or specify a smaller primary run-time system.

### The LOCK/UNLOCK Suboptions

The LOCK suboption is used to lock out certain portions of memory, thus preventing their use during timesharing. This feature is used when a certain section of memory is defective to prevent possible disruption of system operations.

The region to lock must currently be allocated to user space. The regions allocated to the monitor, the primary run-time system, or the extended buffer pool cannot be locked, nor can nonexistent memory.

The UNLOCK suboption is used to free a previously locked region of memory. The unlocked region becomes user space.

### The RTS Suboption

The RTS suboption is used to position the primary run-time system anywhere within the first 124 K-words of memory. The region specified by the user must be available as user space or currently in use by the run-time system. It must be entirely contained within the first 124 K-words of memory. Once the new RTS region is defined, the existing RTS region (or what remains of it) is released to user space.

### The XBUF Suboption

The XBUF suboption is used to reserve a region of memory for use as the extended buffer pool. This buffer pool is used for caching disk data, buffering output to the line printer, local and DECnet messages, and DMA device data transfers. The region specified by the user must be available as user space or currently allocated as an XBUF region. If an XBUF region is currently defined, it (or what is left of it) is released to user space when the new region is allocated. Only one XBUF region can be in use at any one time.

### The ODT Suboption

The ODT suboption is used to locate the monitor debugging tool in memory. This option is hidden from the user, since the use of ODT by an inexperienced user can cause the system to crash. It is intended as a RSTS/E development tool. ODT requires a 4 K-word memory region, which must be within the first 124 K-words of memory.

When all changes have been made to the memory allocation table and the user is satisfied with the allocation, the EXIT suboption is used to terminate the memory allocation portion of DEFALT processing.

Memory allocation can be changed at system startup with the START option. The processing involved is identical to that performed by DEFALT.

At this point, if the monitor SIL was generated to included extended data caching and the XBUF region of memory is more than two K-words long, the DEFALT user dialog prompts the user for the cache cluster size. The cache cluster size is the number of disk blocks (1, 2, 4, or 8) that the monitor will treat as a single unit for disk I/O. During timesharing, all disk read and write operations will take place in transfers of this size.

After this, DEFALT begins crash dump processing. First, DEFALT informs the user whether crash dumps are currently enabled or disabled. It then checks whether file CRASH.SYS is present as a contiguous file on the system disk in account [0,1]. It also checks that the file is large enough to hold the current monitor. If the file does not meet all these criteria, a warning message is printed. If crash dumps are currently enabled, they are not explicitly disabled, but if the user wants the feature available at system startup, he must create a CRASH.SYS file of the required size (with the REFRSH option) before he uses the START option.

The user is then asked whether crash dumps should be enabled or disabled. If the user indicates that they should be enabled, but a proper CRASH.SYS file does not exist, no further warning is given.

This same processing is performed at system startup through the START option. However, if a valid CRASH.SYS file of the proper length does not exist and crash dumps are currently enabled, the feature is disabled and a warning message is printed informing the user that his default cannot be honored. The user is given no opportunity to reenable the feature.

When DEFALT processing is complete, all settings specified have been recorded in buffer DEFUBF. This buffer is then written back to disk as the DEFALT phase of the monitor save-image library.

### 18.4.8 Changing Device Characteristics (SET)

The SET option is used to change the characteristics of devices in the installed monitor SIL. It can disable and enable devices on a unit-by-unit basis, set or reset modem control on keyboards, and change line printer characteristics. Any changes made are written back to disk within the DEFALT phase of the monitor SIL.

The SET option is structured as a collection of suboptions. The suboptions available are as follows:

| | |
|---|---|
| LIST | List the status of a device. |
| MODEM | Enable modem control for specified keyboards. |
| LOCAL | Disable modem control for specified keyboards. |
| LP | Set line printer characteristics. |
| DISABLE | Disable a device unit. |
| ENABLE | Enable a device unit. |
| PRIV | Make device ownership privileged. |
| UNPRIV | Make device ownership nonprivileged. |
| EXIT | Exit from the SET suboption. |

#### *The LIST Suboption*

The LIST suboption is used to list the status of one or more devices supported by the installed monitor. Each line of the listing contains the device name and unit number used during timesharing, the hardware controller type and number, and a comment giving additional information about the device. The following comments can appear:

| *Comment* | *Meaning* |
|---|---|
| DISABLED | The device has been disabled and will not be available for timesharing. |
| MODEM | Modem control is enabled for the device (keyboards only). |
| Lxxx | The device is a line printer of the specified type (LP, LV, LS, LA180, etc). The width of the printer is also specified, along with the notation "LOWER CASE" if the printer supports lowercase. |

#### *The MODEM/LOCAL Suboptions*

MODEM and LOCAL are complementary suboptions, used to enable and disable modem control on keyboards that are capable of such control.

*The LP Suboption*

The LP suboption is used to change the characteristics of any line printer on the system. The user can specify characteristics either by entering the line printer type (if it is one of the standard types supported), or by entering the individual characteristics of the printer. These individual characteristics include such things as whether or not the printer recognizes control characters and vertical tabs, and how line endings should be dealt with. This suboption also permits the user to change the printer width and indicate whether or not the printer supports the lower case character set.

*The ENABLE/DISABLE Suboptions*

ENABLE and DISABLE are complementary suboptions used to make device units available or unavailable for use during timesharing. A timesharing user who attempts to access a unit that has been disabled with the DISABLE suboption of SET will receive an error from the monitor. Note that this suboption differs from the DISABLE suboption of HARDWR (see Section 18.4.6) in that SET permits disabling a single unit on a controller without affecting other units of the same type.

The DISABLE suboption of SET can be used when a particular unit is unavailable due to hardware problems. It can also be used when a device has been configured into the monitor but has not yet been installed. Although the system will automatically disable all units associated with a controller which is not present, disabling those units with SET suppresses the warning messages associated with automatic disabling.

*A device is reenabled with the ENABLE suboption.*

*The PRIV/UNPRIV Suboptions*

PRIV and UNPRIV are complementary suboptions used to indicate the privilege level necessary for ownership of a particular device. The PRIV suboption is used to specify a device that can only be owned (with the ASSIGN or OPEN commands) by a privileged user. UNPRIV is used to specify a device that can be owned by any user. If neither suboption is applied to a device, any user can own the device.

Disks cannot be specified under the PRIV suboption.

### 18.4.9 Starting Timesharing (START)

The START option is used to load the RSTS/E system and bring it up for normal timesharing.

The START dialog asks the user if he wishes to change certain default startup conditions. The job and swap maximum values (LMTCNT and SWPMAX) can be set, the memory allocations can be changed, and crash dumps can be enabled or disabled. If the user changes any of these defaults, processing is identical to that performed by the DEFAULT option (see Section 18.4.7). Note, however, that any changes to these parameters are temporary. They apply only to the current timesharing session (and any auto restart after a system crash). After the timesharing session shuts down, the permanent system defaults are again in effect.

Before timesharing can be started, various locations within the monitor must be initialized according to the system's default parameter settings and the device-specific values that reflect the current configuration. To accomplish this, START builds what is known as the "jam list" — a table of addresses and values to be used to modify memory once the monitor has been loaded.*

Entries within the jam list can take two forms. The first form consists of a word count (N) and a destination address (ADDR), followed by a list of N values to be loaded into memory, starting with the specified address (see Figure 18-6a). The second form of jam list entry consists of a negative pair count (-N), followed by N word pairs, each consisting of a value and a destination address (see Figure 18-6b).

START creates jam list entries for the following types of system parameter:

- Job-related tables
- The memory control list (MEMLST)
- Tables related to system files (SWAP, CRASH, OVR, ERR, and so on).
- The buffer pools — XBUF and both small buffer pools.
- Device-related tables and CSR/vector assignments.

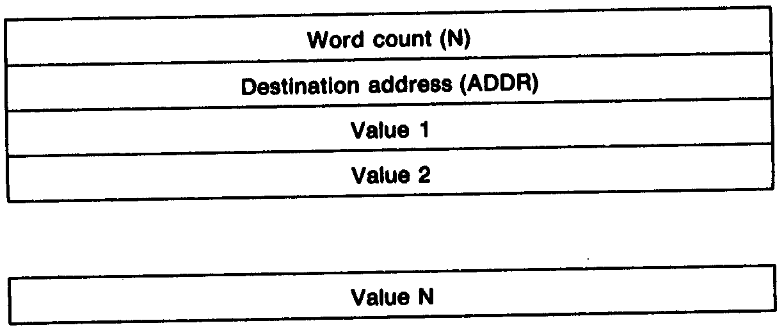


**Figure 18-6a:** Multiword Jam List Entry

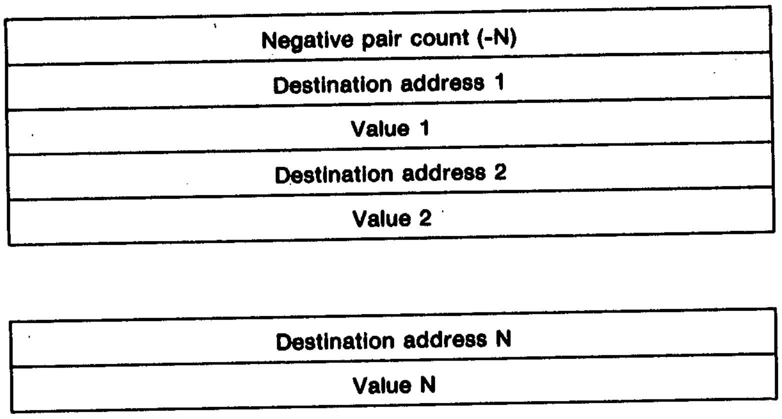


**Figure 18-6b:** Paired Jam List Entry

**Figure 18-6:** Jam List Entries

* These monitor locations are modified in memory, rather than on disk, to provide the flexibility of moving a monitor SIL between hardware configurations without major changes.

Once the jam list is created, START allocates an eight K-word area of free memory to hold the monitor loading code. Since the loader reads the monitor into memory, it cannot be located in any section allocated to the monitor (the EXEC region) or to the extended buffer pool (the XBUF region). (See Section 18.4.7.) The loader can, however, be located in the RTS region since the primary run-time system is not loaded until after the monitor is started.

The loader consists of four segments:

- A copy of the secondary bootstrap code, used as the actual driver to read in the monitor.
- A block map of the monitor SIL, consisting of a word count and logical block number for each contiguous section of the SIL. The block map is terminated with a zero.
- The final load and setup module, containing code to execute the jam list, set up the extended buffer pool, and initialize the memory management hardware. This module exits to the monitor startup code.
- The jam list.

When the loader has been created, START maps it using APR 5 and APR 6 and then jumps to the secondary bootstrap code. When the bootstrap code has completed reading the monitor, the jam list is processed. Following this, the buffer pools are initialized, I-space descriptor registers are set up, and the monitor's stack pointer is loaded. If D-space is to be enabled, the I-space address and descriptor registers are copied to the D-space registers, D-space is enabled, and the I-space registers are marked read-only. Control then transfers to the monitor startup code via an RTI instruction.

### 18.4.10 Bootstrapping a Device (BOOT)

The BOOT option provides a way to simulate a hardware ROM bootstrap. It is used to load another operating system into memory from a disk or tape, or to reload an altered copy of INIT after it has been patched on disk.

The BOOT dialog prompts the user for the name of the device to boot. Valid devices are either disk or magtape.

After determining that the specified device exists, BOOT waits for the terminal to stop printing and then jumps to a device-dependent I/O routine. The bootstrap block is read from the device into memory location 0 and is then started at location 0. If the specified device is not ready or if an error is detected while reading the boot block, a fatal error message is printed and BOOT terminates execution.

Note that on disks the boot block is block 0. However, for magtape, which is not a random-access device, the boot block is the second record on the tape. The first record is reserved for the file label. A standard RSTS/E bootable magtape contains the bootstrap code as the first file on the tape, with standard label information and a terminating tape mark.

### 18.4.11 Invoking Save/Restore (SAVRES)

The SAVRES option is used to invoke the save/restore system program while running INIT. This program is a disk volume backup and copy utility that provides the following main functions:

SAVE Create a copy of a RSTS/E file-structured disk. This function backs up an entire disk volume to disk or magtape. The disk(s) or tape(s) created by this operation are known as a "save set."*

RESTORE Recreate a RSTS/E file-structured disk from a save set.

IMAGE Copy an entire RSTS/E file-structured disk to another disk in such a way that both disks are functionally equivalent.

The save/restore utility is also available online, during normal timesharing. Thus, to permit the use of common code and to reduce software maintenance problems, the save/restore function is structured in a manner quite similar to INIT itself. A root segment (SROOT) contains data structures, device information, buffers and other information common to the various functions. Each main function of the utility is contained in an individual overlay segment:

- DIA - the dialog and device mounting routines
- SAV - the save function
- RES - the restore function
- IMA - the image copy function

Offline (as part of the initialization code), SROOT is linked with the INIT root. Online, it is linked with module ONLSAV — a set of special I/O and data manipulation routines that compensate for the difference between the offline and online environments.** The remaining modules — the overlays — are identical both on- and offline.

When the SAVRES option is invoked, INIT transfers control to the dialog overlay (DIA), which asks all the required questions and ensures that the specified devices are mounted. Then, since one overlay cannot directly call a routine in another overlay, DIA sets a flag word indicating the function requested and calls a special routine in SROOT. This routine transfers control to the proper overlay. When the requested function is complete, control returns to the INIT mainline.

One of the major goals during the implementation of the save/restore function was speed. Thus, when doing a transfer, save/restore bypasses the on-disk directory structure, thereby eliminating directory lookups, window turns, and so on. Instead, save/restore uses the disk storage allocation table to locate all allocated pack clusters and then reads or writes the data directly (including all directory structures).

During a save operation, a secondary bootstrap and a copy of INIT.SYS, various pieces of labelling information, and two copies of the input disk storage allocation table are written on the first volume of the save set.* All allocated clusters of the input disk are then written sequentially to the output medium. The last volume of the save set also contains an extra set of directory blocks, used (if necessary) to recover from any bad blocks discovered on the save set during the restore function. (See the discussion of bad block handling given below.)

[*] Note that there can be more than one volume in a save set.

[**] Many things which can be done successfully offline, under control of INIT, are either prohibited or must be done differently under normal timesharing.

[*] Note that the first volume of a save set is bootable, thereby providing the means of restoring the disk data, right on the volume itself.

During a restore operation, the first copy of the storage allocation table is read into an incore buffer. (The second copy is provided in case a bad block is discovered in the first.) Then, data stored on the save set is written back to the same pack cluster in which it was located on the original disk.

The image copy operation is basically a combination of the save and restore functions that eliminates the intermediate volume(s).

The entire save/restore process is basically quite straightforward, except for the handling of bad disk blocks. Bad blocks can occur on either an input or an output disk and can be either known, as recorded in an existing bad block file, or previously unknown.

During a save operation, existing bad blocks on the RSTS/E (input) disk do not present a problem since, by default, they contain no pertinent data. However, if a previously unknown bad block is detected on the RSTS/E disk, the disk can contain bad data. SAVRES reports the cluster number, file name and account in which the bad block occurred, and then procceds with the transfer.

Bad output blocks (on the save set being created) do not affect the integrity of the data being stored because SAVRES writes save sets in a "linear" fashion. Specifically, the input storage allocation table is scanned for bits which are set, indicating allocated clusters. If a bit is set, the corresponding input cluster is transferred to the output disk or tape. The next allocated cluster is then written immediately following the data of the previous cluster. If an output block is found to be bad, it is merely skipped. Its block number is stored in the save set bad block file, and the transfer proceeds to the next available location. During the restore phase, the save set bad block file is scanned and entries are skipped as necessary.

During a restore operation, previously unknown bad input blocks can be detected on save sets under two circumstances: while reading a data block from a file or while reading a directory block. Note that such an error indicates that the block has "gone bad" since the time at which the save set was created.

- If an error is detected while reading a data block, no corrective action can be made other than reporting the name and account of the file. The block is transferred as normal but the resulting output file may have corrupted data.
- On the other hand, if an error occurs while reading a directory block, an attempt must be made to correct the situation. While a bad block in a data file harms only that single file, a bad block at any point in a directory can render a save set useless. Thus, when a save set is created, an extra copy of all directory blocks is written to the end of the last volume. If a bad directory block is discovered during a later restore operation, SAVRES can then retrieve a good copy of the offending block.

A bad output block detected during a restore or an image copy operation is the most complicated case of bad block handling. Normally, SAVRES restores data to the same pack cluster in which it originally existed. Before transferring the data, it checks the output bad block file to see if there is an entry for the required pack cluster. If so, the storage allocation table is examined to see if there is a free cluster available for relocation. If there is, the SAT is updated and the old and new cluster numbers are noted. If a pack cluster is not in the bad block file, but is found to be bad during an attempted write operation, the cluster is added to an in-core bad block file and the cluster is relocated, as above.

Upon completion of the transfer, a directory pass is made in order to locate accounts or files in which bad blocks occurred. Since actual data has been moved, links and retrieval entries are now invalid, and the directory structure must be updated to reflect all relocated items. To accomplish this, all directories and individual accounts, starting with the first cluster of the MFD, are scanned and integrity checked.

During this procedure, it is possible that portions of a contiguous or "placed" file can be relocated. If this happens, status bits in the file's directory are changed to indicate that the file is no longer contiguous and/or placed. Note that it is possible to relocate an entire contiguous file, but if the file is quite large, this can take a substantial amount of time. In addition, there may not be enough contiguous space on the disk to move the entire file. At any rate, there are frequently many files (such as compiled BASIC-PLUS programs) which are created contiguous only to improve access time but which will run if they are noncontiguous. Thus, the restore and image copy functions do not attempt to reconcile such files, but merely mark their new status.