VMS

digital

Guide to VMS File Applications

Order Number AA–LA78A–TE

Guide to VMS File Applications

Order Number: AA–LA78A–TE

April 1988

This document is intended for application programmers and designers who write programs that use VMS RMS files.

Revision/Update Information:	This document supersedes the *Guide to VAX/VMS File Applications Manual,* Version 4.4.
Software Version:	VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

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.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

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

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC
DEC/CMS
DEC/MMS
DECnet
DECsystem–10
DECSYSTEM–20
DECUS
DECwriter
DIBOL
EduSystem
IAS
MASSBUS
PDP
PDT
RSTS
RSX
UNIBUS
VAX
VAXcluster
VMS
VT
digital™

ZK4506

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript(TM) printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

(TM) PostScript is a trademark of Adobe Systems, Inc.

Contents

EXAMPLES

FIGURES

TABLES

Preface

This guide describes how to use file organizations; how to create, populate, locate, share, and maintain files; and how to process records. It concentrates on the human and program interfaces provided by the File Definition Language Facility (FDL) and VMS Record Management Services (VMS RMS). Both of these interfaces can be used to define file and record characteristics, the location of a file, run-time characteristics, and related options. VMS RMS services can be used to create and access files, and to process records. This guide provides examples of FDL and VMS RMS, as well as the procedures needed to perform tasks usually required in file-based application development.

Intended Audience

This document is intended for applications programmers and designers who create or maintain applications programs that use VMS RMS files. You may also read it to gain a general understanding of the file and record processing options available on a VMS system.

Document Structure

This guide contains ten chapters, one appendix, and a glossary.

- Chapter 1 provides general information on the use of files on a VMS system, including an overview of available media, VMS RMS, FDL, and resource requirements.
- Chapter 2 describes the file organizations and record access modes to help you choose the correct file organization for your application.
- Chapter 3 discusses general performance considerations and specific decisions you can make in the design of your application.
- Chapter 4 describes procedures necessary to create files, populate files with records, and protect files.
- Chapter 5 describes file specifications and the procedures needed to use them.
- Chapter 6 describes the rules of file specification parsing and advanced file specification use. Information about rooted directories is also provided.
- Chapter 7 describes file sharing and buffering, including record locking and the use of global buffers.
- Chapter 8 describes aspects of record processing, including record access modes; synchronous and asynchronous record operations; and retrieving, inserting, updating, and deleting records.
- Chapter 9 describes how to specify run-time options and summarizes the run-time options available when a file is opened and closed and when records are retrieved, inserted, updated, and deleted.
- Chapter 10 describes procedures needed to maintain properly tuned files, with the emphasis on efficiently maintaining indexed files.

- Appendix A describes the algorithms used by the Edit/FDL Utility.

The glossary provides definitions of terms that are commonly used in this guide.

Associated Documents

The reader should be familiar with the information in the following documents:

- The *Introduction to VMS* describes the use of the VMS operating system for a general audience.
- Programmers should be familiar with the appropriate documentation for the VAX language the application will be written in.

Related reference information is available in the following documents:

- The *VMS File Definition Language Facility Manual* contains information about the FDL facility.
- The *VMS Convert and Convert/Reclaim Utility Manual* contains information about the Convert Utility that is often used in conjunction with file applications.
- The *VMS DCL Dictionary* contains information (with examples) about using DCL commands that define system or process defaults, set file protection, or define logical names.
- The *VMS Record Management Services Manual* contains information about calling VMS RMS services directly and about the control block options that are available. This book describes the VMS RMS control blocks that define arguments for VMS RMS service calls performed through language statements or directly from application programs. This document also includes special information for VAX MACRO users, including descriptions of VAX MACRO service and control block macros and examples of VAX MACRO programs.
- The *VMS Utility Routines Manual* contains information about calling FDL routines and Convert routines. It also includes appropriate programming examples.
- Programmers using DECnet to access remote files may need to consult the *VMS Networking Manual* to determine what types of operations are supported for remote files on non-VMS operating systems. Information on accessing remote files on a VMS system can be obtained from the description of the appropriate FDL attributes in the *VMS File Definition Language Facility Manual* and the description of the appropriate control block fields in the *VMS Record Management Services Manual.*

Conventions

Convention	Meaning
RET	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

Version 5.0 supports multinational key enhancements that provide a simplified way to use non-ASCII collating sequences with indexed files. These enhancements are based on the National Character Set Utility that permits you to define alternative collating sequences for special characters. You can define a collating sequence for each key of reference. For example, you can sort a file in German by one key, in French by another key, and so forth.

You implement an alternative collating sequence for a file by specifying the address of the desired collating table. You can also specify whether the collating sequence will be applied in ascending or descending order.

1 Introduction

This chapter illustrates basic data management concepts and how they are applied by the VMS Record Management Services (VMS RMS). VMS RMS is the data management subsystem of the VMS operating system. In combination with the VMS operating system, VMS RMS allows efficient and flexible storage, retrieval, and modification of data on disks, magnetic tapes, and other devices. VMS RMS may be implemented directly at the VAX MACRO level or indirectly through the File Definition Language Facility (FDL). Higher-level languages may also implement VMS RMS through program-specific processing options. Although VMS RMS supports devices such as line printers, terminals, and card readers, the purpose of this guide is to introduce you to VMS RMS record keeping on magnetic tape and disk.

In contrast to magnetic tape storage, disk storage allows faster data access while providing the same virtually limitless storage capacity. Disks provide faster access because the computer can locate files and records selectively without first searching through intervening data. This faster access time makes disks the most appropriate medium for online file processing applications.

1.1 File Concepts

The following file concepts are discussed in this manual:

- Files
- Records
- Fields
- Bytes and bits
- Access modes
- Record formats

A file is an organized collection of data stored on a mass storage volume, such as a disk, and processed by a central processing unit (CPU). Data files are organized to accommodate the processing of data within the file by an application program. The basic unit of electronic data processing is the *record*. A record is a collection of related data that your program treats as an entity. For example, all the information about an employee, such as name, street address, city, and state, constitutes a personnel record. Records are made up of *fields*, which are sets of contiguous bytes. For example, a person's name or address might be a field. A *byte* is a group of binary digits (bits), which are used to represent a single character. You can also think of a field or an item as a group of bytes in a record that are related in some way.

The records in a file must be formatted uniformly. That is, they must conform to some defined arrangement of the fields that make up each record including the lengths of the fields, the location of each field in the record, and the type of data (character strings or binary integers, for instance) in each field. To process file data, an application must know the arrangement of the record fields, especially if the application intends to modify existing records or to add new records to the file.

The *file organization* is the arrangement of data within a file. The file organization, together with the applicable storage medium, determines what techniques are used to store and retrieve data. Currently, VMS RMS supports three file organizations, *sequential, relative,* and *indexed*:

Sequential — Records are arranged one after the other.

Relative — Records occupy cells of equal length, and each cell is assigned a *relative record number,* which represents the cell's position relative to the beginning of the file.

Indexed — Records can either be retrieved randomly or sequentially in sorted order using a key embedded in the record.

The *record access mode* is the way in which a program stores and retrieves data. VMS RMS supports two record access modes.

Sequential Access — Records are stored or retrieved one after another starting at a particular point in the file and continuing in order through the file.

Random Access — Records are stored and retrieved by key, by relative record number, or by file address. Random access by key or by relative record number depends upon the file organization. Indexed file records are stored and retrieved by a *key* in the data record; relative file records are retrieved by their *relative record numbers.*

When a record is accessed randomly by its file address, the distinction is made by its unique location in the file; that is, its *record file address* (RFA).

The *record format* refers to the way all records in a file appear physically on the recording surface of the storage medium and is defined in terms of record length. VMS RMS supports four record formats:

Fixed length — All file records are the same length.

Variable length — Records vary in length.

Variable with fixed-length control — Records do not have to be the same length, but each includes a fixed-length control field that precedes the variable-length data portion.

Stream — Records are delimited by special characters or character sequences called *terminators.* Records with stream format are interpreted as a continuous sequence, or stream, of bytes. The carriage return and the line feed characters are commonly used as terminators.

When you design a file, you specify the file storage medium and the file and record characteristics directly through your application program or indirectly using an appropriate utility. Chapter 2 outlines VMS RMS file organizations, record access modes, and record characteristics in detail.

After VMS RMS creates the file, the application program must consider these record characteristics when storing, retrieving and modifying file records. Chapters 4 and 8 outline techniques for creating, storing, retrieving, and modifying data records in a file.

1.1.1 Disk Concepts

This section describes disk concepts as an aid to understanding how a disk may be configured to enhance data access for improved performance. Disk structures may be defined as either logical or physical and the two types of structure interact with each other to some degree. That is, you cannot manipulate a logical structure without considering the effect on a corresponding physical structure.

1.1.1.1 Logical Structures

VMS RMS disk files reside on *Files–11 On-Disk Structure* disks. The Files–11 On-Disk Structure defines how files are organized on the disk. VMS RMS defines the internal organization of the files and the methods of accessing file data.

Two distinct types of disk structuring are used with VMS RMS: Files–11 On-Disk Structure Level 1 and Files–11 On-Disk Structure Level 2. Structure level 1 is the default for the following operating systems:

- RSX–11M
- RSX–11D
- RSX–11–PLUS
- Micro/RSX

VMS defaults to structure level 2. The primary difference between the two structures is that structure level 2 incorporates control capabilities that permit added features including volume sets (described later).

The term Files–11 On-Disk Structure refers to a logical ordering of the disk, using the following disk structures (listed in order of ascending hierarchy):

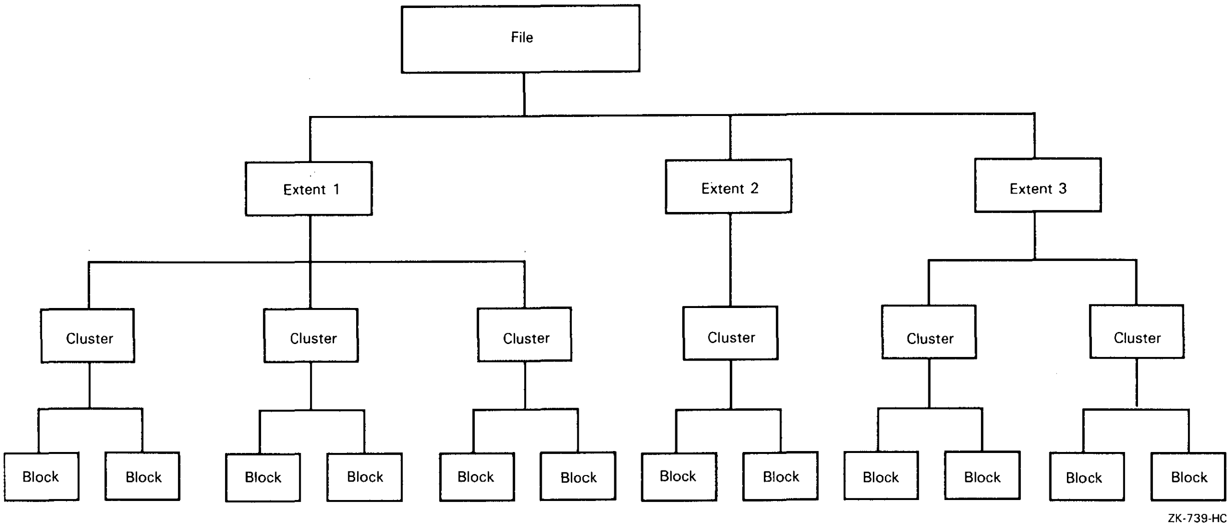
- Blocks
- Clusters
- Extents
- Files
- Volumes
- Volume Sets

Figure 1–1 shows the hierarchy of blocks, clusters, extents, and files in the Files–11 On-Disk Structure.

The next higher level of Files–11 On-Disk Structure is the *volume* (not illustrated) which is the ordered set of blocks that comprise a disk. However, a volume may include several disks that together make up a structure called a *volume set*. Because a volume set consists of two or more related volumes, the system treats it as a single volume.

Note: The terms *disk* and *volume* are used interchangeably in this manual.

Figure 1–1 Files–11 On-Disk Structure Hierarchy



The smallest addressable logical structure on a Files–11 On-Disk Structure disk is a *block*, comprising 512, 8-bit bytes. During input/output operations, one or more blocks may be transferred as a single unit between a Files–11 On-Disk Structure disk and memory.

VMS RMS allocates disk space for new files or extended files using multiblock units called *clusters*. The system manager specifies the number of blocks in a cluster as part of volume initialization.

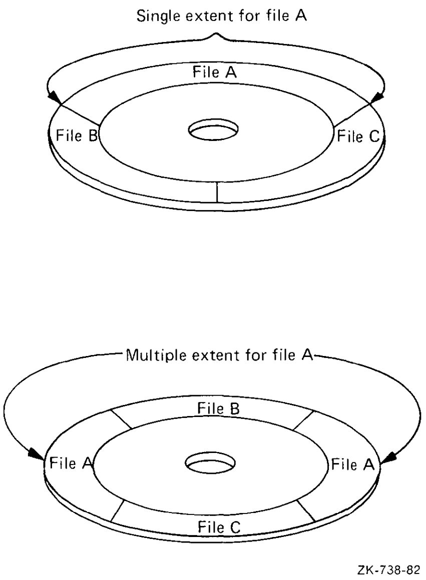
Note: **Do not confuse cluster with a VAXcluster, which is a configuration of multiple VMS systems sharing common resources.**

Clusters may or may not be contiguous, or share a common boundary, on a disk. Cluster sizes may range from 1 to 65,535 blocks. Generally, a system manager assigns a disk with a relatively small number of blocks a small cluster size. Relatively larger disks are assigned a larger cluster size to minimize the overhead for disk space allocation.

An *extent* is one or more adjacent clusters allocated to a file or to a portion of a file. If enough contiguous disk space is available, the entire file is allocated as a single extent. Conversely, if there is not enough contiguous disk space, the file is allocated using several extents, which may be scattered physically on the disk. Figure 1–2 shows how a single file (File A) may be stored as a single extent or as multiple extents.

With VMS RMS, you can exercise varying degrees of control over file space allocation. At one extreme, you can specify the number of blocks to be allocated and their precise location on the volume. At the other extreme, you can allow VMS RMS to handle all disk space allocation automatically. As a compromise, you might specify the size of the initial space allocation and have VMS RMS determine the amount of space allocated each time the file is extended. You can also specify that unused space at the end of the file is to

Figure 1–2 Single and Multiple File Extents



be deallocated from the file, making that space available to other files on the volume.

When you need a large amount of file storage space, you can combine several Files–11 On-Disk Structure volumes into a *volume set* with file extents located on different volumes in the set. You need not specify a particular volume in the set to locate or create a file, but you may improve performance if you explicitly specify a volume for a particular allocation request.

1.1.1.2 Physical Structures

For performance reasons, you should be aware of certain physical aspects of a disk.

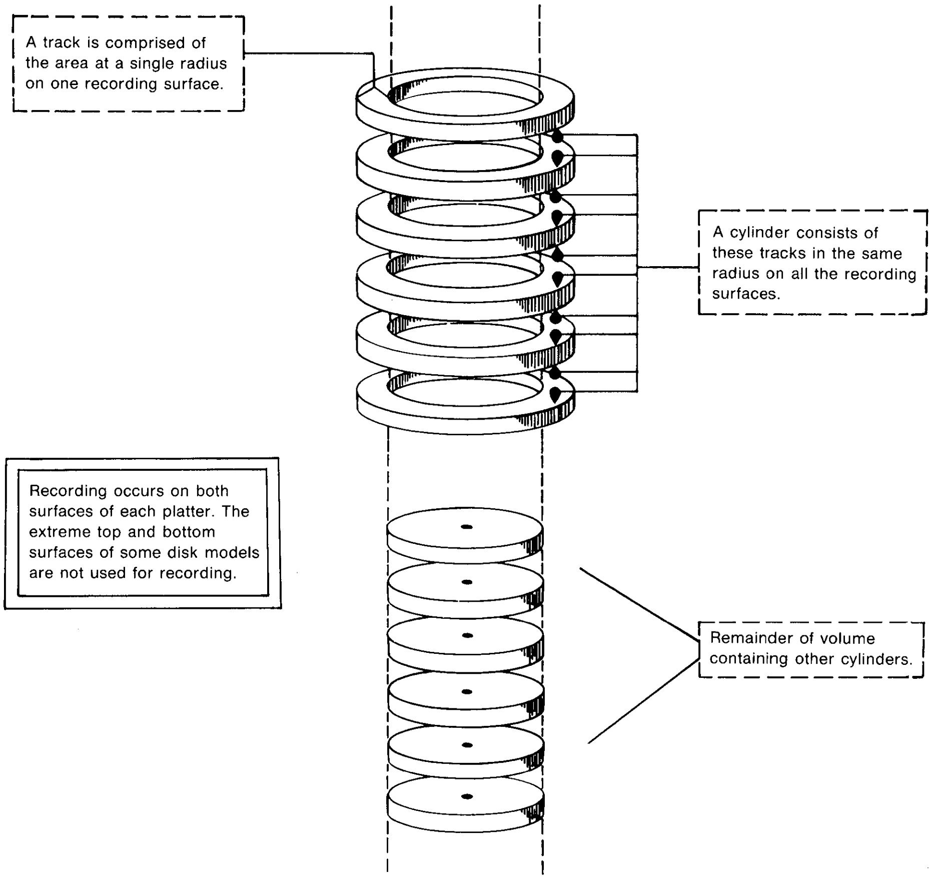
A disk (or volume) consists of one or more *platters* that spin at very high, constant speeds and usually contain data on both surfaces (upper and lower). A *disk pack* is made up of two or more platters having a common center.

Data is located at different distances from the center of the platter and is stored or retrieved using read/write heads that move to access data at various radii from the platter's center. The time required to position the read/write heads over the selected radius (referred to as a *track*) is called *seek time*. Each track is divided into 512-byte structures called *sectors*. The time required to bring the selected sector (logical block) under the read/write heads at the selected radius (track) is called the *rotational latency*. Because seek time usually exceeds the rotational latency by a factor of 2 to 4, related blocks (sectors) should be located at or near the same track to obtain the best performance when transferring data between the disk and VMS RMS-maintained buffers in memory. Typically, related blocks of data might include the contents of a file or several files that are accessed together by a performance-critical application.

Another physical disk structure is called a *cylinder*. A cylinder consists of all tracks at the same radius on all recording surfaces of a disk.

Figure 1–3 illustrates the relationship between tracks and cylinders.

Figure 1–3 Tracks and Cylinders



A track is comprised of the area at a single radius on one recording surface.
A cylinder consists of these tracks in the same radius on all the recording surfaces.
Recording occurs on both surfaces of each platter. The extreme top and bottom surfaces of some disk models are not used for recording.
Remainder of volume containing other cylinders.
ZK-740-82

Because all blocks in a cylinder can be accessed without moving the disk's read/write heads, it is generally advantageous to keep related blocks in the same cylinder. For this reason, when choosing a cluster size for a large-capacity disk, a system manager should consider one that divides evenly into the cylinder size.

1.1.1.3 Files–11 On-Disk Structure Index File

Each Files–11 On-Disk Structure volume has an *index file* containing control information that Files–11 On-Disk Structure software maintains for VMS RMS. This information is transparent to your program but VMS RMS uses it to give your program access to individual file records. The *Guide to VMS Files and Devices* contains details on the various Files–11 On-Disk Structure control structures; but the discussion here is limited to two components of the index file, the *home block* and the *file headers*. The home block provides specific information about the volume including default file values. The following information is included within the home block:

- The volume name
- Information to locate the remainder of the index file
- The maximum number of files that can be present on the volume at any given time
- The user identification code (UIC) of the volume owner
- Volume protection information (specifies which users can read and/or write the entire volume)

The home block identifies the disk as a Files–11 On-Disk Structure volume. Initially, the home block is the second block on the volume. Files–11 On-Disk Structure volumes contain several copies of the home block to ensure that accidental destruction of this information does not affect VMS RMS ability to locate other files on the volume. If the current home block becomes corrupted, the system selects an alternate home block.

The volume's index file provides at least one file header for each file on the volume. Each file header includes the following information:

- File ownership
- Protection
- Creation date
- Creation time

Each file header also contains a list of the extents that define the physical location of the file. When a file has many extents, it may be necessary to have multiple file headers for locating them. When this occurs, each header is assigned a file identifier number to associate it with the appropriate file.

When you create a file, you normally specify a name that VMS RMS assigns to the file on a Files–11 On-Disk Structure volume. VMS RMS places the file name and file identifier associated with the newly-created file in a directory that contains an entry defining the location for each file. To subsequently access the file, you specify its name. The system uses the name to define a path through the directory entry to the file identifier. The file identifier in turn points to the file header that lists the file's extents.

1.1.2 Tape Concepts

You can also use magnetic tape as a file storage medium with VMS RMS. VMS RMS supports the standard magnetic tape structure defined by American National Standard X3.27-1978 (Level 3), *Magnetic Tape Labels and Record Formats for Information Interchange.*

Data records are organized on magnetic tape in the order in which they are entered; that is, sequentially.

Characters of data on magnetic tape are measured in *bits per inch* (bpi). This measurement is called *density.* A 1600-bpi tape can accommodate 1600 characters of data in 1 inch of recording space. A tape has 9 parallel tracks containing 8 bits and 1 parity bit.

A *parity bit* is used to check for data integrity using a scheme where each character contains an odd number of marked bits, regardless of its data bit configuration. For example, the alphabetic character **A** has an ASCII bit configuration of 100 0001 where two bits, the most significant and the least significant, are marked. With an odd-parity checking scheme, a marked eighth bit is added to the character so that it appears as 1100 0001. When this character is transmitted to a receiving station, the receiver logic checks to make sure that the character still has an odd number of marked bits. If media distortion corrupts the data resulting in an even number of marked bits, the receiving station asks the sending station to retransmit the data.

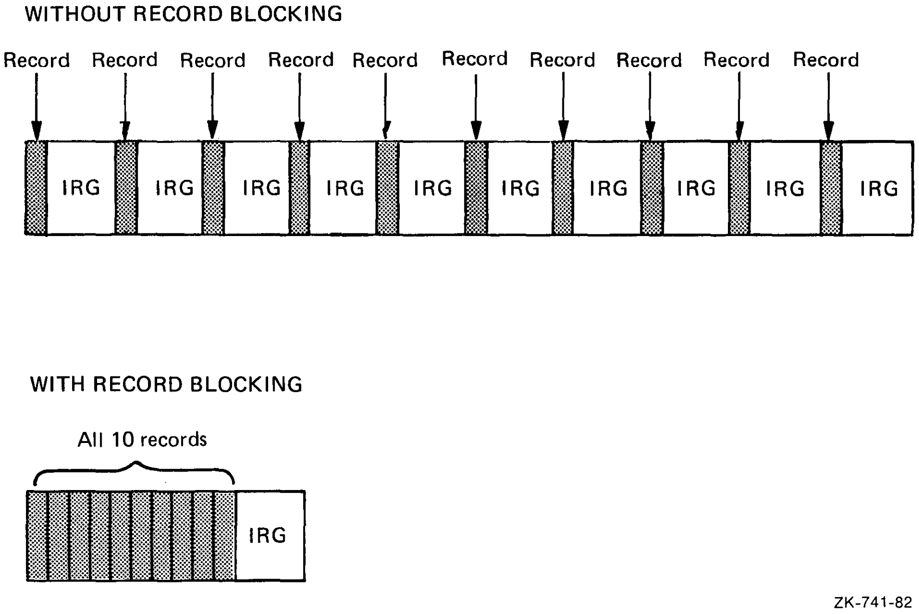
Even though a tape may have a density of 1600 bpi, there are not always 1600 characters on every inch of magnetic tape because of the *interrecord gap* (IRG). The IRG is an interval of blank space between data records that is created automatically when records are written to the tape. After a record operation, this breakpoint allows the tape unit to decelerate, stop if necessary, and then resume working speed before the next record operation.

Each IRG is approximately 0.6 inch in length. Writing an 80-character record at 1600 bpi requires 0.05 inch of space. The IRG, therefore, requires twelve times more space than the data with a resultant waste of storage space.

VMS RMS can reduce the size of this wasted space by using a record blocking technique that groups individual records into a block and places the IRG after the block rather than after each record. (A block on disk is different from a block on tape. On disk, a block is fixed at 512 bytes; on tape, you determine the size of a block.) However, record blocking requires more buffer space for your program, resulting in an increased need for memory. The greater the number of records in a block, the greater the buffer size requirements. You must determine the point at which the benefits of record blocking cease, based on the configuration of your computer system.

Figure 1–4 shows how space can be saved by record blocking. Assume that a 1600-bpi tape contains 10 records not grouped into record blocks. Each record is 160 characters long (0.1 inch at 1600 bpi) with a 0.6-inch IRG after each record; this uses 7 inches of tape. Placing the same 10 records into 1 record block uses only 1.6 inches of tape (1 inch for the data records and 0.6 inch for the IRG).

Figure 1–4 Interrecord Gaps



Record blocking also increases the efficiency of the flow of data into the computer. For example, 10 unblocked records require 10 separate physical transfers, while 10 records placed into a single block require only 1 physical transfer. Moreover, a shorter length of tape is traversed for the same amount of data thereby reducing operating time.

Like disk data, magnetic tape data is organized into files. When you create a file on tape, VMS RMS writes a set of header labels on the tape immediately preceding the data blocks. These labels contain information such as the user-supplied file name, creation date, and expiration date. Additional labels, called *trailer labels,* are also written following the file. Trailer labels indicate whether or not a file extends beyond a volume boundary.

To access a file on tape by the file name, the file system searches the tape for the header label set that contains the specified file name.

When the data blocks of a file or related files do not physically fit on one volume (a reel of tape or a tape cartridge), the file is continued on another volume, creating a multivolume tape file that contains a volume set. When a program accesses a volume set, it searches all volumes in the set. For additional information about magnetic tapes, see the *Guide to VMS Files and Devices.*

1.2 Volume Protection

The system protects data on disk and tape volumes to make sure that no one accesses the data accidentally or without authorization. For disk volumes, the system provides protection at the file, directory, and volume levels. For tape volumes, the system provides protection at the volume level only.

In addition to protecting the data on mounted volumes, the system provides device protection coded into the home block of the disks and tapes. See Section 1.1.1 for more information.

In general, you can protect your disk and tape volumes with user identification codes (UICs) and access control lists (ACLs). The standard protection mechanism is UIC-based protection. Access control lists permit you to customize security for a file or a directory.

UIC-based protection is determined by an owner UIC and a protection code, whereas ACL-based protection is determined by a list of entries that grant or deny access to specified files and directories.

Note: **You cannot use ACLs with magnetic tape files.**

When you try to access a file that has an ACL, the system uses the ACL to determine whether or not you have access to the file. If ACL does not explicitly allow or refuse you access or if the file has no ACL, the system uses the UIC-based protection to determine access. (See the *Guide to VMS System Security* for additional information about system security.)

For detailed information about protecting your files, directories, or volumes, see Section 4.3.

1.3 Record Management Services

VMS Record Management Services (VMS RMS) is the file and record access subsystem of the VMS operating system. Used with the VMS operating system, VMS RMS allows efficient and flexible data storage, retrieval, and modification for disks, magnetic tapes, and other devices.

You can use VMS RMS from low-level and high-level languages. If you use a high-level language, it may not be easy or feasible to use the VMS RMS services directly, because you must allocate control blocks and access fields within them. Instead, you can rely on certain processing options of your language's input/output (I/O) statements or upon a specialized language provided as an alternative to using VMS RMS control blocks directly, the *File Definition Language* (FDL).

If you use a low-level language, you can either use VMS RMS services directly, or you may use FDL.

1.3.1 File Definition Language (FDL)

FDL is a special-purpose language you can use to specify file characteristics. FDL is particularly useful when you are using a high-level language or when you want to ensure that you create properly tuned files. Properly tuned files can be created from an existing file or from a new design for a file. The performance benefits of properly tuned files can greatly affect application and system performance, especially when using large indexed files.

FDL allows you to use all of the creation-time capabilities and many of the run-time capabilities of VMS RMS control blocks including the *file access block* (FAB), the *record access block* (RAB), and the *extended attribute blocks* (XABs).

For more information about FDL, see Section 4.1.2.

1.3.2 VMS RMS Data Structures

VMS RMS control blocks generally fall into two groups: those pertaining to files and those pertaining to records.

To exchange file-related information with VMS RMS file services, you use a control block called a file access block (FAB). You use the FAB to define file characteristics, file specifications, and various run-time options. The FAB has a number of fields, each identified by a symbolic offset value. For instance, the allocation quantity for a file is specified in a longword-length field having a symbolic offset value of FAB$L_ALQ. FAB$L_ALQ indicates the number of bytes from the beginning of the FAB to the start of the field.

To exchange record-related information with VMS RMS record services, you use a control block called a record access block (RAB). You use the RAB to define the location, type, and size of the input and output buffers, the record access mode, certain tuning options, and other information. The symbolic offset values for the RAB fields have the prefix RAB$ to differentiate them from the values used to identify FAB fields. The RAB symbolic offset values have the same general format, where the letter following the dollar sign indicates the field length and the letters following the underscore are a mnemonic for the field's function. For example, the multibuffer count field (MBF) specifies the number of VMS RMS buffers to be used for I/O and has the symbolic offset value RAB$B_MBF. The value of RAB$B_MBF is equal to the number of bytes from the beginning of the RAB to the start of the field.

Where applicable, VMS RMS uses control blocks called *extended attribute blocks* (XABs), together with FABs and RABs, to support the exchange of information with VMS RMS services. For example, a Key Definition XAB specifies the characteristics for each key within an indexed file. The symbolic offset values for XAB fields have the common prefix XAB$.

For more information about VMS RMS control blocks, see Chapter 4.

1.3.3 VMS RMS Services

Because VMS RMS performs operations related to files and records, services generally fall into one of two groups:

- Services that support file processing. These services create and access new files, access (or open) previously created files, extend the disk space allocated to files, close files, get file characteristics, and perform other functions related to the file.
- Services that support record processing. These services get (extract), find (locate), put (insert), update (modify), delete (remove) records and perform other record operations.

For more information about the VMS RMS services, see Chapters 7 and 8.

1.4 VMS RMS Utilities

The following are VMS RMS file-related utilities:

- The Analyze/RMS_File Utility
- The Convert Utility
- The Convert/Reclaim Utility
- The Create/FDL Utility
- The Edit/FDL Utility

These utilities let you design, create, populate, maintain, and analyze data files that can use the full set of VMS RMS create-time and run-time options. They help you create efficient files that use a minimum amount of system resources, while decreasing I/O time.

For more information, see the appropriate utility manual.

1.4.1 The Analyze/RMS_File Utility

With the Analyze/RMS_File Utility (ANALYZE/RMS_FILE), you can perform five functions:

- Inspect and analyze the internal structure of a VMS RMS file
- Generate a statistical report on the file's structure and use
- Interactively explore the file's internal structure
- Generate an FDL file from a VMS RMS file
- Generate a summary report on the file's structure and use

ANALYZE/RMS_FILE is particularly useful in generating an FDL file from an existing data file that you can then use with the Edit/FDL Utility (also called the FDL editor) to optimize your data files. Alternatively, you can provide general tuning for the file without the FDL editor.

To invoke the Analyze/RMS_File Utility, use the following DCL command:

```
ANALYZE/RMS_FILE filespec[,...]
```

The **filespec** parameter lets you select the data file you want to analyze.

For more information about the Analyze/RMS_File Utility, refer to Chapter 10 of this manual and the *VMS Analyze/RMS_File Utility Manual.*

1.4.2 The Convert Utility

The Convert Utility (CONVERT) copies records from one or more files to an output file, optionally changing the record format and file organization to that of the output file.

CONVERT is particularly useful in the tuning cycle of a file. After you have analyzed and optimized the file, you can use CONVERT to create a new file having the new, optimized characteristics and to copy the records in the old file to the new file. You can also use CONVERT to reformat an indexed file that has had many record insertions or deletions.

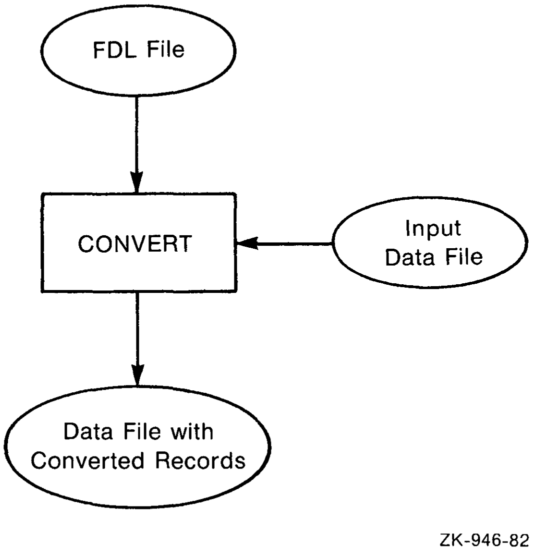
To invoke the Convert Utility, use the following DCL command:

```
CONVERT input-filespec[,...] output-filespec
```

Use the **input-filespec** parameter to specify the file or files you want to convert, and use the **output-filespec** parameter to specify a destination file for the converted records.

Figure 1–5 shows how CONVERT creates data files and loads them with converted records from an input file.

Figure 1–5 Using CONVERT to Create a Data File



For more information about the Convert Utility, refer to Chapter 4 and the *VMS Convert and Convert/Reclaim Utility Manual.*

1.4.3 The Convert/Reclaim Utility

The Convert/Reclaim Utility reclaims empty buckets in Prolog 3 indexed files so that new records can be added to them. A bucket is a storage structure that VMS RMS uses to build and process files.

The Convert/Reclaim Utility does an "in-place" reorganization of the file in contrast to the Convert Utility, which creates a new file from the old file. For this reason, the Convert/Reclaim Utility is more appropriate for large disk files where space is limited. Before using the Convert/Reclaim Utility, be sure to back up the file.

For more information about the Convert/Reclaim Utility, see Chapter 4 of this manual the *VMS Convert and Convert/Reclaim Utility Manual.*

1.4.4 The Create/FDL Utility

The Create/FDL Utility (CREATE/FDL) uses the specifications in an existing FDL file to create a new, empty data file.

To invoke this utility, use the following DCL command:

```
CREATE/FDL=fdl-filespec [filespec]
```

The **fdl-filespec** parameter specifies the source FDL file for creating the data file. The **filespec** parameter gives you the option of assigning a file specification to the data file.

Figure 1–6 shows how the CREATE/FDL Utility creates empty data files from the specifications in an FDL file.

For more information about the CREATE/FDL Utility, see Chapter 4 and the *VMS File Definition Language Facility Manual.*

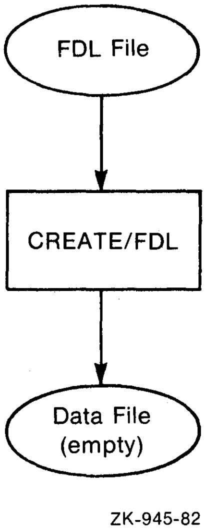
1.4.5 The Edit/FDL Utility

The Edit/FDL Utility (EDIT/FDL) creates and modifies files that contain specifications for VMS RMS data files. The specifications are written in the file definition language, and the files are called FDL files.

A completed FDL file is an ordered sequence of file attribute keywords and their associated values. By using an FDL file to specify the characteristics of a data file, you can use most of the VMS RMS capabilities without having to access the VMS RMS control blocks directly.

While you are designing the data model, EDIT/FDL informs you of syntax errors and the effects of altering file characteristics. Using EDIT/FDL, you can experiment with attributes that are critical to the record-processing performance of the file, and you can calculate optimum file size.

Figure 1–6 Using CREATE/FDL to Create an Empty Data File



For example, the depth of an index is an important consideration in designing an indexed file, and bucket size is one variable that determines the number of levels. EDIT/FDL can show the effects of varying the bucket size on the index depth to help you choose the optimum bucket size.

To invoke this utility, use the following DCL command:

```
EDIT/FDL  fdl-filespec
```

The *fdl-filespec* parameter specifies the FDL file you want to create, modify, or optimize.

For more information about the Edit/FDL Utility, see the *VMS File Definition Language Facility Manual.*

1.5 Process and System Resources for File Applications

To use VMS RMS files efficiently, your application requires various process and system resources. You may have to adjust specific resources and quotas for the process running a file application. Before using VMS RMS options, you should consider their impact on process and system resources. In some cases, you may need additional memory or disk drives to ensure that sufficient system resources are available.

1.5.1 Memory Requirements

One of the most important ways to improve application performance is to allocate larger buffer areas or more buffers for an application. As described in Chapter 7, the number of buffers and the size of buckets and blocks can be fine tuned on the basis of the way the file will be accessed. For indexed files, the index structure and other factors must also be considered.

When a file is opened or created, VMS RMS maintains the buffers and control structures charged to process memory use. Memory use generally increases with the number of files to be processed at the same time. The amount of memory needed for I/O buffers can vary greatly for each file, but the amount of memory needed for control structures is fairly constant.

The memory use (working set) of a process is governed by three resource limits:

- Working set default (WSDEFAULT)
- Working set quota (WSQUOTA)
- Working set extent (WSEXTENT)

These values can ensure that the process has sufficient memory to perform the application with minimum paging. For a complete description of these limits, see the *Guide to Setting Up a VMS System.*

In addition to process requirements, you may want a shared file to use global buffers to avoid needless I/O when the desired buffer is already in memory. Global buffer usage is limited by the following SYSGEN parameters:

- VMS RMS global buffer quota (RMS_GBLBUFQUO)
- Global sections (GBLSECTIONS)
- Global pages (GBLPAGES)
- Global page-file pages (GBLPAGFIL)

When DCL opens a process-permanent file, VMS RMS places internal structures for the file in a special portion of P1 space called the *process I/O segment.* The segment size is determined by the SYSGEN parameter PIOPAGES and cannot be expanded dynamically. If there is insufficient space in the process I/O segment for the internal structures, DCL generates an error message and does not open the file.

For a complete description of these parameters, see the *VMS System Generation Utility Manual.*

1.5.2 Process Limits

If you anticipate asynchronous record I/O or are going to access a shared file, you should consider the following process limits:

- Asynchronous system trap limit (ASTLM)
- Buffered I/O limit (BIOLM)
- Direct I/O limit (DIOLM)
- Enqueue quota limit (ENQLM)
- Open file limit (FILLM)

For a complete description of these process limits, see the *Guide to Setting Up a VMS System*.

2 Choosing a File Organization

When you write an application program, you want the program to input data, process it, store it, update it if necessary, and output it at the right time in the right format. Moreover, the program should perform these functions quickly and accurately.

To achieve this objective, you should consider the structure of your data files and the data processing capabilities available to you through VMS RMS.

You should consider these factors when you write the application program, especially if you have many users simultaneously accessing large files, or if you have a high level of file activity where many records are stored, retrieved, updated, or deleted in a given time period.

You may later reconsider these factors if you are not satisfied with the application program's performance.

This chapter describes file design and structure to help you make the first important design decision: selecting a file organization. Section 2.1 covers record access modes and formats. Section 2.2 describes file concepts and organization.

See Chapter 3 for a description of performance criteria that will help you to evaluate the performance of your data files.

All of the VMS RMS features described in this chapter are available at the VAX MACRO programming level, and many are available to higher-level programming languages that use FDL as an intermediary to the VMS RMS control blocks. (See the descriptions of the FDL routines in the *VMS Utility Routines Manual* for details.)

High-level languages may support only a subset of VMS RMS features. If you intend to use VMS RMS from a high-level language, refer to your language manual to determine the VMS RMS capabilities available to you.

2.1 Record Concepts

In considering the structure of your data files, note that a file is an ordered collection of logically related records treated as a unit.

One design consideration is the way records are transferred to your program from storage. For disk files, the smallest unit of transfer is a *block,* but records are usually transferred in multiple blocks using transfer units that are primarily dictated by file organization. If you use the sequential file organization, the multiblock run-time option allows multiple blocks to be transferred during a single I/O operation. Relative files and indexed files use buckets to transfer records. A *bucket* is a storage structure, consisting of 1 to 63 blocks, that is used for building and processing relative and indexed files.

Another design consideration is how records are accessed. This is called the *record access mode.* The record access mode specifies the way your program stores and retrieves file records.

A third consideration in designing files is how records are formatted. The program that creates the file specifies its record format. Any program that accesses the file must conform to the defined record format.

A fourth consideration is record layout. The record layout defines the number and length of record fields. For example, a program that creates records in a payroll file might use a record layout containing the following fields:

- Employee name
- Social security number
- Pay rate
- Deductions

The next two sections describe VMS RMS record access modes and record formats, respectively.

2.1.1 Record Access Modes

VMS RMS provides two record access modes, sequential access and random access. Random access can be further catalogued as one of the three following modes:

- Random access by key value
- Random access by relative record number
- Random access by record file address (RFA)

Although you cannot change its file organization after you create a file, you can change the record access mode each time you access a record in the file. For example, a relative file can be processed in sequential record access mode one time and in a random access mode the next time. Table 2–1 lists the combinations of record access modes and file organizations supported by VMS RMS.

Table 2–1 Record Access Modes and File Organizations

	File Organization		
Access Mode	**Sequential**	**Relative**	**Indexed**
Sequential	Yes	Yes	Yes
Random by relative record number	Yes[1]	Yes	No
Random by key value	No	No	Yes
Random by record file address	Yes[2]	Yes	Yes

[1] Permitted with fixed-length record format on disk devices only

[2] Permitted on disk devices only

The following sections describe the record access modes and the capability for switching from one mode to another during program execution.

2.1.1.1 Sequential Access

In sequential access mode, storage or retrieval begins at a designated point in the file and continues sequentially through the file. VMS RMS begins accessing records at the start of the file, unless you specify the starting point explicitly or establish a starting point through a previous operation.

In the sequential access mode, your program issues a series of requests to VMS RMS to retrieve or store succeeding records in a file. Before acting on these requests, VMS RMS checks the file organization to determine how to proceed. The following sections describe how VMS RMS handles sequential access for each of the three file organizations.

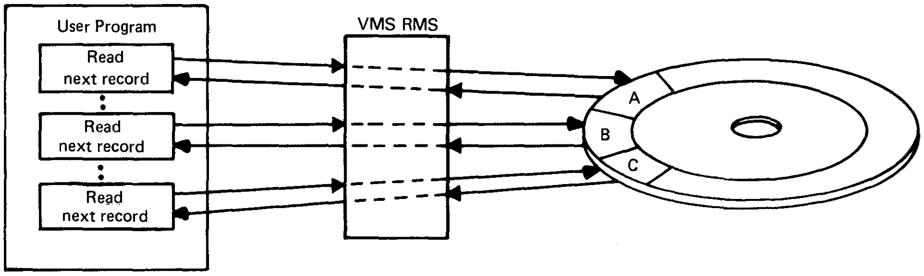
Sequential Access to Sequential Files

In a sequential file, records are stored adjacent to each other. To retrieve a particular record within the file, your program must open the file and successively retrieve all records between the current record position and the selected record.

Figure 2–1 shows a disk volume surface. Each lettered section on the surface represents a record in a sequential file, beginning with record A. When the program requests sequential access to the file records, VMS RMS interprets each request in the context of the file's organization.

Because this particular file is sequential, VMS RMS complies with each request (except for the first request) by accessing the record immediately following the previously accessed record. For example, after VMS RMS accesses record A, it updates the current-record position to record B in anticipation of the next request.

Figure 2–1 Sequential Access to a Sequential File



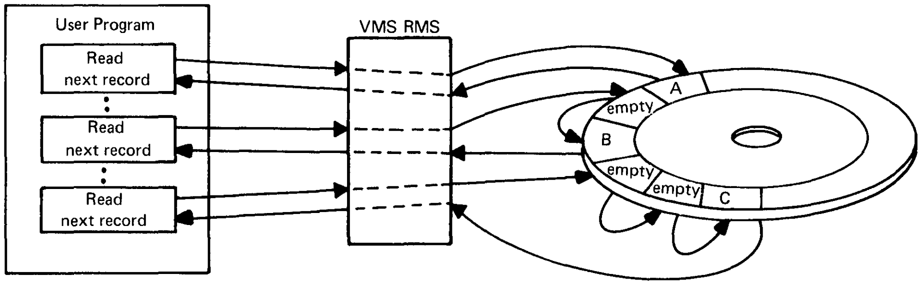
There are limitations imposed by sequential access. When accessing data sequentially, a program can access a previous record only by reopening or rewinding the file, or by switching to a random access mode. (See Chapter 8 for details.) Another limitation of sequential access is that you can add records only to the end of the file.

Sequential Access to Relative Files

Relative files may be accessed sequentially even if some of the fixed-length file cells are empty (because records were never stored in them or because records were deleted from them). If a cell is empty, VMS RMS ignores it and sequentially searches for the next cell that contains a record. For example, assume a relative file contains records only in cells 1, 3, and 6. VMS RMS responds to a sequential retrieval request by retrieving the record in cell 1, then the record in cell 3, then the record in cell 6.

Figure 2–2 shows how VMS RMS checks each cell, ignores an empty cell when it finds one, and then checks the next cell for a record.

Figure 2–2 Sequentially Retrieving Records in a Relative File



ZK-748-82

When storing records sequentially in a relative file, VMS RMS places each new record in the cell whose relative record number is one higher than the most recently accessed cell, provided the cell is empty. If the cell is not empty, the new record cannot be stored in it. Instead, VMS RMS returns an error status.

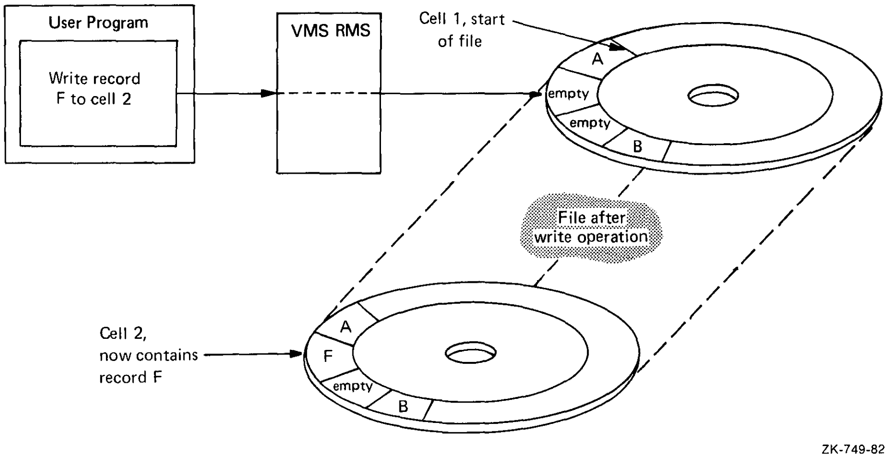
As Figure 2–3 shows, the program directs VMS RMS to store record F in cell 2. Record A already occupies cell 1 but cell 2 is empty, so VMS RMS can store the record in this cell. If this request is followed by a request to sequentially store the next record, VMS RMS stores the record in cell 3, which is also empty. However, if the program tries to store a new record in the next cell (which already contains record B), the attempt fails.

Note that although VMS RMS cannot store a new record in a cell that is already occupied, your program is permitted to modify the record currently occupying the cell.

Sequential Access to Indexed Files

When a program sequentially accesses an indexed file, VMS RMS uses one or more indexes to determine the order in which to process the file records. Because index entries are ordered by key values, an index represents a logical ordering of the records in the file. If you define more than one key for the file, each index associated with a key will represent a different logical ordering of the records in the file. Your program, then, can use the sequential access mode to retrieve records in the logical order represented by any index.

Figure 2–3 Sequentially Storing Records in a Relative File



To retrieve records sequentially from an indexed file, your program must first specify to VMS RMS a key of reference (for example, primary key, first alternate key, second alternate key, and so on). For succeeding retrievals, VMS RMS uses the appropriate index to retrieve records based on how the records are ordered in the index. If VMS RMS accesses the index in ascending sort order, it returns the record with a key value equal to or higher than the key value in the previously accessed record. Conversely, if VMS RMS accesses records in descending order, it accesses the next record having a key value equal to or lower than the key value in the previously accessed record.

In contrast to a request to retrieve data sequentially from an indexed file, a request to store data sequentially in an indexed file does not require a key of reference. Rather, VMS RMS uses the stored definition of the primary key to place the record in the primary index and, where applicable, VMS RMS uses the definition of the appropriate alternate key to place a record pointer in the alternate index. When a program issues a series of requests to sequentially store data, VMS RMS verifies that the key value in each successive record is in the specified order.

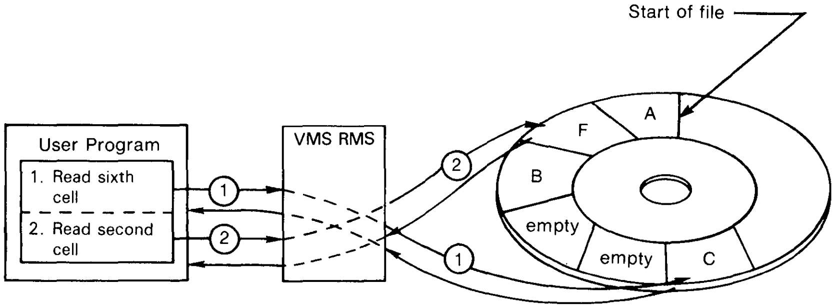
2.1.1.2 Random Access by Key Value or Relative Record Number

Random access is supported for all relative files, all indexed files, and a restricted set of sequential disk files—those having fixed-length records. In random access mode, your program (not the file organization) determines the record processing order. For example, to randomly access a record in a relative file or a record in a sequential disk file having fixed-length records, your program must provide VMS RMS with the relative record number of the cell containing the record. Similarly, to randomly access a record from an indexed file, your program must provide VMS RMS with the appropriate key of reference and key value.

Random Access to Sequential and Relative Files

Unlike sequential access, random access follows no specific pattern. Your program may make successive requests for storing or retrieving records anywhere within the file. In Figure 2–4, the program directs VMS RMS to retrieve the record from the sixth cell in a relative file (record C) and then requests VMS RMS to retrieve record F, which occupies the second cell.

Figure 2–4 Random Access by Relative Record Number



Compare Figure 2–4 with Figures 2–1 and 2–2.

Random Access to Indexed Files

To randomly access a record from an indexed file, your program must specify both a key value and the index that VMS RMS must search (for example, primary index, first alternate key index, second alternate key index, and so on). When VMS RMS finds a record with a matching key value, it passes the record to your program.

Your program can use several methods to randomly access a record by key:

- Exact match of key values.
- Approximate match of key values. When accessing an index in ascending sort order, VMS RMS returns the record that has the next higher key value. Conversely, when the index is accessed in descending sort order, VMS RMS returns the record that has the next lower key value.
- Generic match of key values. Generic matching is applicable to string data-type keys only. For a generic match, the program need specify only a match of some specified number of leading characters in the key.
- Combination of approximate and generic match.

Chapter 8 describes these key match conditions in more detail.

In contrast to record retrieval requests, program requests to store records randomly in an indexed file do not require the separate specification of a key value. All keys (primary and any alternate key values) are in the record itself.

When your program opens an indexed file to store a new record, VMS RMS uses the key definitions stored in the file to find each key field in the record and to determine the length of each key. After writing the new record into the file, VMS RMS uses the record's key values to make appropriate entries in the related indexes so that the record can be subsequently accessed using any of its key values.

2.1.1.3 Random Access by Record File Address

Every record on disk has a unique file address—the *record file address (RFA)*—that provides another way to randomly retrieve records in all types of file organizations.

Note: **RFA mode provides the *only* means of randomly accessing variable-length records in a sequential file.**

An important feature of the RFA is that it remains constant as long as the record is in the file. VMS RMS returns the RFA to your program each time the record is retrieved or stored. Your program can either ignore the RFA or keep it as a random-access pointer to the record for subsequent accesses.

Figure 2–5 contains two illustrations. The first shows that when a record is stored in a file, its RFA is returned to the program. The second shows that when the program wants to subsequently access this record randomly, it simply provides VMS RMS with the RFA.

2.1.2 Record Formats

Except for the key values that are part of the records stored in indexed files, VMS RMS is not concerned with record content. Rather, it looks at the record's format, that is, the way the record physically appears on the recording surface of the storage medium.

VMS RMS supports four record formats:

- Fixed-length format
- Variable-length format
- Variable-length with fixed-length control field (VFC) format
- Stream format

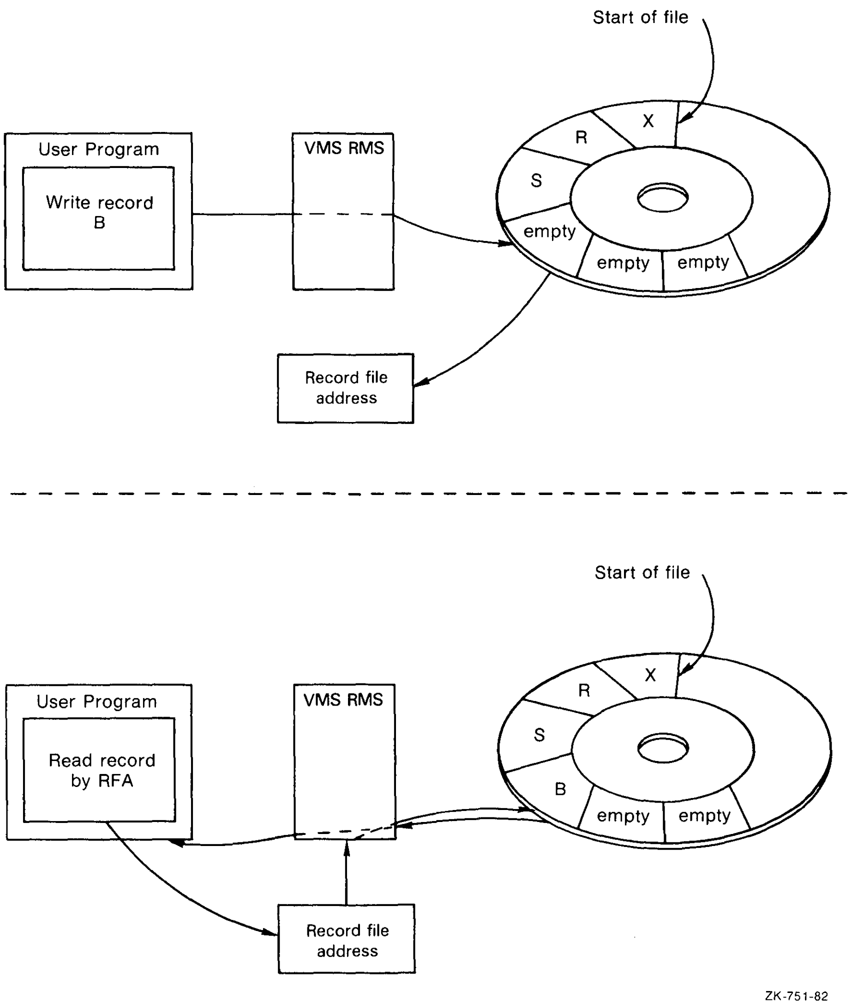
The fixed-length and variable-length record formats are supported for all three file organizations. The variable-length with fixed-length control field (VFC) record format is supported only for sequential and relative files.

Note: **In relative files, all records are stored in fixed-length cells regardless of their format.**

The stream format is supported for sequential files only.

At the VAX MACRO level, you may specify the record format for a file directly by using the FAB$B_RFM field in the FAB.

Figure 2–5 Random Access by Record File Address



2.1.2.1 Fixed-Length Record Format

When you specify fixed-length record format, all file records are the same length. The record length set at file-creation time cannot be changed. It becomes part of the information that VMS RMS stores and maintains for the file.

For the fixed-length record format, each record occupies the same amount of space in the file, and the specified length must be able to accommodate the longest record in the file. If any record fields are not used, your program must be able to detect them and provide appropriate error processing.

2.1.2.2 Variable-Length Record Format

When the variable-length record format is specified, each record is only as long as the data within it requires. When VMS RMS stores a variable-length record in a file, it prefixes a *count field* to the record. The count field contains the number of bytes in the record to accommodate retrieval. VMS RMS builds the count field from information in your program and treats it separately from the associated record data field.

VMS RMS uses the following two types of variable-length record formats, V format and D format:

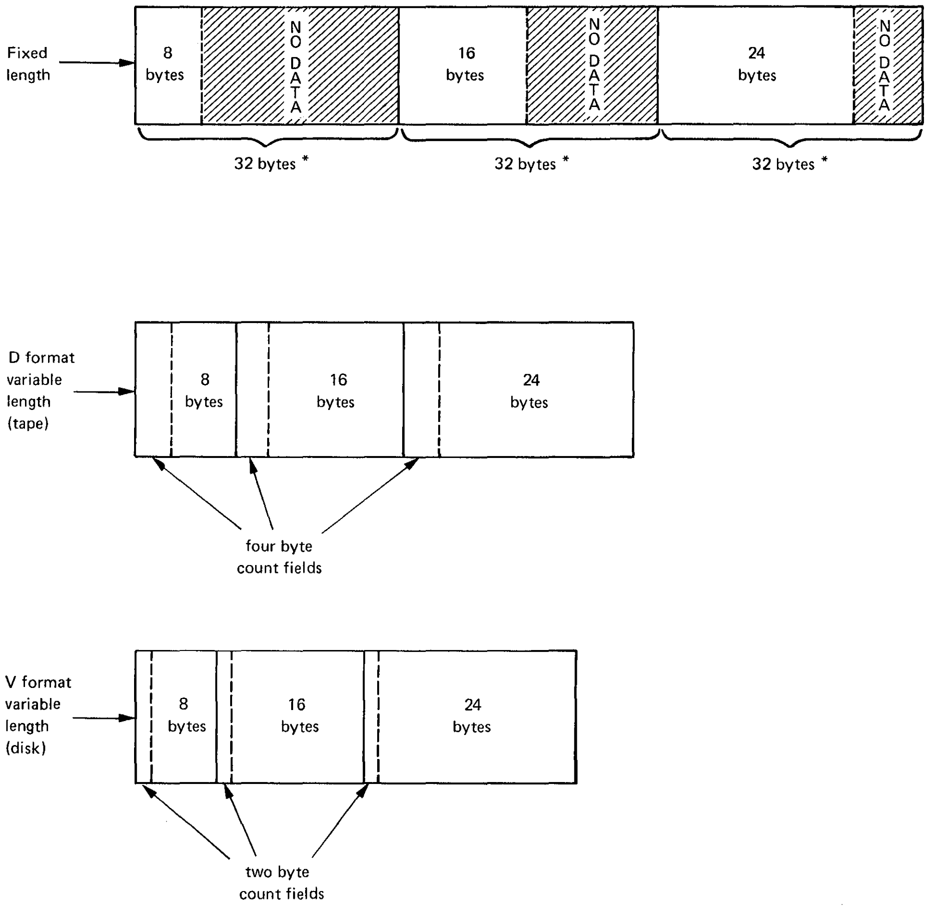
V format — Applies to variable-length records in disk files. VMS RMS prefixes the data portion of each record with a 2-byte binary count field that specifies the length of the record in bytes, excluding the byte field itself.

D format — Applies to variable-length records in tape files. To comply with the American National Standard X3.27-1978 (Level 3), *Magnetic Tape Labels and Record Formats for Information Interchange*, VMS RMS stores a 4-byte decimal count field before the data field of each record on a magnetic tape volume. In contrast to V-format records, the count field is considered as part of the record; but before returning the count, VMS RMS adjusts it to include only the length of the record data.

When you create a file of variable-length records, specify the value (in bytes) of the largest record permitted in the file. Any attempt to store a record containing more bytes than the specified value results in an error. If you specify a value of 0, any length record can be stored in the file; however, you must consider the bucket capacity limitation defined for relative and indexed files.

Figure 2–6 compares fixed-length record formats and variable-length record formats as they apply to sequential files. Each format shows a portion of a file that contains three records. The comparable record in each format contains the same number of bytes. The first record has 8 bytes, the second, 16, and the third, 24. For the fixed-length record format, the record length is set at 32 bytes. Therefore, VMS RMS considers all 32 bytes to be used.

Figure 2–6 Comparison of Fixed- and Variable-Length Records



*VMS RMS considers all 32 bytes to be used, even though they may not contain useful information in the eyes of the user.

ZK-754-82

Clearly, variable-length records can save space; but if records are updated in place, you should consider trading off some space efficiency for update flexibility. All records in a relative file are stored in fixed-length cells. Here, variable-length records do not save space; in fact, the two count-field bytes prefixing each record actually consume additional space.

In the indexed file organization, record length is limited by the capacity of the data bucket and the maximum record size.

2.1.2.3 Variable-Length with Fixed-Length Control Field (VFC) Record Format

VFC records are similar to variable-length records except that a fixed-length control field is prefixed to the variable-length data portion. Unlike variable-length records, VFC records cannot be used in indexed files.

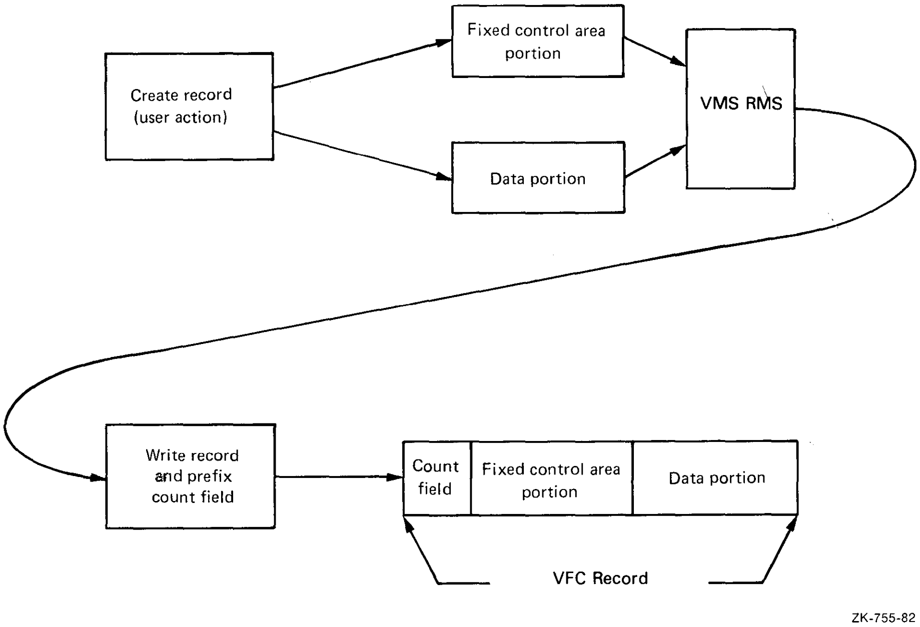
When you create a file for VFC records, you must specify the value (in bytes) of the longest record permitted in the file. Any attempt to store a record containing more bytes than the specified value results in an error. If you specify a value of 0, any length record can be stored in the file.

You must also specify the value in bytes of the fixed-length control field. The fixed-length control field lets you include within the record additional data that may have no direct relationship to the other contents of the record. For example, the fixed-length control field may contain line-sequence numbers for every record in the file. The program does not use the line-sequence numbers, but they are helpful in locating records during file editing.

At the VAX MACRO level, you establish the length of the control field for VFC records using the FAB$B_FSZ field in the FAB. The Open, Create, and Display services provide the control field length in the XAB$B_HSZ field of the File Header Characteristic XAB. For more information, see the *VMS Record Management Services Manual.*

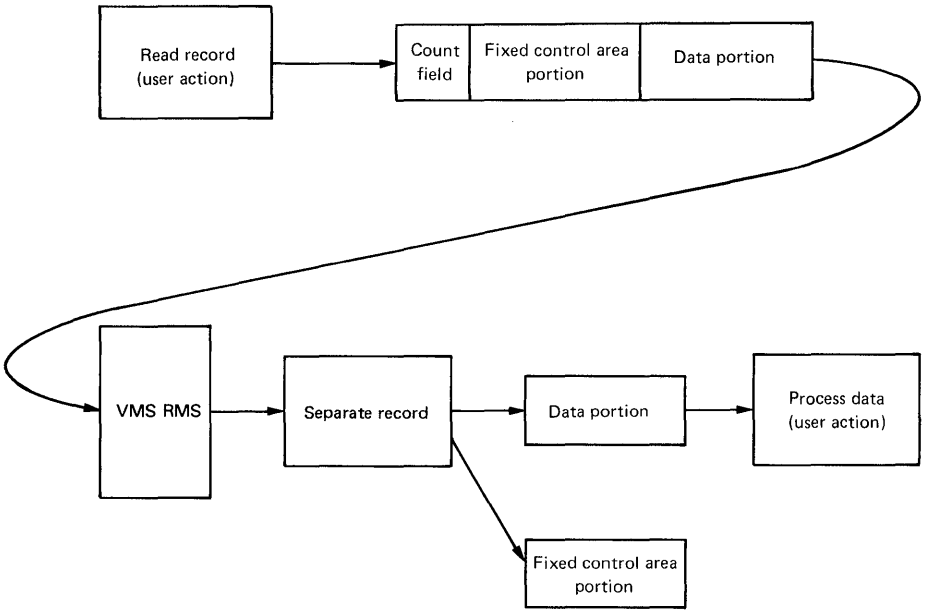
When writing a VFC record to a file, VMS RMS merges the fixed-length control field with the variable-length record data and prefixes the merged record with the count field. Figure 2–7 shows how VMS RMS writes a VFC record to a file.

Figure 2–7 Writing a VFC Record to a File


Create record
(user action)
Fixed control area portion
Data portion
VMS RMS
Write record and prefix count field
Count field
Fixed control area portion
Data portion
VFC Record
ZK-755-82

When VMS RMS reads a VFC record, it uses the count field to determine the overall length of the record, and it uses the appropriate file attribute to determine the length of the control field. After subtracting the control-field length from the overall record length, VMS RMS uses the result to separate the data from the control information. It then processes the data and stores the control information in a designated storage area for program use, if applicable. See Figure 2–8.

Figure 2–8 Retrieving a VFC Record


Read record (user action)
Count field
Fixed control area portion
Data portion
VMS RMS
Separate record
Data portion
Process data (user action)
Fixed control area portion

ZK-756-82

2.1.2.4 Stream Record Format

Special characters or character sequences called *terminators* delimit the records in files using the stream record format. VMS RMS treats the terminators as an integral part of each record.

There are three variations of stream record format:

STREAM_CR	This variation uses a carriage return as the terminator.
STREAM_LF	This variation uses a line feed as the terminator.
STREAM	This variation uses a terminator from a limited set of special characters: the carriage return (CR); the carriage-return/line-feed combination (CR/LF); or the form feed (FF).

In a stream-formatted file, the data is treated as a continuous stream of bytes, without control information. VMS RMS supports the stream record format for sequential files on disk devices *only*.

2.2 File Organization Concepts

The terms *file organization* and *access mode* are closely related, but they are distinct from each other, nonetheless.

You establish the physical arrangement of records in the file—the file organization—when you create it. The organization of a file cannot be changed unless you use a utility conversion routine (such as the Convert Utility) to create the file again with a different organization.

One of the file attributes you specify prior to creating a file is how records are inserted into it and subsequently retrieved from it—the access mode.

The terms *file organization* and *access mode* are sometimes confused because they share common elements. That is, files are *organized* sequentially, relative to some reference value, or by keyed index value. Similarly, a file may be *accessed* sequentially, relative to some reference value, or by using a keyed index value. The following sections emphasize the distinctions between the types of file organization.

Table 2–2 lists important features of each file organization.

Table 2–2 File Organization Characteristics

Characteristics	Sequential	Relative	Indexed
MEDIUM			
Disk	Yes	Yes	Yes
Magnetic tape	Yes	No[1]	No[1]
Unit record[5]	Yes	No	No
RECORD FORMATS			
Fixed-length	Yes	Yes	Yes
Variable-length	Yes	Yes	Yes
VFC (disk only)	Yes	Yes	No
Stream (disk only)	Yes	No	No
Undefined (disk only)	Yes	No	No
OVERHEAD PER RECORD			
	0, 1, or 2 bytes[2]	1 or 3 bytes[3]	7 to 13 bytes[4]

[1] Although these file organizations are not compatible with magnetic tape operations, you may use magnetic tape to transport the files.

[2] Fixed-length records and records with undefined format use no overhead; stream records use either 1 or 2 bytes of overhead; variable-length and VFC records use 2 bytes of overhead.

[3] Fixed-length records use 1 byte of overhead; variable-length records and VFC records use 3 bytes of overhead; extra overhead applies to each cell.

[4] Prolog 1 and Prolog 2 fixed-length records use 7 bytes of overhead. Prolog 1 and Prolog 2 variable-length records use 9 bytes of overhead. For Prolog 3, fixed-length records use 9 bytes of uncompressed overhead, and variable-length records use 11 bytes of uncompressed overhead. For key compression, add 2 bytes of overhead.

[5] Unit record devices include printers, terminals, card readers, mailboxes, and so forth.

Table 2–2 (Cont.) File Organization Characteristics

Characteristics	Sequential	Relative	Indexed
RECORD OPERATIONS			
Connect	Yes	Yes	Yes
Delete	No	Yes	Yes
Disconnect	Yes	Yes	Yes
Find	Yes	Yes	Yes
Flush	Yes	Yes	Yes
Free	Yes	Yes	Yes
Get	Yes	Yes	Yes
Rewind	Yes	Yes	Yes
Truncate	Yes	No	No
Update (disk only)	Yes	Yes	Yes
Put	Yes	Yes	Yes
I/O UNIT			
	1 or more blocks	Bucket	Bucket
I/O TECHNIQUES			
Deferred write	Normal mode	Selectable	Selectable
Multiblock count	Yes	Bucket size	Bucket size
Multiple access streams	Yes	Yes	Yes
Multiple buffers	Yes	Yes	Yes
Access sharing	Read/write	Read/write	Read/write
Other features	Block-spanning records	Maximum record number	Areas

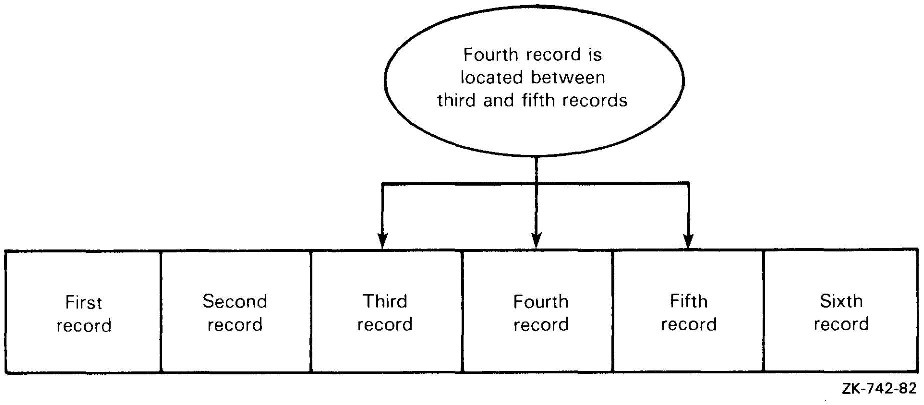
The next three sections describe file organizations.

2.2.1 Sequential File Organization

VMS RMS supports the sequential file organization for all device types. It is the only organization supported for nondisk devices.

In sequential file organization, records are arranged one after the other in the order in which they are stored in the file. For example, the fourth record is located between the third and fifth records, as illustrated in Figure 2–9.

Figure 2–9 Sequential File Organization



You cannot insert new records between existing records because no physical space separates them. Therefore, you can only add records to the current end of the file, that is, immediately following the most recently added record. For the same reason, you cannot add to the length of an existing record when updating it.

Some advantages and disadvantages of the sequential file organization are outlined in Table 2–3.

Table 2–3 Sequential File Organization: Advantages and Disadvantages

Advantages	Disadvantages
Simplest organization	To get a particular record, most higher-level languages must access all the records prior to it—no random access by key[1]
Minimum overhead on disk	Interactive processing is awkward; operator must wait as the program searches for a record
Allows block spanning	You can add records only to the end of the file
Optimal if application accesses all records on each run	
Most versatile format: exchange data with systems other than VMS RMS; compatible with ANSI magnetic tape format	
No restrictions on the type of storage media; the file is portable	

[1] This restriction does not apply to disk sequential files with fixed-length record format. Records in such files can be stored and retrieved using random access by key, depending on language capabilities.

Table 2–3 (Cont.) Sequential File Organization: Advantages and Disadvantages

Advantages	Disadvantages
Random access by key available on fixed-format disk sequential files	

2.2.2 Relative File Organization

The relative file organization allows sequential and random access of records but is supported on disk devices only.

Note: **Although relative files are not supported for magnetic tape operations, magnetic tape can be used to transport relative files.**

In fact, relative files provide the fastest random access, and they require fewer tuning considerations.

A relative file consists of a series of fixed-length record positions (or cells) numbered consecutively from 1 to *n* that enables VMS RMS to calculate the record's physical position on the disk. The number, referred to as the *relative record number,* indicates the record cell's position relative to the beginning of the file.

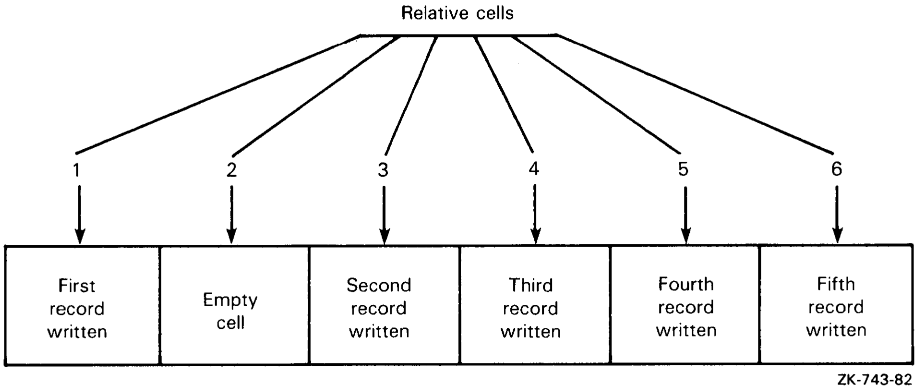
VMS RMS uses the relative record number as the key value to randomly access records in a relative file. The preferred method of tracking relative record numbers is to assign them based on some numeric field within the record; for example, the account number.

See Section 2.1.1.2 for a description of random access by key.

Each record in the file may be randomly assigned to a specific cell. For example, the first record may be assigned the first cell and the second record may be assigned the third cell, leaving the second cell empty. Unused cells and cells from which records have been deleted may be used to store new records.

Figure 2–10 illustrates the relative file organization.

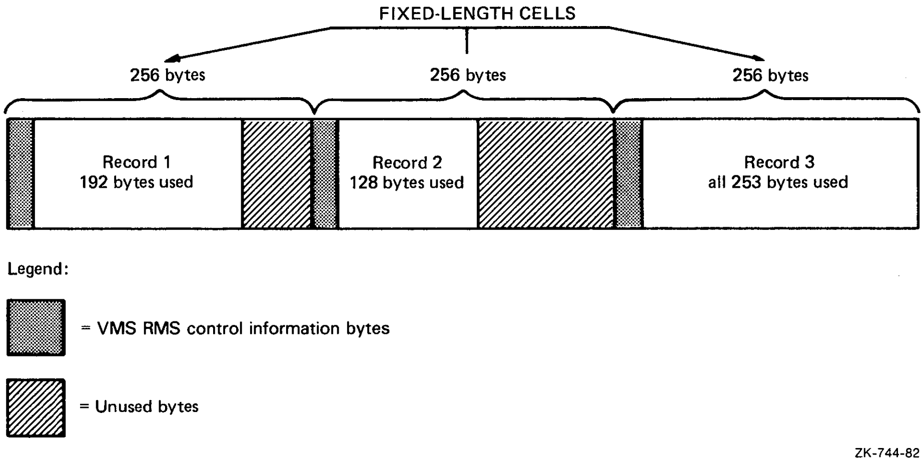
Figure 2–10 Relative File Organization



In a relative file, the actual length of the individual records may vary (that is, different size records can be in the same file) up to the limits imposed by the specified cell length. For example, think of a relative file configured as shown in Figure 2–11.

Note that because the records are variable-length records, each is prefixed by 3 bytes: the 2-byte count field (described in Section 2.1.2.2) and a 1-byte field that indicates whether or not the cell is empty (a delete flag). These bytes are used only by VMS RMS—you need not be concerned with them, except when planning the file's space requirements.

Figure 2–11 Variable-Length Records in Fixed-Length Cells



Some advantages and disadvantages of relative file organization are outlined in Table 2–4.

Table 2–4 Relative File Organization: Advantages and Disadvantages

Advantages	Disadvantages
Random access in all languages	Restricted to disk devices
Allows deletions	File contains a cell for each cell number between first and last record in file; limits data density
Allows random Get and Put operations	Program must know relative record number or RFA before it can randomly access the data; no generic access as in indexed file organization
Random and sequential access with low overhead	Interactive access can be awkward if you do not access records by relative record number
Can be write-shared	You can only insert records into unused record cells, but you can update existing records
	VMS RMS does not allow duplicate relative record numbers
	The space taken up by each record is as long as the maximum record size

2.2.3 Indexed File Organization

The indexed file organization allows sequential and random access of records but is supported on disk devices only.

Note: **Although relative files are not supported for magnetic tape operations, magnetic tape can be used to transport relative files.**

This type of file organization lets you store data records in an index structure ordered by the primary key and to retrieve data using index structures ordered by primary or alternate keys. The alternate index structures do not contain data records; instead, they contain pointers to the data records in the primary index.

For example, an indexed file may be ordered in ascending sort order by the primary key "employee number." However, you may want to set up additional (alternate) indexes for retrieving records from the file. Typically, you might set up an alternate index that is ordered in descending sort order by each employee's social security number.

Note: **The physical location of records in an indexed file is transparent to your program because VMS RMS controls record placement.**

In addition to the indexes, each indexed file includes a prolog structure that contains information about the file, including file attributes. VMS RMS currently supports three distinct prologs—Prolog 1, Prolog 2 and Prolog 3—but VMS RMS will normally create a Prolog 3 indexed file. However, you can specify a previous prolog version, typically for compatibility with RMS–11.

2.2.3.1 Sequentially Retrieving Indexed Records

To sequentially retrieve indexed records, your program must specify the key for the first access. VMS RMS then uses the index for that key to retrieve successive records. For example, assume there is an index file with three records, having primary keys of A, B, and C, respectively. To retrieve these records sequentially in ascending sort order, your program must provide the key *A* on the first access; VMS RMS will access the next two records without further key inputs from your program.

To randomly retrieve records in an index file, your program must provide the appropriate key value for each access. Now assume an index file with three records having primary keys A, B, and C that will be retrieved in C, A, B order. On the first access, your program must provide the key *C*, on the next access the key *A*, and on the final access the key *B*.

2.2.3.2 Index Keys

In an indexed file, each record includes one or more key fields (or simply keys) that VMS RMS uses to build related indexes. Each key is identified by its location within the record, its length, and whether it is a simple key or a segmented key.

A simple key may be any one of the following data types:

- A single contiguous character string
- A packed decimal number
- A 2-, 4-, or 8-byte unsigned binary number
- A 2-, 4-, or 8-byte signed integer

Note: RMS-11 cannot process 8-byte numeric keys.

Segmented keys are fields of character strings having from 2 to 8 segments which may be or may not be contiguous; however, VMS RMS treats all key segments as a logically contiguous string.

For an indexed file, you must define at least one key, the primary key, and you may optionally define one or more alternate keys. VMS RMS uses alternate keys to build indexes that identify records in alternate sort orders. As with the primary key, each alternate key is defined by its location and length within the record.

2.2.3.3 Other Key Characteristics

In addition to defining keys, you can specify various key characteristics (FDL secondary key attributes) including the following:

Duplicate keys — This characteristic permits you to use the key value in more than one record. However, only the first record having the key value can be accessed randomly; other records having the same key value can only be accessed sequentially.

Changeable keys — This characteristic applies to alternate keys only. When you specify *changeable* alternate keys, the *alternate* keys in a record can be changed when the record is updated. When an alternate key value is changed, VMS RMS automatically adjusts the appropriate index to reflect the new key value.

Null keys — This characteristic applies to alternate keys only. When you fill an alternate key field with null characters, VMS RMS does not insert the record in the related index.

Note: **VMS RMS excludes from the related index any record that is not long enough to contain a complete alternate key.**

Key characteristics can be defined separately for each key.

When you do not allow duplicate key values, VMS RMS rejects any attempt to put a record into a file if it contains a key value that duplicates a key value already present in another record. Similarly, when alternate key values cannot be changed, VMS RMS does not allow your program to update a record by changing the alternate key value. If you disallow a null value for a key, VMS RMS inserts an entry for the record in the associated alternate index.

Figure 2–13 illustrates the general structure of an indexed file in which only the primary key is defined. The primary key is the names of employees in an employment record file. Figure 2–12 illustrates the general structure of an indexed file in which the primary key and one alternate key are defined. The primary key is the names of employees, and the alternate key is the badge numbers of employees in an employment record file.

2.2.3.4 Specifying Sort Order

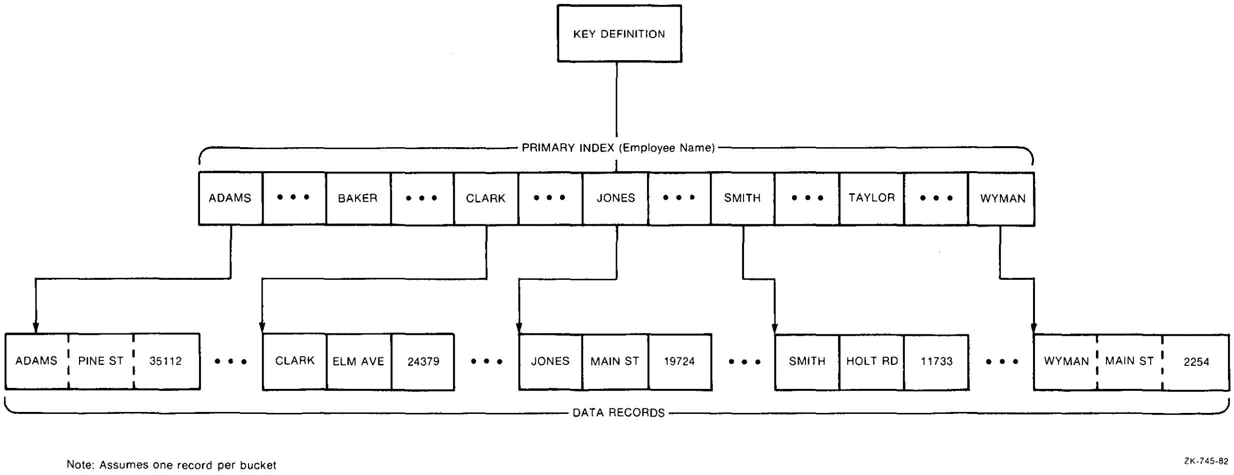
VMS RMS lets you specify either ascending sort order or descending sort order for each key. At the VAX MACRO level, you encode sort order within the key data type field (XAB$B_DTP) of the associated key XAB; you use the attribute KEY TYPE at the FDL level. For example, if you want to build an index of string data type keys in ascending sort order using VAX MACRO, you enter the following line in the associated key XAB:

```
DTP = STG
```

To build an index of string data type keys in descending sort order, you enter this line in the associated key XAB:

```
DTP = DSTG
```

Figure 2–12 Single-Key Indexed File Organization



See the *VMS Record Management Services Manual* for a complete listing of key data types used to specify ascending and descending sort order.

2.2.3.5 Using Collated Keys

The VMS RMS multinational key feature lets you assign alternative (non-ASCII) collating sequences to a key. For example, a program can sort records using a key that accesses a collating sequence based on French or alternatively accesses a collating sequence based on Spanish.

The basis for this feature is the National Character Set Utility (NCS). When an application program creates an index file with an alternative collating sequence, it calls NCS. NCS responds by retrieving the collating sequence from the NCS library, storing it in local memory and providing the calling program with a pointer to it. In addition to naming the collating sequence, the calling program must provide NCS with a location for storing the pointer (CS_ID) to the memory location of the collating sequence. (For information about NCS, see the *VMS National Character Set Utility Manual*.)

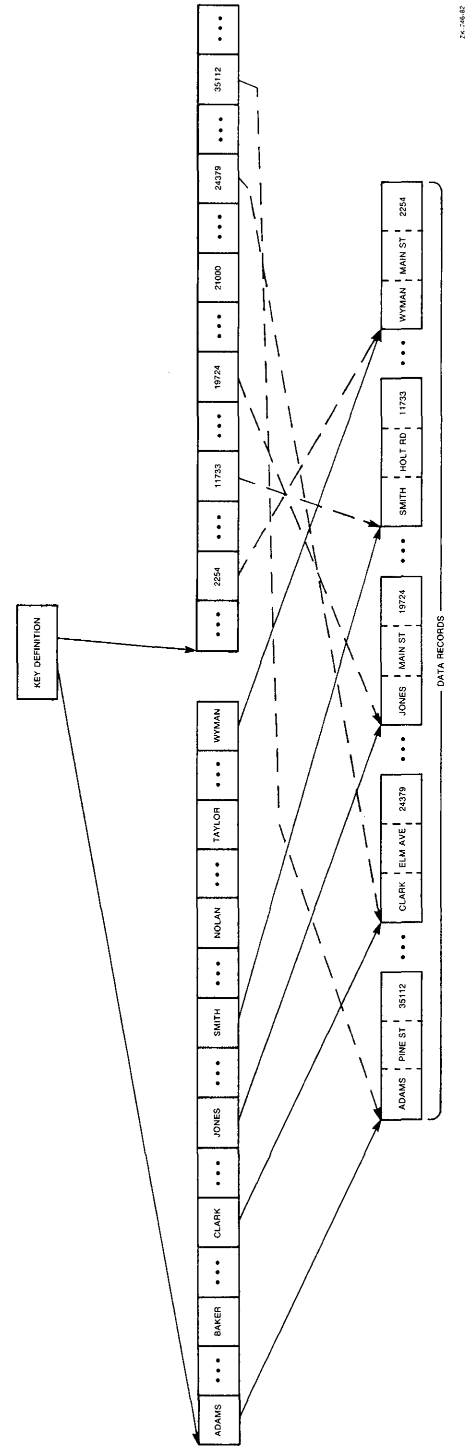
When the application program creates the data file, it uses the pointer to copy the collating sequence from local memory into the data file's prolog space. A collating sequence is typically one block long.

The application program may specify a collated key from either the RMS interface or the FDL interface.

From the RMS interface, the application program identifies the collating sequence using an appropriate string descriptor and includes a symbolic reference to the location of the pointer. As with all other keys, the application program may specify either ascending or descending sort order. From the RMS interface, you specify the key data type COL for an ascending sort order or the key data type DCOL for descending sort order.

From FDL, you specify a collated key by selecting the one of the collated key data types (collated for ascending sort order, decollated for descending sort order) from the INDEXED file script. FDL responds by prompting for the name of the collating sequence. If you enter an invalid collating sequence,

Figure 2–13 Multiple-Key Indexed File Organization



any attempt to use the FDL file for creating a data file will be unsuccessful and NCS generates the following error message:

```
%NCS-F-NOT_CS, name or id is not a CS
```

Example 2–1 illustrates the use of collating keys in a MACRO-32 program segment.

Example 2–1 Creating a File Containing Collated Keys

```
   .
   .
   .
        .TITLE Example
;
; Define key type as COL or DCOL
;
KEY0:   $XABKEY
   .
   .
   .
               DTP=COL
;
; Descriptor for collating sequence name
;
CS_DESC:       .ASCID /Spanish/
               .EXTRN NCS$GET_CS
   .
   .
   .
; Collating sequence name descriptor
;
               PUSHAL   CS_DESC
;
; Where to store address of collating sequence
;
               PUSHAL   KEY0+XAB$L_COLTBL
;
; Fetch collating sequence
;
               CALLS    #2,G^NCS$GET_CS
               BLBC     R0,ERROR
;
; Create file
;
               $CREATE  FAB=OUTFAB
               BLBC     R0,ERROR
```

2.2.3.6 Summary of Indexed File Organization

Some advantages and disadvantages of the indexed file organization are outlined in Table 2–5.

Table 2–5 Indexed File Organization: Advantages and Disadvantages

Advantages	Disadvantages
Most flexible random access: by any one of multiple keys or RFA; key access by generic or approximate value	Highest overhead on disk and in memory
Duplicate key values possible	Restricted to disk
Automatic sort of records by primary and alternate keys; available during sequential access	Most complex programming
Record location is transparent to user	Longest record access times
Potential range of key values not physically present as in relative file organization	
Variety of data formats for keys	
Transparent data compression	

3 Performance Considerations

When you design a file, your decisions regarding record access mode, record format, and file organization should be aimed at achieving optimum data processing performance for your application. This chapter discusses general performance considerations and specific trade-offs you can make in the design of your data files. In Sections 3.3, 3.4, and 3.5, these trade-offs are discussed in the contexts of the three file organizations, sequential, relative, and indexed.

3.1 Design Considerations

In designing files for optimum data processing performance, you should emphasize the following performance factors:

- Speed—You want to increase the speed with which your program processes data.
- Space—You want to decrease the space required to store data on disk and to process data in memory.
- Shared access—You want your data to be accessible to other users who are authorized access to it.
- Impact on application design—You want to minimize the application design effort.

3.1.1 Speed

The first guideline you can apply to the design process is to decrease the amount of program I/O time.

Storing data on, and retrieving date from, mass storage devices is the most time-consuming VMS RMS operation. For example, when an application needs data, the disk controller must first search for the data on the disk. The disk controller must then transfer the data from the disk to main memory. After processing the data, the program must provide for returning the results to mass storage via the I/O subsystem.

One way to reduce I/O time is to have the data in memory so that you can minimize search and transfer operations. If data must be transferred to memory for processing, you should consider design variables that reduce transfer time.

The first variable you might consider is the set of *file attributes* that may affect I/O time:

- Initial file allocation
- Default extension quantity
- Bucket size (for a relative or indexed file)
- Number of keys (for an indexed file)

- Number of duplicate key values (for an indexed file)

The second variable is the *file size* as measured by the number of records in the file. File size affects the time it takes to scan a file sequentially or to access records using an index.

A third variable is the *storage device* on which your program and data files reside. Crucial to I/O performance are the type of device chosen (moving-head, fixed-head, and so on) and the amount of I/O activity for that device within the system.

To make your applications run faster, consider the following:

- Keep as much data in memory as possible, but be wary of any significant increase in the page fault rate.
- Minimize the number of I/O transfers by transferring larger portions of data.
- Arrange your data on the disk to minimize disk head motion.

3.1.2 Space

When you run your application, you need space to buffer data in memory. You can reduce data processing time by increasing the size of the I/O buffers VMS RMS uses; however, avoid exceeding the space limitations imposed by the working set.

In addition to the data buffers themselves, the space required to store data can vary depending on the file organization you choose.

For example, sequential file organization requires VMS RMS to add an empty byte to a record when the record has an odd number of bytes but must be aligned on an even-numbered byte boundary. At the record level, you should consider the added space required to prefix a two-byte count field to each variable-length record.

For the relative file organization, VMS RMS constructs a series of record storage cells based on the maximum length of the records. The record cells are 1 byte longer than the size of fixed-length records or 3 bytes longer than the maximum size specified for variable-length records.

For the indexed sequential file organization, VMS RMS must add the following informational components to your data files:

- An index for each defined key
- 15 bytes of formatting information for each bucket
- A seven-byte header for each record
- A count field for each variable-length record
- Other overhead of varying lengths that is needed by VMS RMS to move files and to delete records. You should keep the size of records to the minimum required for your application.

You should also consider the effects of compression on the size of your indexed files. You can compress keys in data buckets and in index buckets, and you can compress data in the primary buckets. If you use key, index, or data compression, the file requires less space on the disk, and each I/O buffer can hold more information. Compression may even eliminate one index level thereby reducing the number of disk transfers needed for random access.

Note: **You cannot use key compression or index compression with the collated key data type.**

Random access of compressed files requires slightly more CPU time, but this is usually offset by the improved performance you achieve with fewer index levels.

3.1.3 Shared Access

A file management technique that allows more than one user to simultaneously access a file or a group of files is called *shared access* or *file sharing*. When you try to adjust the performance of shared files, you need to pay particular attention to record locking options and the use of global buffers. Avoid assigning sharing attributes to files that are not actually shared.

There are essentially three sharing conditions: no sharing, sharing without interlocking, and sharing with interlocking. Chapter 7 discusses each of these in detail.

3.1.4 Impact on Applications Design

The impact on applications design increases as file design complexity increases. That is, your application programs require more design effort for processing indexed files than for processing sequential files. The primary consideration here should be to evaluate whether the benefits derived by having direct access to records is worth the added cost of the application program design needed to interface with the file management system.

3.2 Tuning

The process of designing your files to achieve better processing performance is called *tuning*.

Tuning requires you to make a number of trade-offs and design decisions. For example, if a process had sole access to the processor, it could keep all of its data in memory and tuning would be unnecessary; but this situation is unlikely. Instead, several processes are usually running simultaneously and are competing for the memory resource. If all processes demand large amounts of memory, the system responds by paging and swapping, which slows down system performance.

The way you intend to use your programs and data files can determine some of the basic tuning decisions. For example, if you know that three files are accessed 80 percent of the time, you might consider locating the files in a common area on the disk to speed up access to them. The performance of programs that use the other files is slower, but the system as a whole runs faster.

In tuning your file management system, you implement these trade-offs and design decisions by specifying file design attributes together with various file-processing options and record-processing options.

3.2.1 File Design Attributes

The following file design attributes control how the file is arranged on the disk and how much of the file is transferred to main memory when needed. These file design attributes generally apply to all three types of file organization; other file design attributes that specifically pertain to the various file organizations are described under the appropriate heading.

- Initial file allocation
- Contiguity
- File extend quantity
- Units of I/O
- The use of multiple areas (for indexed files)
- Bucket fill factor (for indexed files)

The following sections discuss how each file design attribute can maximize efficiency.

3.2.1.1 Initial File Allocation

When you create a file, you should allocate enough space to store it in one contiguous section of the disk. If the file is contiguous on the disk, it requires only one retrieval pointer in the header; this reduces disk head motion.

You should also consider allocating additional space in anticipation of file growth to reduce the number of required extensions.

You can allocate space either by using the FDL attribute FILE ALLOCATION or by using the file access control block field FAB$L_ALQ.

3.2.1.2 Contiguity

Use the FILE secondary attribute CONTIGUOUS to arrange the file contiguously on the disk, if you have sufficient space. If you assign the CONTIGUOUS attribute and there is not enough contiguous space on the disk, VMS RMS does not create the file. To avoid this, consider using the FDL attribute BEST_TRY_CONTIGUOUS instead of the CONTIGUOUS attribute. The BEST_TRY_CONTIGUOUS attribute arranges the file contiguously on the disk if there is sufficient space or noncontiguously if the space is not available for a contiguous file.

You can make this choice by accepting the FDL default values for both attributes—NO for CONTIGUOUS, YES for BEST_TRY_CONTIGUOUS or by taking the VMS RMS option FAB$V_CBT in the FAB$L_FOP field.

3.2.1.3 Extending a File

VMS RMS automatically extends files when more space is needed than was originally allocated. You may specify a default extension value (in blocks) or you may have VMS RMS automatically specify the extension value.

If you expect to add large amounts of data to a file over a relatively short time period, you should consider specifying an extension value that is large enough to minimize file fragmentation and to reduce the total overhead consumed by EXTEND operations. As a file becomes fragmented, access time becomes greater and processing performance degrades. Similarly, added overhead due to VMS RMS having to extend files can degrade performance.

Conversely, if you only add small amounts of data to the file over a relatively long time period, specifying a large extension value results in wasted disk space.

If you do not set a default extension quantity, VMS RMS automatically chooses an extension size. It does this by using an algorithm that allocates a minimal extension value and writes the data to the file. When a file becomes filled, this approach can result in severe file fragmentation.

You can specify the default extension value in the following ways:

- When you create the file, you can have the FDL editor (EDIT/FDL) calculate the value using your responses to the script questions. This is the recommended method.

  The FDL editor assigns the value using the FILE EXTENSION attribute. For index files with multiple areas, the FDL editor assigns a separate value to each area using the specified AREA EXTENSION attributes.

- If you choose not to use FDL to establish the extension value when you create the file, you can specify the value directly in the FAB field FAB$W_DEQ if you are using a low-level language. For index files with multiple areas, you can assign separate values to each area using appropriate XAB$B_AID fields and related XAB$W_DEQ fields. In this case, you must determine the value using the average record size, the number of records to be added to the file during a given period of time, the number of records per bucket, and the bucket size.

- If you elect not to assign an extension value using FDL or by using FABs and XABs directly, you can establish the extension value at run time using the DCL command SET FILE/EXTENSION = n, where *n* is the default extension size for the volume.

- If you decide not to use any of the previously described methods, you can use the value established by the DCL command SET RMS_DEFAULT /EXTEND_QUANTITY = n, where *n* is the extension value.

  The SET RMS_DEFAULT command applies the VMS RMS default extension value to files for your process *only*, unless you use the /SYSTEM qualifier with the command. Note that you need the CMKRNL privilege to use the /SYSTEM qualifier.

Note: The value assigned to the default extension size by any other method overrides the value assigned by the SET RMS_DEFAULT command.

For a description of how EDIT/FDL allocates file space, see Appendix A.

3.2.1.4 Units of I/O

Another file design consideration is to reduce the number of times that VMS RMS must transfer data from disk to memory by making the I/O units as large as possible. Each time VMS RMS fetches data from the disk, it stores the data in an I/O memory buffer whose capacity is equal to the size of one I/O unit. A larger I/O unit makes more records immediately accessible to your program from the I/O buffers.

In general, using larger units of I/O for disk transfers improves performance, as long as the data does not have to be swapped out too frequently. However, as the total space used for I/O buffers in the system approaches a point that results in excessive paging and swapping, increasing I/O unit size degrades system performance.

VMS RMS performs I/O operations using one of the following I/O unit types:

- Blocks
- Multiblocks
- Buckets

A *block* is the basic unit of disk I/O and it consists of 512 contiguous bytes. Even if your program requests less than a block of data, VMS RMS automatically transfers an entire block.

The other I/O units—multiblocks and buckets—are made up of block multiples. A *multiblock* is an I/O unit that includes up to 127 blocks but whose use is restricted to sequential files. See Section 3.3.2 for details. A *bucket* is the I/O unit for relative and indexed files and it may include up to 63 blocks. See Section 3.4 and Section 3.5 for details.

3.2.1.5 Multiple Areas for Indexed Files

For indexed files, another design strategy is to separate the file into multiple *areas*. Each area has its own extension size, initial allocation size, contiguity options, and bucket size. You can minimize access times by precisely positioning each area on a specific disk volume, cylinder, or block.

For instance, you can place the data area on one volume of a volume set and place the indexed area on another volume of the volume set. If your application is I/O bound, this strategy could increase its throughput. You can also ensure that your data buckets are contiguous for sequential access by allocating extra space to the data area for future extensions.

3.2.1.6 Bucket Fill Factor for Indexed Files

Any attempt to insert a record into a filled bucket results in a *bucket split*. When a bucket split occurs, VMS RMS tries to keep half of the records (including the new record if applicable) in the original bucket and moves the remaining records to a newly created bucket.

Excessive bucket splitting can degrade system performance through wasted space, excessive processing overhead, and file fragmentation. For example, each record that moves to a new bucket must maintain a forward pointer in the original bucket. The forward pointer indicates the record's new location. At the new bucket, the record must maintain a backward pointer to its original bucket. VMS RMS uses the backward pointer to update the forward pointer in the original bucket if the record later moves to yet another bucket.

When a program attempts to access a record by alternate key or by RFA, it must first go to the bucket where the record originally resided, read the pointer to the record's current bucket residence, and then access the record.

To avoid bucket splits, you should permit buckets to be only partially filled when records are initially loaded. This provides your application with space to make additional random inserts without overfilling the affected bucket.

Section 3.5.2.2 describes fill factors in more detail.

3.2.2 Processing Options

Five processing options can be used to improve I/O operations: two file-processing options and three record-processing options. The file-processing options include the deferred-write option and the global buffer option. The global buffer option may be used with all three file organizations, but the deferred-write option is restricted to use with relative and indexed files.

The record-processing options include the multiple buffer option, the read-ahead option and the write-behind option. The multiple buffer option may be used with all three file organizations, but the read-ahead option and the write-behind option may be used only with sequential files.

This section summarizes the options. Section 3.3 through Section 3.5 describe the options in the context of tuning files. For additional information about buffering, see Chapter 7.

3.2.2.1 Multiple Buffers

When a program accesses a data file, it transfers the file from disk into memory using I/O units of blocks, multiblocks, or buckets. The I/O units are subsequently placed in memory *I/O buffers* sized to be compatible with the I/O units.

If you do not have enough buffers, excessive I/O transfers may degrade the performance of your application. On the other hand, if you have too many buffers, performance may degrade because of an overly large working set. As a rule, however, increasing the size and number of buffers can improve performance if the data read into the buffers will soon be processed and if your working set can efficiently maintain the buffers. In fact, changing the size and number of buffers is the quickest way to improve the performance of your application when you are processing localized data.

The optimum number of buffers depends on the organization and use of your data files. The recommended way to determine the optimum number of buffers for your application is to use the Edit/FDL Utility.

The number of I/O buffers is a run-time parameter you set with the FDL editor by adding the CONNECT secondary attribute MULTIBUFFER_COUNT to the definition file. (see Chapter 9. With a low-level language, you can set the value directly into the RAB$B_MBF field of the record access block.

Alternatively, the number of buffers may be specified for a process using the DCL command SET RMS_DEFAULT/BUFFER_COUNT=n, where the variable *n* represents the desired number of buffers. With this command, you may set distinct values for your sequential, relative, and indexed files using the appropriate file organization qualifier. If you omit the file organization qualifier, sequential organization is assumed. To examine the current settings for the process and system default multibuffer count, use the DCL command SHOW RMS_DEFAULT.

If none of the above methods is used, VMS RMS uses the system-wide default value established by the system manager. If the system-wide default is omitted or is set to 0, VMS RMS uses a value of 1 for sequential and relative files and a value of 2 for indexed files.

For more details about using multiple buffers with sequential files, see Section 3.3.3. For more details about using multiple buffers with relative files, see Section 3.4.2. For more details about using multiple buffers with indexed files, see Section 3.5.2.3.

Chapter 7 describes the use of multiple buffers in the context of shared files.

3.2.2.2 Deferred-Write Processing

One way to improve performance through minimized I/O is to use the deferred-write option to keep data in memory as long as practicable. However, you must determine if this added performance benefit is worth the increased risk of losing data if the system crashes before a buffer is transferred to disk.

With indexed files and relative files, you may use the deferred-write option to defer writing modified buckets to disk until the buffer is needed for another purpose or until the file is closed.

Typically, the largest gains in performance come from using the deferred-write option with sequential access. Retrieving and modifying records one after the other permits you to access all of the records from one bucket while the bucket is in memory.

You may also improve performance by using the deferred-write option to prevent VMS RMS from writing a shared sequential file to disk on each modification. However, this increases the risk of losing data if the system crashes before the full buffer is transferred to disk.

Note that nonshared sequential files behave as if the deferred-write option is always specified, because a buffer is only written to disk after it is completely filled.

Deferred-write processing is a default run-time option for some high-level languages and can be specified by using clauses in other languages. You can activate this option through FDL by adding the CONNECT secondary attribute DEFERRED_WRITE. From a low-level language, you can activate the deferred-write option by setting the FAB$V_DFW bit in FAB$L_FOP field.

3.2.2.3 Global Buffers

If several processes are to share a file, you may want to provide the file with *global buffers*—I/O buffers that two or more processes can access. With global buffers, processes may access file information without allocating dedicated buffers. If you do not allocate dedicated buffers, you can conserve buffer space and buffer management overhead. You gain this benefit at the cost of additional system resources, as described in the *VMS Record Management Services Manual.*

When you create a file, you can assign the desired number of global buffers by using the FDL editor to set the value for the FILE secondary attribute GLOBAL_BUFFER_COUNT. From a low-level language, you can optionally set the value directly into the FAB$W_GBC field. Alternatively, you may use the DCL command SET FILE/GLOBAL_BUFFERS to set the global buffer count.

Global buffers are not used directly to retain modified information when the deferred-write option is enabled. If a global buffer is modified and the deferred-write option is enabled, the contents of the global buffer are copied to a process local buffer before other processes are allowed to access the global buffer contents. Subsequent use of the modified buffer by the process that deferred the writeback refer to the process local buffer while it remains in the process local cache. Reference to the global buffer by another process causes the contents of the process local buffer to be written back to disk.

If a global buffer is modified and the deferred-write option is not enabled, then the contents are written out to disk from the global buffer. Therefore, using global buffers along with the deferred-write option may cause a slight increase in processing overhead if in fact no further references to the modified buffer occur before it is flushed from the cache anyway. For that reason, you may want to disable the deferred-write option for processes that do not reaccess buffers after records have been written to them.

Sections 3.3 through 3.5 discuss the use of global buffers in tuning the various file types.

3.2.2.4 Read-Ahead and Write-Behind Processing

The operation of sequentially organized files can be improved by implementing read-ahead and write-behind processing. These features improve performance by permitting record processing and I/O operation to occur simultaneously. The read-ahead and write-behind features are default run-time attributes in some languages, but they must be explicitly specified in others.

You implement read-ahead and write-behind processing by using two buffers. The processing program uses one buffer and the I/O subsystem uses the other. In read-ahead processing, the program reads data from one buffer as the second buffer inputs data from the disk. In write-behind processing, one buffer accepts output data from the program, while the second buffer outputs program data to disk.

The next section provides additional information about read-ahead and write-behind processing.

3.3 Tuning a Sequential File

Sequential files consist of a file header and a series of data records. Records are stored in the order in which they are written to the file, which may use one of the following record formats:

- Fixed-length record format
- Variable-length record format
- VFC record format
- Stream record format

- Undefined record format

If records in the fixed-length record format cross block boundaries (using the block spanning option), the maximum record length for the fixed-length record format in a sequential file is 32,767 bytes; otherwise, the maximum record length is limited to 512 bytes (one block).

Although variable-length records can be different lengths, you may specify the maximum length for a file. Each variable-length record begins with a two-byte count field that VMS RMS uses when it retrieves the record. As with the fixed-length format, the maximum record length is 32,767 bytes if records are permitted to cross block boundaries. Otherwise, the maximum length is limited to 512 bytes, including the two-byte count field.

VFC records include a fixed-length control field (having up to 255 bytes) and a variable-length data area. The fixed-length control field can be used for any purpose, but it is most commonly used to store information such as line numbers and carriage returns. The maximum size of a VFC record with block spanning (see Section 3.3.1) is 32,767 bytes minus the number of bytes in the fixed-length control field.

There are three variations of the stream record format:

STREAM_CR	This variation uses a carriage return as the terminator.
STREAM_LF	This variation uses a line feed as the terminator.
STREAM	This variation uses a terminator from a limited set of special characters: the carriage return (CR); the carriage-return/line-feed combination (CR/LF); or the form feed (FF).

All three variations feature a continuous stream of bytes but are different in the way records are delimited. Records are limited to 32,767 bytes in length for all variations of the stream record format.

When the record format is undefined, records are processed as a continuous stream of bytes with no specified terminator.

The following sections provide guidelines for improving the performance of sequential file processing using various tuning options.

3.3.1 Block Spanning Option

You should always specify that records in a sequential file are permitted to span blocks, that is, cross block boundaries. In this way, VMS RMS can pack the records efficiently and avoid wasted space at the end of a block.

By default, the FDL editor activates *block spanning* for files organized *sequentially* by setting the RECORD secondary attribute BLOCK_SPAN to YES. If you are using a low-level language, you activate the block spanning option directly in the FAB by resetting the FAB$V_BLK bit in the FAB$L_RAT field.

3.3.2 Multiblock Size Option

A *multiblock* is an I/O unit that includes up to 127 blocks but can be used only with sequential files. When a program instructs VMS RMS to fetch data within a multiblock, the entire multiblock is copied from disk to memory.

You specify the number of blocks in a multiblock using the *multiblock count*, a run-time option. If you are using the FDL editor, specify the multiblock count option using the secondary CONNECT attribute, MULTIBLOCK_COUNT. From a lower-level language, you may set the value into the RAB$B_MBC field, directly. Another alternative is to establish the count using a DCL command of the following form:

```
SET RMS_DEFAULT/BLOCK_COUNT=n
```

The variable *n* represents the specified number of blocks. Here, the specified multiblock count is limited to your process unless you specify the /SYSTEM qualifier.

In most cases, the largest practical multiblock value to specify is the number of blocks in one track of the disk, a number that varies with the various types of disks. (See the *VMS I/O User's Reference Volume* for the track sizes in blocks of DIGITAL-supported disks). However, the most efficient number of blocks for your application may be more or less than the number of blocks in a track. You should try various sizes of multiblocks until you find the optimum value.

3.3.3 Number Of Buffers

For sequential files, you can specify the number of buffers at run time. From FDL, you can set the number of buffers with the secondary CONNECT attribute MULTIBUFFER_COUNT. From a low level language you can set the value directly into the RAB$B_MBF field in the RAB. From the DCL interface, you can establish the number of buffers using a DCL command in the following form:

```
SET RMS_DEFAULT/SEQUENTIAL/DISK/BUFFER_COUNT=n
```

The variable *n* represents the number of buffers.

In simple operations with sequential files, one I/O buffer is sufficient. Increasing the number of buffers takes space in the process working set and could degrade performance.

With nonshared sequential files, particularly if you want to perform sequential access, you can use read-ahead and write-behind processing. With this type of processing, a buffer contains the next record to be read or written to the disk while a separate buffer completes the current I/O operation.

The length of the buffers used for sequential files is determined by the specified multiblock count. The optimal number of blocks per buffer depends on the record size for sequential access to a sequential file, but a value such as 16 may be appropriate.

3.3.4 Global Buffer Option

If a file is shareable, you may want to allocate it global buffers. A *global buffer* is an I/O buffer that two or more processes can access. If two or more processes are requesting the same information from a file, I/O can be minimized because the data is already in the global buffer. This is especially true for program sequences in which all of the processes are reading data.

Note that VMS RMS also provides each process with local I/O buffers to attain efficient buffering capacity.

3.3.5 Read-Ahead and Write-Behind Options

Specifying the read-ahead and write-behind options for sequential files can improve performance. The read-ahead and write-behind options require at least two I/O buffers and the *multibuffer* attribute. Note that using more than two I/O buffers usually does not improve performance. (See Section 3.3.3.)

Most languages incorporate the read-ahead and write-behind options by default. With some languages, you must specify the read-ahead and write-behind options explicitly using a clause in the language. If a language does not have a clause for specifying the read-ahead and write-behind options, you must use a VAX MACRO routine to select these options when you open the file.

At the VAX MACRO level, you can select these options by setting the RAB$V_RAH bit in the RAB$L_ROP field for read-ahead processing and the RAB$V_WBH bit for write-behind processing prior to requesting the Connect service.

You can also use FDL to select these options by using the secondary CONNECT attributes READ_AHEAD and WRITE_BEHIND respectively.

3.4 Tuning a Relative File

A relative file consists of a file header, file attributes, a prolog, and a series of fixed-length cells. Each cell contains one record that includes a deleted-record byte followed by the data portion of the record, which may or may not be blank.

VMS RMS assigns each cell a sequential number, called the relative record number, that can be used to randomly access the record.

A relative file can contain fixed-length records, variable-length records, or VFC records. Fixed-length records are particularly useful in relative files because of the fixed cell size.

The maximum size for fixed-length records in a relative file is 32,255 bytes. For variable records the maximum size is 32,253 bytes. The maximum size for VFC records is 32,253 bytes minus the size of the fixed-length control field, which may be up to 255 bytes long.

3.4.1 Bucket Size

With relative files, buckets are used as the unit of transfer between the disk and memory. You specify bucket size when you create the file, but you can change the size later by converting the file (see Chapter 10.) You can specify the bucket size using the FDL FILE secondary attribute BUCKET_SIZE or by inserting the value directly into the VMS RMS control block fields FAB$B_BKS and XAB$B_BKZ. Although the size can be as large as 63 blocks, a bucket size larger than one disk track usually does not improve performance.

If you choose to select the bucket size, you should also consider how your application accesses the file. For random access, you may want to choose a small bucket size; for sequential access, a large bucket size; and for mixed access, a medium bucket size.

One way to improve performance for a relative file is to align the file on a cylinder boundary and specify the size of one disk track as the bucket size. However, this requires that you can perform an exact alignment on the file.

If you use the FDL editor to establish the bucket size (this is recommended), the editor fixes the size at the optimum value based on your script inputs.

If you intend to access the file randomly, EDIT/FDL sets the bucket size equal to four records because it assumes that four records are a reasonable amount of data for a random access. If you intend to access records sequentially, EDIT/FDL sets the bucket size equal to 16 records because it assumes that 16 records is a reasonable amount of data for one sequential access.

If you find that your application needs more data per access, then use the EDIT/FDL command MODIFY to change the assigned values.

3.4.2 Number of Buffers

The multibuffer count is a run-time option that you can set with the DCL command SET RMS_DEFAULT/RELATIVE/BUFFER_COUNT=n, the FDL attribute CONNECT MULTIBUFFER_COUNT, or the VMS RMS control block field RAB$B_MBF. The type of record access determines the best use of buffers.

The two extremes of record access are when records are processed either completely randomly or completely sequentially. Also, there are cases in which records are accessed randomly but may be reaccessed (random with temporal locality) and cases where records are accessed randomly but adjacent records are likely to be accessed (random with spatial locality).

In completely sequential processing, the first record may be located randomly and the following records accessed sequentially (records are usually not referenced more than once). For best performance, you should specify one buffer with a large bucket size unless you use the read-ahead option, which requires two buffers.

Large buckets hold more records, so you can access a greater number of records before performing I/O operations. However, a small multibuffer count, such as the default of 1 buffer, is sufficient.

When you want to improve sequential access performance, you may get better results by tuning the bucket size rather than changing the number of buffers.

Completely random processing means that records are not accessed again, and adjacent records are not likely to be accessed. You should use one buffer with a minimal bucket size. You do not need to build a memory cache because the records are likely to be scattered throughout the file. New requests for records most likely result in an I/O operation, and caching extra buckets wastes space in your working set.

In random with temporal locality processing (reaccessed records), records are processed randomly, but the same records may be accessed again. You should use multiple small buffers to cache records that are to be reaccessed. The bucket size can be small for this type of access because the records near the record currently accessed are not likely to be accessed. Caching reaccessed records in large buckets wastes space in memory. Multiple buffers allow the previously accessed records to remain in memory for subsequent access.

In random with spatial locality processing (adjacent records), records are processed randomly, but the next or previous record has a good chance of being accessed. You should use a large buffer and bucket size to improve the probability that the next record to be processed is in the same bucket as the record most recently processed. One or two buffers should be sufficient.

If you process your data file with a combination of these patterns, you should compromise between the processing strategies. An application illustrating both temporal and spatial access uses the first record in the file as a pointer to the last record accessed. The program reads the first record to find the location of the next record to be processed, processes the record, and updates the pointer in the first record. Because the application accesses the first record frequently, the access pattern exhibits temporal locality, but because it adds records sequentially to the end of the file, the access pattern also exhibits spatial locality.

When you add records to a relative file, you might consider choosing the deferred write option (FDL attribute FILE DEFERRED_WRITE, FAB$L_FOP field FAB$V_DFW). With this option, the contents of the write buffer are not transferred from memory to disk until the buffer is needed for another purpose or until the file is closed. Note however, that the possibility of losing data during a system crash increases when you use the deferred write option.

To see what the current default buffer count is, give the DCL command SHOW RMS_DEFAULT. To set the default buffer count, use the DCL command SET RMS_DEFAULT/RELATIVE/BUFFER_COUNT=n, where *n* is the number of buffers.

3.4.3 Global Buffer Option

If several processes share a relative file, you may want to specify that the file use the global buffer option. A *global buffer* is an I/O buffer that two or more processes can access. If two or more processes simultaneously request the same information from a file, each process can use the global buffers instead of allocating its own dedicated buffers. Only one copy of the buffers resides at any time in memory, although the buffers are charged against each process's working set size.

Using the global buffer option to form a memory cache may not reduce the number of I/O operations necessary to process the file, in all cases. Regardless of how many global buffers you allocate, VMS RMS always allocates one I/O buffer per process, which provides efficient buffering capacity.

If your application has several processes sharing the file and accessing the same records in a transaction sequence, then you may benefit from allocating enough global buffers to cache these shared records.

3.4.4 Deferred-Write Option

The deferred-write option is a run-time option that can improve performance. It is the default operation for some languages and can be specified by clauses in other languages. If there is no language support, you can use a VAX MACRO subroutine to set the FAB$V_DFW bit in the FAB$L_FOP field before opening the file.

When you select the deferred-write option, VMS RMS delays writing a modified bucket to disk until the buffer is needed for another purpose or until another process needs to use the bucket. This delay improves performance because it reduces the number of disk I/O operations. You achieve the largest performance gains using the deferred-write option with sequential access file operations.

For example, in a relative file with 100-byte records and 2-block buckets, 10 records fit in one bucket. Without the deferred-write option, writing records 1 through 10 in order results in eleven I/O operations—one for the initial file access and one for each of the records.

With the deferred-write option, you need only two I/O operations—one for the initial file access and one to write the bucket.

A larger cache might be useful in situations in which the accesses are not strictly sequential but follow some local pattern.

3.5 Tuning an Indexed File

This section discusses the structure of indexed files and ways to optimize their performance.

3.5.1 File Structure

An indexed file consists of a file header, a prolog, and one or more index structures. The primary index structure contains the data records. If the file has alternate keys, it has an alternate index structure for each alternate key. The alternate index structures contain secondary index data records (SIDRs) that provide pointers to the data records in the primary index structure. The index structures also contain the values of the keys by which VMS RMS accesses the records in the indexed file.

3.5.1.1 Prologs

VMS RMS places information concerning file attributes, key descriptors, and area descriptors in the prolog. You can examine the prolog with the Analyze/RMS_File Utility described in Chapter 10.

There are three types of prologs—Prolog 1, Prolog 2, and Prolog 3.

Prolog 1 and Prolog 2

Any indexed file created with a version of the VMS operating system lower than Version 3.0 is either a Prolog 1 file or a Prolog 2 file. Prolog 1 files and Prolog 2 files operate identically.

If an indexed file uses only string data type keys, the file is a Prolog 1 file. If an indexed file uses numeric type keys, it is a Prolog 2 file.

You cannot use the Convert/Reclaim Utility on a Prolog 1 file or a Prolog 2 file to reclaim empty buckets. If your file undergoes a large number of deletions, resulting in empty, unusable buckets, you must use the Convert Utility (CONVERT) to reorganize the file. (Note that CONVERT establishes new RFAs for the records.)

The compression allowed with Prolog 3 files is not possible with Prolog 1 or Prolog 2 files.

Prolog 3

Prolog 3 files can accept multiple (or alternate) keys and all data types (including the nonstring 8-byte BIN8 and INT8 types). They also give you the option of saving space by compressing your data, indexes, and keys.

Key compression compresses the key values stored in the data buckets. Likewise, index compression compresses the key values stored in index buckets, and data compression compresses the data portion of the records in the data buckets.

Note: You cannot use key compression or index compression with the collated key data type.

When keys are compressed in index or data records, repeating leading and trailing characters are compressed. With front key compression, any characters that are identical to the characters at the front of the previous key are compressed. For example, the keys JOHN, JOHNS, JOHNSON, and JONES appear as JOHN, S, ON, and NES.

With rear key compression, any repeating characters at the end of the key are compressed to a single character. For instance, the key JOHNSON00000 appears as JOHNSON0.

With data compression, instances of five or more repeated characters in a row are compressed to a single character.

Compression has a direct effect on CPU time and disk space. Compression increases CPU time, but the keys are smaller, so your application can scan more quickly through the data and index buckets.

The disk space saved by using Prolog 3 indexed files can significantly improve performance. With compression, each I/O buffer can hold more information to improve buffer space efficiency. Compression can also decrease the number of index levels, which decreases the number of I/O operations per random access.

Prolog 3 files can have segmented primary keys, but the segments cannot overlap. If you want to use a Prolog 3 file in this case, consider defining the overlapping segmented key as an alternate key and choosing a different key to be the primary key. If you want to use overlapping primary key segments, you must use a Prolog 2 file.

If record deletions result in empty buckets in Prolog 3 files, you can use the Convert/Reclaim Utility to make the buckets usable again. Because CONVERT/RECLAIM does not create a new file, RFAs remain the same.

Note that RMS–11 does not support Prolog 3 files. To use a Prolog 3 file with RMS–11 you must first use the Convert Utility to transform the file into a Prolog 1 file or into a Prolog 2 file.

3.5.1.2 Primary Index Structure

The primary index structure consists of the file's data records and a key pathway based on the primary key (key 0). The base of a primary index structure is the data records themselves, arranged sequentially according to the primary key value. The data records are called level 0 of the index structure.

The data records are grouped into buckets, which is the I/O unit for indexed files. Because the records are arranged according to their primary key values, no other record in the bucket has a higher key value than the last record in that bucket. This high key value, along with a pointer to the data bucket, is copied to an *index record* on the next level of the index structure, known as level 1.

The index records are also placed in buckets. The last index record in a bucket itself has the high key value for its bucket; this high key value is then copied to an index record on the next higher level. This process continues until all of the index records on a level fit into one bucket. This level is then known as the root level for that index structure.

Figure 3–1 is a diagram of an index structure.

Figure 3–2 illustrates a primary index structure. (For simplicity, the records are assumed to be uncompressed, and the contents of the data records are not shown.) The records are 132 bytes long (including overhead), with a primary key field of 6 bytes. Bucket size is one block, which means that each bucket on Level 0 can contain three records. You calculate the number of records per bucket in this case as follows:

```
512 bytes - 15 bytes of overhead = 497 bytes
497 / 132 = 3.77
```

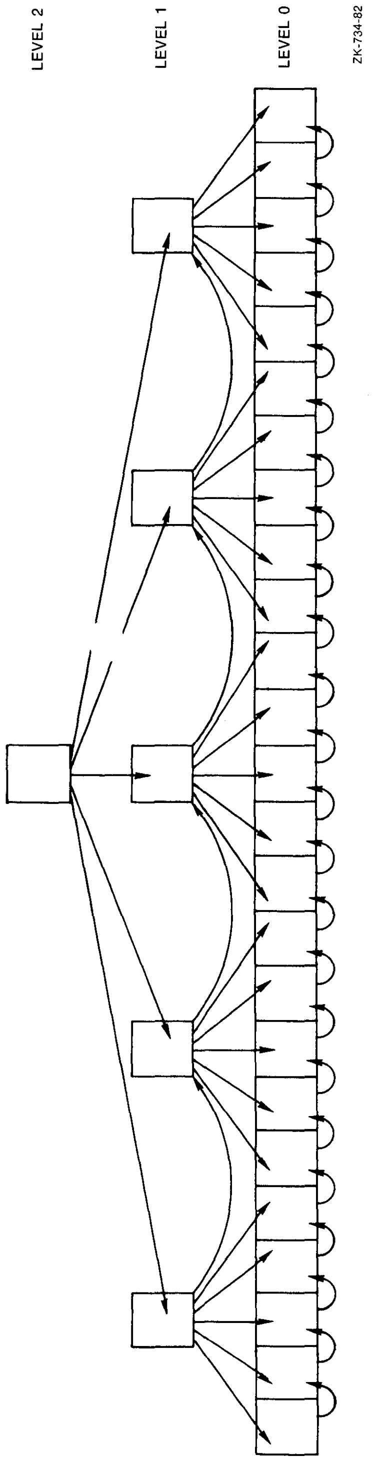
Note that you must round the remainder to the lowest integer, which is 3.

Because the key size is small and the database in this example consists of only 27 records, the index records can all fit in one bucket on level 1. The index records in this example are 6 bytes long. Each index record has one byte of control information. In this example, the size of the pointers is 2 bytes per index record, for a total index record size of 9 bytes. You calculate the number of records per bucket in this case as follows:

```
512 bytes - 15 bytes of overhead = 497 bytes
497 / 9 = 55.2
```

Again, you must round the remainder to the lowest integer, 55.

Figure 3–1 VMS RMS Index Structure



To read the record with the primary key 14, VMS RMS begins by scanning the root level bucket, looking for the first index record with a key value greater than or equal to 14. This record is the index record with key 15. The index record contains a pointer to the level 0 data bucket that contains the records with the keys 13, 14, and 15. Scanning that bucket, VMS RMS finds the record (see Figure 3–3).

3.5.1.3 Alternate Index Structure

Alternate indexes (also referred to as secondary indexes) provide your program with alternate record processing. If you have one or more alternate indexes, you can process data records using any of the alternate keys in addition processing data with the primary key. Note that a file with alternate indexes does require additional disk space.

The alternate index structure is similar to the primary index structure. except that instead of containing data records, alternate indexes contain secondary index data records (SIDRs). An SIDR includes an alternate key value from a data record stored in the primary index and one or more pointers to data records in the primary index. (SIDRs have pointers to more than one record only if you allow duplicate keys and there are duplicate key values in the database.) You do not need an SIDR for every data record in the database. If a variable-length record is not long enough to contain a given alternate key, an SIDR is not created. For example, if you define an alternate key field to be bytes 10 through 20 and you insert a 15-byte record, no SIDR is created in that alternate index structure.

When you create a file, you can use null key values to improve performance when your program uses alternate keys. When a secondary index has relatively few entries, VMS RMS performance may diminish because it tries to treat the null entries (typically blank keys) as duplicates. The resultant duplicate-key processing is unnecessary and can be avoided by assigning a null key value for the index. By using a null key value, you minimize the list of duplicates and this can improve performance when you insert records because the null key entries do not get processed as index entries. Note that when you sequentially access a records in a file that uses null key processing, VMS RMS does not process records that have null values for the key.

If you use the string data type, VMS RMS uses the ASCII null character as the default null key value. However, you can specify any character as the null value. If you use numeric keys, VMS RMS uses zero (0) as the null value.

3.5.1.4 Records

Records in an indexed file can be fixed-length records or variable-length records. Fixed-length records begin with a record header. Variable-length records include a record header followed by 2 bytes of record length overhead. Unlike variable-length records in relative files, each variable-length record in an indexed file requires only enough space for the record. See Table 2–2 for more information on record overhead.

Records cannot span bucket boundaries.

Figure 3–2 A Primary Index Structure

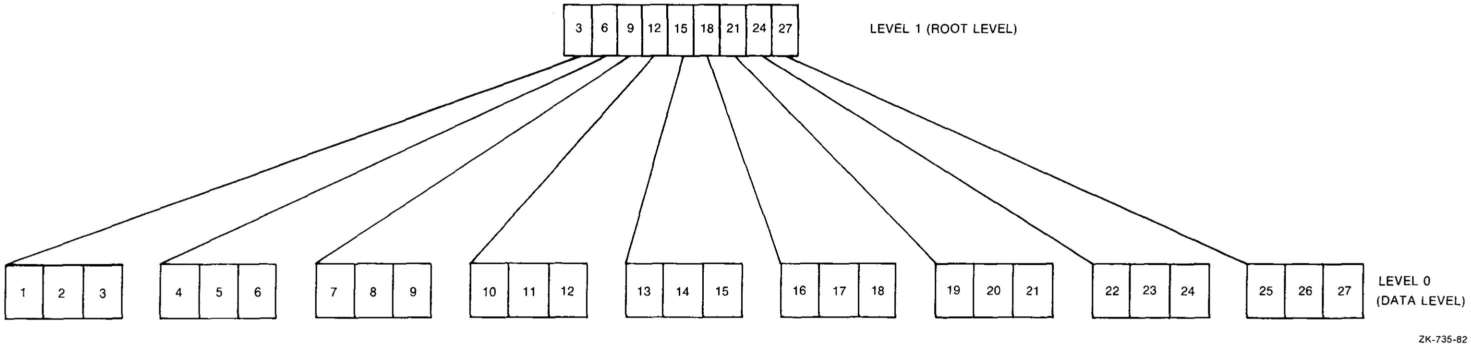
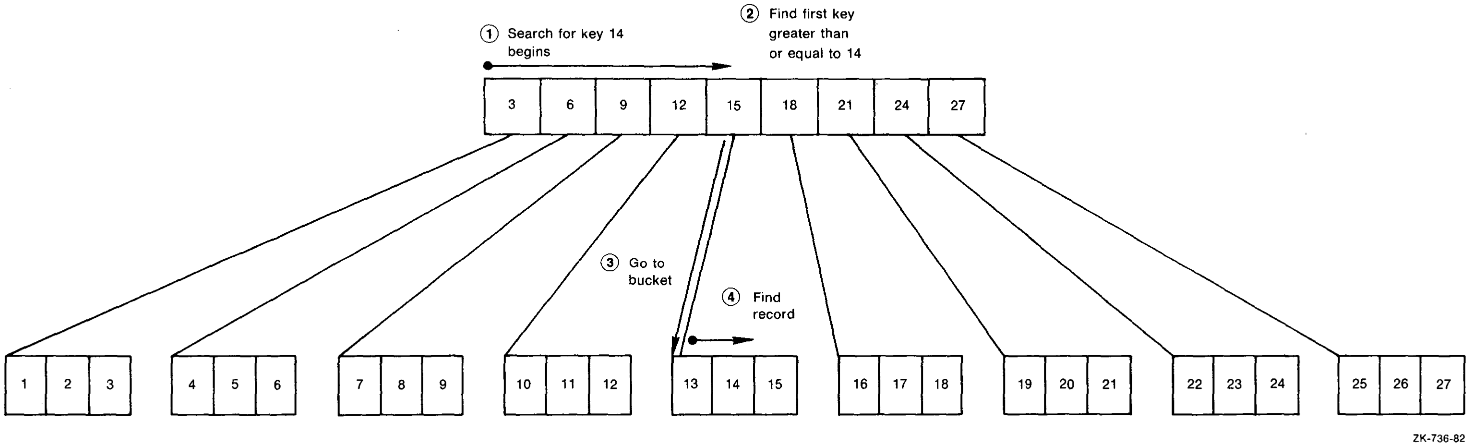


Figure 3–3 Finding the Record with Key 14



For Prolog 3 files, the maximum record size is 32,224 bytes. For Prolog 1 and Prolog 2 files, the maximum length for a fixed-length record is 32,234 bytes; the maximum length for a variable-length record is 32,232 bytes. Note that when you specify a record length for a Prolog 3 file that is greater than the maximum record length, VMS RMS automatically converts the file to a Prolog 1 or Prolog 2 file.

Record length should reflect application requirements. There is no advantage to using a record length that is based on the number of bytes in a bucket.

The value of the primary key must be contained within the records. The records can contain either a valid key field value for the alternate keys or, if you specify that null keys are allowed, a field of null characters.

3.5.1.5 Keys

A key is a record field that identifies the record to help you retrieve the record. There are two types of keys—primary keys and alternate keys. Data records are stored in the file in the order of their primary key. The most time-efficient value for primary keys is a unique value that begins at byte 0 of the record. You can allow duplicate keys in the primary index, but duplicate keys may slow performance.

The primary key and alternate keys can be character strings or numerical values. Key type is specified by the FDL attribute KEY TYPE.

If it is not possible to put the records into the file in order of their primary key, you should specify that the buckets not be filled completely when the file is loaded. If you attempt to write a record to a full bucket, a bucket split occurs. VMS RMS keeps half of the records in the original bucket and moves the other records to the newly created bucket. Each time a record moves to a new bucket, it leaves behind a forwarding pointer called a *record reference vector (RRV)*. You should avoid bucket splits because they use additional disk space and CPU time. An extra I/O operation is required to access a record in a split bucket when the program accesses a record by an alternate key or by RFA.

Alternate keys have a direct impact on I/O operations, CPU time, and disk space. The number of I/O operations and the CPU time required for Put, Update, and Delete operations are directly proportional to the number of keys. For example, inserting a record with a primary key and three alternate keys takes approximately four times longer than inserting a record with only a primary key.

To update the value of an alternate key, you have to traverse the alternate index structure twice, and bucket splits are more likely to occur. Randomly accessing an alternate key generally requires an extra I/O operation over a comparable access by the primary key, and extra disk space is required to store each alternate index structure.

Alternate keys are more likely than primary keys to have duplicate values. For example, the zip code is a common alternate key. However, allowing many duplicate values can have a performance cost. Duplicate values can cause clustered record or pointer insertions in data buckets, long sequential searches, a large number of I/O operations, and loss of physical contiguity due to continuation buckets (especially for the primary key).

Where possible, you should validate record keys before inserting the record, especially when you have primary and alternate keys.

In general, as the number of keys increases, so does the time it takes to add and delete records from your file. If CPU time is a critical resource on your system, you should define as few keys as possible.

If you are reading records in your file, the number of keys has relatively little impact on performance.

3.5.1.6 Areas

An *area* is a portion of an indexed file that VMS RMS treats as a separate entity. You can divide an indexed file into separate areas where each area has its own bucket size, initial allocation, extension size, and volume positioning, just as if each area were a separate file.

Using multiple areas has distinct advantages. However, if each area has a different bucket size, all buffers are as large as the largest bucket. If you use multiple areas, the file itself is probably not be contiguous; however, you can make each area within the file contiguous by specifying the FDL attribute AREA CONTIGUOUS. To ensure that the area is created without error, use the AREA BEST_TRY_CONTIGUOUS attribute.

When you separate key and data areas, you tend to keep related buckets close together, thereby decreasing disk seek time. You also minimize the number of disk-head movements for a series of operations. For example, if you have a dedicated multidisk volume set, you could place the data level of a file in an area on one disk and the index levels of the file in an area on a separate disk. Then there is little or no competition for the disk head on the disk that contains the index structures.

One strategy is to allocate a separate area for level 0 of a primary index (the data level). These buckets are the only ones referenced when you access the records sequentially by their primary key, so keeping them in a separate area optimizes that type of operation.

Do not allocate separate areas for level 1 of an index and the other index levels if the index has just one level. In such a case, you force VMS RMS to create an additional level in the index structure.

In most cases, you should allocate at least one area for each alternate index structure. By default, EDIT/FDL creates two areas in an indexed file for each index structure—one for the data level and one for all of the index levels. You can allocate up to 255 areas, so with most applications you can set up enough areas to handle all alternate index structures.

It is possible to set up a separate area for each of the following:

- Primary index level 0 (the data records)
- Primary index level 1 (the lowest index level)
- Primary index levels 2+ (the rest of the index levels)
- Alternate index level 0 (the secondary index data records)
- Alternate index level 1 (the lowest index level)
- Alternate index levels 2+ (the rest of the index levels)

Be sure to allocate sufficient space for each area and to specify area contiguity, because extending an area generally creates a noncontiguous area extent. The resulting noncontiguous extent may be anywhere on the disk, and you may lose the benefits of multiple areas.

If you are using a single area for the file, you should allocate enough contiguous space at creation time for the entire file. If you plan to add data to the file later, you should allocate extra space. The FDL attribute FILE ALLOCATION sets this value. To make sure the allocation is contiguous, set the FDL attribute FILE CONTIGUOUS to YES.

If you are using multiple areas, you should allocate each one by specifying a value for the FDL attribute AREA ALLOCATION.

If the file is relatively small or if you know that it needs to be extended, you do not have to use multiple areas. In such cases, it is more important to calculate the proper extension size.

To specify multiple areas using an FDL file, you assign each area its own AREA primary attribute. The AREA primary attribute takes as an argument a number whose value identifies the area.

Use the KEY primary attribute with its secondary attributes DATA_AREA, LEVEL1_INDEX_AREA, and INDEX_AREA to match each area specified with its index level. Key 0 is known as the primary key. Therefore, in the primary index structure, the primary attribute KEY must take the value 0. Then, within the KEY 0 section, you assign to the DATA_AREA secondary attribute the number that you used to define the area where you want the data records to appear.

You then match the KEY LEVEL1_INDEX_AREA secondary attribute with an AREA primary attribute by assigning the appropriate area number to the LEVEL1_INDEX_AREA secondary attribute. You also assign the number of an area to the INDEX_AREA secondary attribute for the other index levels in the primary index structure. For each alternate index structure, you use the same secondary attributes (DATA_AREA, LEVEL1_INDEX_AREA, INDEX_AREA) in another KEY primary attribute. In KEY sections that define alternate keys, the DATA_AREA is where VMS RMS puts the SIDRs.

3.5.2 Optimizing File Performance

This section discusses adjustments in file design that can improve a file's performance.

3.5.2.1 Bucket Size

For indexed files, the bucket size controls the number of levels in the index structure, which has the greatest impact on performance for most applications. You can specify the bucket size with the FDL attribute FILE BUCKET_SIZE or the VMS RMS control block fields FAB$B_BKS and XAB$B_BKZ. When you sequentially access files, large buckets are generally beneficial.

For keyed access to index files, set the bucket size so that the the number of index levels does not exceed four. In general, the smaller the bucket size, the deeper the tree structure. If you find that a small increase in bucket size eliminates one level, use a larger bucket size. At some point, however, the benefit of having fewer levels is offset by the cost of scanning through the larger buckets.

As a rule, you should never increase bucket size unless the increase reduces the number of levels. For example, you may find that a bucket size of 4 or more yields an index with four levels, and a bucket size of 10 or more yields an index with three levels. In this case, you never want to specify a bucket size of 9 because that reduces the number of levels, and performance does not improve. In fact, such a specification could hurt performance because each I/O operation takes longer, yet the number of accesses remains the same. However, larger bucket sizes always improve performance if you are accessing the records sequentially by primary key because more records fit in a bucket.

Conversely, with smaller buckets you have to search fewer keys. So if you can cache your whole structure (except for level 0), you can save a lot of time. Also, performance in this case is comparable to flat file design although add operations may take a little longer.

You can decrease the depth of your index structure in two ways. First, you can increase the number of records per bucket by increasing the bucket size, increasing the fill factor, using compression, or decreasing the size of keys and records.

Note: You cannot use key compression or index compression with the collated key data type.

However, changing the bucket size also has disadvantages. Larger buckets use more buffer space in memory. And the number of records per bucket determines bucket search time, which directly affects CPU time. A larger fill factor decreases the room for growth in the file, so bucket splits may occur. Compression increases the record search time.

Alternatively, you can reduce the index depth by decreasing the number of records in the file.

If you are using multiple areas, you can set a different bucket size for each area. You should use different bucket sizes if you are performing random accesses of records in no predictable pattern and if the data records are large. Using different bucket sizes allows you to specify a smaller size for the index structures and SIDRs than for the primary data level.

You can use the Edit/FDL Utility to determine the optimum bucket size.

Use the same bucket size for all areas if the data records are small or if the record accesses follow a clustered pattern, that is, if the records that you access have keys that are close in value.

In general, decreasing the bucket size increases other resources:

- Levels in the tree structure
- Buckets needed to maintain the tree structure
- Buffers needed for cache

Conversely, decreasing the bucket size *decreases* the pages per bucket and the average number of keys searched while traversing the tree.

3.5.2.2 Fill Factor

If you know that the application makes random insertions into the database, you should reserve some space in the buckets when records are first loaded into the file. You can specify a fill factor from 50% to 100%. For example, a fill factor of 50% means that VMS RMS writes records in only half of each bucket when the records are first loaded, leaving the remainder of the bucket empty for future write operations. This fill factor minimizes the number of bucket splits.

The fill factor is set with the FDL attributes KEY DATA_FILL and KEY INDEX_FILL. The value assigned to both attributes should be the same.

When you specify a fill factor, consider the following:

- If the inserted records are distributed unevenly (highly skewed) by their primary key value, then specifying a fill factor of less than 100% does not reduce the number of bucket splits.
- If the records have key values that are close or if they are added at one end of the file, many bucket splits occur anyway, and the partially filled buckets in the database just waste space. If this is the case, you should either specify a fill factor of 100% and use the Convert Utility to reorganize the file after the insertions are made, or you should choose a different primary key.
- If the inserted records are distributed fairly evenly or by their primary key, then specifying a fill factor of less than 100% could significantly reduce bucket splits. However, the trade-off is initially wasted disk space.

3.5.2.3 Number Of Buffers

At run time, you can specify the number of buffers with the FDL attribute CONNECT MULTIBUFFER_COUNT or the VMS RMS control block field RAB$B_MBF. The number of buffers each application needs depends on the type of record access your application performs.

The minimum number of buffers for indexed files is two. If the application performs sequential access on your database, two buffers are sufficient. More than two buffers for sequential access could actually degrade performance. During a sequential access, a given bucket is accessed as many times in a row as there are records in the bucket. After VMS RMS has read the records in that bucket, the bucket is not referenced again. Therefore, it is unnecessary to cache extra buckets when accessing records sequentially.

When you access indexed files randomly, VMS RMS must read the index portion of the file to locate the record you want to process. VMS RMS tries to keep the higher-level buckets of the index in memory; the buffers for the actual data buckets and the lower level index buckets tend to be reused first when other buckets need to be cached. Therefore, you should use as many buffers as your process working set can support so you can cache as many buckets as possible.

When you access records sequentially, even after you have located the first record randomly, you should use a large bucket size. A small multibuffer count, such as the default of two buffers, is sufficient.

If you process your data file with a combination of the above access modes, you should compromise on the recommended bucket sizes and number of buffers.

When you add records to an indexed file, consider choosing the deferred-write option (FDL attribute FILE DEFERRED_WRITE; FAB$L_FOP field FAB$V_DFW). With this option, the buffer into which the records have been moved is not written to disk until the buffer is needed for other purposes, the Flush service is used, or until the file is closed. The deferred-write option, however, may cause records to be lost if a system crashes before VMS RMS transfers the records to the disk.

In general, you must consider several trade-offs when you set the number of buffers your application needs:

- CPU time
- Availability of memory and number of page faults
- I/O operations

With indexed files, buckets (not blocks) are the units of transfer between the disk and memory. You specify the bucket size when you create the file, although you can change the bucket size of an existing file with the Convert Utility (see Chapter 10).

3.5.2.4 Global Buffers

If several processes share the indexed file concurrently, you may want to specify that the file use global buffers. A *global buffer* is an I/O buffer that two or more processes can access. If two or more processes request the same information from a file, each process can use the global buffers instead of allocating its own.

Only one copy of the buffers resides at any one time in memory although the buffers are charged against each process's working set size.

The guideline for using global buffers is the same as the guideline for using local process I/O buffers. Global buffers only provide significant benefits if more than one process refers to the same bucket in the global cache. If bucket contention is high, I/O transfers can be minimized and performance improved. However, global buffers do not always improve performance. For example, multiple processes independently reading records and using sequential access are most apt to refer to separate buckets. In that case, bucket contention is low and the number of I/O transfers is not reduced, so global buffers do not improve performance.

3.5.2.5 Using the Deferred-Write Option

The deferred-write option is a run-time option that can improve performance. It is the default operation for some languages and can be specified by clauses in other languages. If there is no language support, you can call a VAX MACRO subroutine that sets the FAB$L_FOP field, the FAB$V_DFW option.

When you select the deferred-write option, VMS RMS delays writing a modified bucket to the disk until the buffer is needed to read another bucket into the cache, or until another process needs to reference the modified bucket. If a subsequent operation references the bucket before it is flushed out to disk, then one I/O operation has been eliminated. Typically, the largest performance gains come from using the deferred-write option with sequential access, because random accesses of the file usually result in several I/O operations to bring in the single records.

Not all operations on indexed files can be deferred. Any operation that causes a bucket split forces the writeback of the modified buckets to disk. (This forced writeback decreases the chances of lost information should a system failure occur.)

Using the deferred-write option improves performance if you are performing multiple I/O operations on a bucket. Consider the following example. The indexed file has a single key and its records are 100 bytes long. The bucket size is 3 blocks with a fill factor of 67%. Thus, there is an average of 10 records in each bucket. A batch program reads each record and updates part of it, beginning at the first record in the file and moving through the records sequentially. Without the deferred-write option, 11 disk I/O operations occur for every 10 records—one to read the bucket and one to write the bucket for each record. With the deferred-write option, only two disk I/O operations occur for every 10 records—one to read the bucket and one to write the bucket after the record operations are completed.

3.6 Processing in a VAXcluster

This section discusses designing file applications for a multiple node VAXcluster and the performance you can reasonably expect from the VAXcluster environment.

Processing in a VAXcluster environment offers many advantages:

- Performance—In general, the performance of each node in a VAXcluster is similar to that of a single-node system that has the same processing load, assuming the aggregate I/O per disk drive is reasonable.
- Availability—With the appropriate configuration, a node that leaves the VAXcluster does not stop the VAXcluster.
- Flexibility—You can process shared applications on more than one node.
- Accessibility—Shared resources are very easy to use in a VAXcluster. The synchronized access to the data provides data integrity with no redundancy.

For more information about VAXclusters, see the *VMS VAXcluster Manual*.

3.6.1 VAXCluster Shared Access

Shared access is one of the chief advantages of processing in a VAXcluster environment. Many applications that run on a single-node system can run on a multiple-node VAXcluster with no changes.

However, applications that access shared files in a VAXcluster incur some additional overhead for the VAXcluster synchronization; the amount of additional overhead depends on the locking requirements of your application.

3.6.1.1 Locking Considerations

The distributed lock manager allows several users to share files concurrently in an organized manner. VMS RMS uses the lock manager to control file access.

The *lock-mastering node* controls the record and bucket locking for a given file for users on every node of the VAXcluster. Initially, it is the first node from which the file is opened. However, another node may become the lock-mastering node when a node either joins or leaves the VAXcluster.

The lock-mastering node may also change every time the file is opened. When another process opens the file (provided that the file was closed), the node on which that process resides becomes the new lock-mastering node for that file.

Lock requests issued by processes on the lock-mastering node incur less cost than lock requests issued from other nodes. Conversely, the lock-mastering node has the additional work of processing lock requests for that file for all other nodes.

The *lock-requesting node* is any node in the VAXcluster *other* than the lock-mastering node for a given file.

VMS RMS locks buckets and records during record operations only if the file is open for shared writing. Conversely, VMS RMS does no locking during record operations if the file is open for shared read-only access or for exclusive access.

Lock requests for *root locks* (top-level or parent locks) in a VAXcluster may be slightly slower than on a single-node system. However, these locks are used when you open and close files, so the time for lock operations is only a fraction of the total time needed to open and close files.

There is no performance difference between a single-node system and a VAXcluster if the file sharing takes place on a single node of the VAXcluster. Only when sharing spans across the VAXcluster nodes does distributed locking occur.

As a result, the record locking itself may take a little longer, but since you have multiple CPUs in the VAXcluster, your application benefits from the added processing power.

Sharing files in a VAXcluster also requires enough memory for nonpaged pool to store additional lock data structures. This requirement, however, is dependent upon your processing load.

3.6.1.2 I/O Considerations

Sharing files in a VAXcluster environment also means sharing resources, such as disks and other pieces of I/O hardware. When applications on many nodes share data on one disk, VAXcluster performance may degrade due to excessive I/O operations.

3.6.2 Performance Recommendations

Four general recommendations about performance in a VAXcluster environment are described in the following list:

- Estimate the I/O needs of your application. In a VAXcluster, and particularly with a shared file, multiple nodes can generate many I/O requests to a single disk. The capacity of the disk to handle I/O traffic can affect VAXcluster performance if you allow your applications to become I/O bound. The Monitor Utility is a good tool for estimating how many I/O requests your application generates. For more information about the Monitor Utility, see the *VMS Monitor Utility Manual*.
- Process files with exclusive access to obtain better performance than processing files with shared-write access. Opening files for unnecessary shared-write access incurs needless locking cost (even on a single node system).
- If possible, confine your application to a single CPU. If sufficient CPU resources and I/O capacity are available, your application performs faster than if it was spread over many nodes.
- Provide for sufficient memory because the space overhead for the lock database and other system software can be significant.

4 Creating and Populating Files

After you have designed your file, you need to create it. First you must specify the file characteristics you selected during the design phase. Then you need to create the actual file with those characteristics and to protect it (decide who has access to the file). Last, you need to put records in the file, or "populate" it.

This chapter describes the process of creating and populating files. Section 4.1 tells how to select and specify file-creation characteristics. Section 4.2 describes how to create a file. Section 4.3 explains how to define file protection, and Section 4.4 describes how to populate the file. A summary of the options related to file creation is provided in Section 4.5.

4.1 File Creation Characteristics

You can specify the characteristics you need to create a file in two ways. If you use VAX MACRO or BLISS–32, you can specify file characteristics by including VMS RMS control blocks in your application program.

If you use a high-level language, you can use the File Definition Language (FDL), a special-purpose language that is used to write specifications for data files. Of course, you also have the option of using FDL with MACRO or BLISS–32.

The following sections describe how you can specify file-creation characteristics by using VMS RMS control blocks or by creating FDL files.

4.1.1 Using VMS RMS Control Blocks

You can establish characteristics for the file you create by using a VMS RMS file access block (FAB) and VMS RMS extended attribute control blocks (XABs). These control blocks allow you to take the defaults that VMS RMS provides or to override the defaults and define the characteristics that suit your particular application.

4.1.1.1 File Access Block

The FAB is made up of fields that describe various file characteristics and contain the following file-related information:

- The addresses of the file name string and the default name string
- The file organization
- The record format
- Information about disk storage space

The FAB lets you use both the creation-time characteristics and the run-time characteristics of VMS RMS. You must define one FAB for each file your program opens or creates.

For more information about the FAB, see the *VMS Record Management Services Manual.*

4.1.1.2 Extended Attribute Blocks

Extended attribute blocks (XABs) are optional user control blocks that contain supplementary file-attribute information. The following is a partial list of XABs that can be used to declare supporting file information:

- Initial size and extent information (XABALL)
- File protection (XABPRO)
- Key definition (XABKEY)
- Date and time information (XABDAT)

Like the FAB, the XABs allow you to use both the creation-time characteristics and the run-time characteristics of VMS RMS.

With the XABs, you can define various file attributes beyond those specified in the FAB.

For more information about the extended attribute blocks, see the *VMS Record Management Services Manual.*

4.1.2 Using File Definition Language

FDL provides a way to create data files using special text files, generally called FDL files. FDL files are written in a file definition language, which permits you to specify appropriate attributes and values for the file.

You create and modify FDL files using the Edit/FDL Utility (EDIT/FDL). EDIT/FDL contains built-in design algorithms to help you optimize data file design. EDIT/FDL recognizes correct FDL syntax and informs you immediately of syntax errors. (You can use a text editor or the DCL command CREATE to create an FDL file, but you must then follow the validity rules listed in the *VMS File Definition Language Facility Manual.*)

You can also use the Analyze/RMS_File Utility to create FDL files from existing data files. FDL files created in this manner contain special analysis sections that you can use with EDIT/FDL to tune your data files.

You can use the Create/FDL Utility and the Convert Utility to create data files from the specifications in the FDL files.

By using an FDL file to create a data file from a higher-level language, you can specify most of the VMS RMS creation-time characteristics that are available with VMS RMS control blocks (FABs and XABs). However, to use all of the VMS RMS connect-time features, including wildcard characters, you must use the VMS RMS control blocks.

4.1.2.1 Using EDIT/FDL

You can use EDIT/FDL in two ways: with a terminal dialog (interactively) or without one (noninteractively).

If you use EDIT/FDL noninteractively, you can execute only the OPTIMIZE script. The OPTIMIZE script lets you optimize an existing FDL file without an interactive session. For more information, see Section 10.3.

Alternatively, if you use EDIT/FDL interactively, you can use all the scripts, each of which has a series of menus. When you invoke EDIT/FDL, it displays a main menu. To select a menu item, you only need to enter the first letter of the item because each selection has a unique first letter.

Table 4–1 summarizes the EDIT/FDL commands.

Table 4–1 Summary of EDIT/FDL Commands

Command	Function
ADD	Inserts one or more lines into the FDL definition. If the line already exists, you can replace it with your new line. Once you have inserted a line, you can continue to add lines until you are satisfied with that particular primary section. If no primary section exists to hold the secondary attribute being added, EDIT/FDL creates one.
DELETE	Removes one or more lines from the FDL definition. If you delete all of the secondary attributes in a primary section, you effectively remove the primary attribute. Once you have removed a line, you can continue to delete lines under that particular primary section.
EXIT	Creates the output FDL file, stores the current FDL definition in it, and terminates the EDIT/FDL session. EDIT/FDL leaves unchanged any FDL file that it used as input. The FDL file that is created is, by default, a sequential file with variable-length records and carriage-return record attributes, and has your process's default VMS RMS protection and ownership. To change these default settings, see Section 4.1.2.3.
HELP	Displays the top level help text for EDIT/FDL and then continues to prompt for more keywords. Pressing the RETURN key in response to the "Topic?" prompt or pressing CTRL/Z will return you to the main function prompt.
INVOKE	Prompts you for your choice of scripts and starts a series of logically ordered questions that help you create new FDL files or modify existing ones.
MODIFY	Allows you to change the value of one or more lines in the FDL definition. Once you have changed a line, you can continue to modify lines under that particular primary section.
QUIT	Aborts the session without creating an output FDL file. You can also press CTRL/C or CTRL/Y to abort the session.
SET	Allows you to establish defaults or to select any of the FDL editor characteristics you forgot to specify on the command line.

Table 4–1 (Cont.) Summary of EDIT/FDL Commands

Command	Function
VIEW	Displays the current FDL definition, which is what is put into the output FDL file if you gave the EXIT command.
?	Causes the utility to display more information about that question. You can enter the question mark character in response to any question asked by EDIT/FDL. In all cases, it will result in repetition of the question. Note too, that the utility responds to an invalid response in the same manner that it responds to a question mark.

CTRL/Z is equivalent to the EXIT command if you use it at the main menu level. If you use it from any other level, CTRL/Z returns you to the main menu level.

In most cases, a command from the main menu brings up a second level menu. For instance, typing the ADD command displays the following menu:

```
                    Legal Primary Attributes

ACCESS    attributes set the run-time access mode of the file
AREA x    attributes define the characteristics of file area x
CONNECT   attributes set various VMS RMS run-time options
DATE      attributes set the date parameters of the file
FILE      attributes affect the entire VMS RMS data file
KEY y     attributes define the characteristics of key y
NETWORK   attributes set run-time network access parameters
RECORD    attributes set the non-key aspects of each record
SHARING   attributes set the run-time sharing mode of the file
SYSTEM    attributes document operating system-specific items
TITLE     is the header line for the FDL file

Enter desired primary     (Keyword)[FILE]  :
```

One of the most important features of EDIT/FDL is that it helps you create FDL files that define indexed, relative, and sequential data files. To do this, EDIT/FDL provides seven scripts that guide you through an interactive session. You can choose one of these scripts at the start of a session, or you can instruct EDIT/FDL to automatically invoke a particular script each time that you enter the EDIT/FDL command.

Table 4–2 lists the seven scripts.

Table 4–2 EDIT/FDL scripts

Script	Function
ADD_KEY	Allows you to model or add to the attributes of a new index.
DELETE_KEY	Allows you to remove attributes from the highest-level index of your file.
INDEXED	Begins a dialog in which you are prompted for information about the indexed data file you want to create from the FDL file. EDIT/FDL supplies values for certain attributes.

Table 4–2 (Cont.) EDIT/FDL scripts

Script	Function
OPTIMIZE	Helps you redesign an FDL file using an analysis file from the Analyze/RMS_File Utility (ANALYZE/RMS_FILE/FDL). The FDL file itself is one of the inputs to EDIT/FDL. In effect, this script allows you to tune the parameters of your indexes using the file statistics from the FDL ANALYSIS sections produced by ANALYZE/RMS_FILE.
RELATIVE	Begins a dialog in which you are prompted for information about the relative data file to be created from the FDL file. EDIT/FDL supplies values for certain attributes.
SEQUENTIAL	Begins a dialog in which you are prompted for information about the sequential data file to be created from the FDL file. EDIT/FDL supplies values for certain attributes.
TOUCHUP	Begins a dialog in which you are prompted for information about how you want to change an existing index.

An interactive session is controlled by these EDIT/FDL scripts. You can invoke a script in two ways:

- You can select the INVOKE command from the main menu and then choose your script. When you answer the script questions, EDIT/FDL displays a list of FDL attributes and their assigned values. At this point, you can use EDIT/FDL commands to further modify the attribute values or to end the editing session.
- You can begin a script by entering the a DCL command in the following form:

  ```
  EDIT/FDL/SCRIPT=script-name
  ```

  This command bypasses the main menu to directly display the menu for the selected script.

Example 4–1 below shows a sample session with the FDL Editor.

Example 4–1 Sample FDL Edit Session

```
                  VAX-11 FDL Editor

  Add     to insert one or more lines into the FDL definition
  Delete  to delete one or more lines from the FDL definition
  Exit    to leave the FDL Editor after creating the FDL file
  Help    to obtain information about the FDL Editor
❶ Invoke  to initiate a script of related questions
  Modify  to change existing line(s) in the FDL definition
  Quit    to abort the FDL Editor with no FDL file creation
  Set     to specify FDL Editor characteristics
  View    to display the current FDL Definition
❷ Main Editor Function            (Keyword)[Help] : INVOKE

              Script Title Selection

  Add_Key       modeling and addition of a new index's parameters
  Delete_Key    removal of the highest index's parameters
  Indexed       modeling of parameters for an entire Indexed file
❸ Optimize      tuning of all indexes' parameters using file statistics
  Relative      selection of parameters for a Relative file
  Sequential    selection of parameters for a Sequential file
  Touchup       remodeling of parameters for a particular index
❹ Editing Script Title            (Keyword)[-] : INDEXED
❺ Target disk volume Cluster Size (1-1Giga)[3]    : 3
❻ Number of Keys to Define        (1-255)[1]      : 1

  Line    Bucket Size vs Index Depth      as a 2 dimensional plot
  Fill    Bucket Size vs    Load Fill Percent     vs Index Depth
❼ Key     Bucket Size vs        Key Length        vs Index Depth
  Record  Bucket Size vs       Record Size        vs Index Depth
  Init    Bucket Size vs Initial Load Record Count vs Index Depth
  Add     Bucket Size vs  Additional Record Count  vs Index Depth
❽ Graph type to display           (Keyword)[Line] : LINE
❾ Number of Records that will be Initially Loaded
  into the File                   (0-1Giga)[-]  : 100000
❿ (Fast_Convert NoFast_Convert RMS_Puts)
  Initial File Load Method        (Keyword)[Fast] : FAST
⓫ Number of Additional Records to be Added After
  the Initial File Load           (0-1Giga)[0]    : 0

⓬ Key  0 Load Fill Percent        (50-100)[100]   : 100
⓭ (Fixed Variable)
  Record Format                   (Keyword)[Var]  : VARIABLE
⓮ Mean Record Size                (1-32229)[-]    : 80
⓯ Maximum Record Size             (0,80-32229)[0] : 0

⓰ (Bin2 Bin4  Bin8  Int2  Int4  Int8  Decimal  String  Collated
      Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring Dcollated)
  Key  0 Data Type                (Keyword)[Str]  : STRING
⓱ Key  0 Segmentation desired     (Yes/No)[No]    : NO
⓲ Key  0 Length                   (1-255)[-]      : 9
⓳ Key  0 Position                 (0-32220)[0]    : 0
⓴ Key  0 Duplicates allowed       (Yes/No)[No]    : NO
㉑ File Prolog Version             (0-3)[3]        : 3
㉒ Data Key Compression desired    (Yes/No)[Yes]   : YES
㉓ Data Record Compression desired (Yes/No)[Yes]   : YES
```

Example 4–1 Cont'd. on next page

Example 4–1 (Cont.) Sample FDL Edit Session

```
㉔ Index Compression desired       (Yes/No)[Yes]   : YES

               *|
               9|
               8|
  Index        7|
               6|
  Depth        5|
               4|
               3|  3 3
               2|      2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
               1|                                                  1 1 1 1 1 1 1 1
                +- + - - - + - - - - + - - - - + - - - - + - - - - + - - - - + - +
㉕                 1       5         10        15        20        25        30  32
                                 Bucket Size (number of blocks)

   PV-Prolog Version      3 KT-Key  0 Type     String EM-Emphasis  Flatter ( 3)
   DK-Dup Key  0 Values  No KL-Key  0 Length        9 KP-Key  0 Position      0
   RC-Data Record Comp   0% KC-Data Key Comp       0% IC-Index Record Comp   0%
   BF-Bucket Fill      100% RF-Record Format Variable RS-Mean Record Size    80
   LM-Load Method Fast_Conv IL-Initial Load    100000 AR-Added Records        0
          (Type "FD" to Finish Design)
㉖        Which File Parameter (Mnemonic)[refresh]    : FD
㉗ Text for FDL Title Section      (1-126 chars)[null]
  : FDL_SESSION_EXAMPLE
㉘ Data File file-spec             (1-126 chars)[null]
  : EXAMPLE.DAT
㉙ (Carriage_Return FORTRAN None Print)
  Carriage Control                (Keyword)[Carr] : CARRIAGE_RETURN

  Emphasis Used In Defining Default:      (    Flatter_files   )
  Suggested Bucket Sizes:                 (     3     3    27 )
㉚ Number of Levels in Index:              (     2     2     1 )
  Number of Buckets in Index:             (    72    72     1 )
  Pages Required to Cache Index:          (   216   216    27 )
  Processing Used to Search Index:        (   168   168   766 )
㉛ Key  0 Bucket Size              (1-63)[3]       : 3
㉜ Key  0 Name                     (1-32 chars)[null]
  : SSNUM
㉝ Global Buffers desired          (Yes/No)[No]    : NO
㉞ The Depth of Key  0 is Estimated to be No Greater
  than 2 Index levels, which is 3 Total levels.
㉟ Press RETURN to continue (^Z for Main Menu)

                   VAX-11 FDL Editor

  Add     to insert one or more lines into the FDL definition
  Delete  to delete one or more lines from the FDL definition
  Exit    to leave the FDL Editor after creating the FDL file
㊱ Help    to obtain information about the FDL Editor
  Invoke  to initiate a script of related questions
  Modify  to change existing line(s) in the FDL definition
  Quit    to abort the FDL Editor with no FDL file creation
  Set     to specify FDL Editor characteristics
  View    to display the current FDL Definition
㊲ Main Editor Function            (Keyword)[Help] : EXIT
㊳ DISK$:[FOX.RMS]FDL_SESSION_EXAMPLE.FDL;1  40 lines
```

❶ The Main Editor Function menu displays the EDIT/FDL commands.

❷ The INVOKE command displays the Script Title Selection menu. Note that HELP is the default command so if you want online help, just press the RETURN key.

❸ The Script Title Selection menu shows the seven scripts you can choose to help you design your file. There is no default so you must explicitly select one of the scripts.

❹ Choose the INDEXED script to design an indexed data file.

❺ Choose a disk volume cluster size of three.

❻ Define only one key—the primary key.

❼ This menu provides a selection of graphic display types.

❽ Select a line plot display.

❾ Select 100,000 records to be loaded initially.

❿ Select the CONVERT/FAST_LOAD method of loading records into the data file.

⓫ Opt for no additional records after the initial load.

⓬ Elect a fill level of 100 percent for the primary index buckets.

⓭ Choose the variable-length record format.

⓮ Select an average record size of 80 characters.

⓯ Select an unlimited maximum record size.

⓰ Select the string data type for the primary key.

⓱ Opt to disallow segmentation in the primary key.

⓲ Set the length of the primary key to 9 bytes.

⓳ Define the initial position of the primary key at column 0.

⓴ Opt to disallow duplicates of the primary key.

㉑ Choose the Prolog 3 version.

㉒ Select data key compression.

㉓ Select data record compression.

㉔ Select index compression.

㉕ This is a line plot showing bucket size against index depth.

㉖ Type "FD" to finish the design session.

㉗ Enter the title of your FDL file specification.

㉘ Enter the file specification of your data file.

㉙ Select the CARRIAGE_RETURN carriage control.

㉚ This display shows the tuning emphasis you chose to design your file. It also shows suggested bucket sizes for various index level depths and other tuning information.

(31) Select the default bucket size for the primary key.

(32) Enter the name of the primary key.

(33) Choose whether you want global buffers.

(34) This message shows the depth of the primary key index and gives the total number of levels.

(35) Press the RETURN key to display the main menu.

(36) This is the main menu.

(37) Use the EXIT command to exit the editor and to create the FDL file.

(38) This message shows the resulting FDL file specification and the number of lines it contains.

Note that the example uses most of the suggested defaults. There are three ways to accept defaults:

- Press the RETURN key without entering a value.
- Use the /RESPONSES=AUTOMATIC qualifier when you invoke EDIT/FDL.
- Use the following sequence:
  1. Select the SET command from the main menu.
  2. Select RESPONSES from the SET menu.
  3. Accept the default (AUTO) when EDIT/FDL prompts for "Default responses in script."

Key compression and index compression are not acceptable options when you select a collated key data type.

When EDIT/FDL creates an FDL file, it groups the attributes into major sections. The section headings are called *primary attributes,* and the attributes within a primary section are called *secondary attributes.* Certain secondary attributes contain a third level of attributes called *qualifiers.*

The objective of using EDIT/FDL is to create an FDL file with optimum values for the various attributes. An FDL file contains a list of the primary and attributes with related qualifiers. If a primary or secondary attribute does not appear in the FDL file, it is assigned its default value.

Example 4–2 shows an FDL file. IDENT, SYSTEM, FILE, RECORD, AREA n, and KEY n are primary attributes; the others are secondary attributes.

Example 4–2 Sample FDL File

```
IDENT  " 1-MAR-1985 14:07:46   VAX-11 FDL Editor"

SYSTEM
        SOURCE                  VMS
FILE
        GLOBAL_BUFFER_COUNT     0
        NAME                    DISK$RMS:[RMSTEST]INDEXED.DAT;3
        ORGANIZATION            indexed
        OWNER                   [RMS1,TEST]
        PROTECTION              (system:RWED, owner:RWED, group:RE, world:)

RECORD
        BLOCK_SPAN              yes
        CARRIAGE_CONTROL        none
        FORMAT                  variable
        SIZE                    2048

AREA 0
        ALLOCATION              233
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             5
        EXTENSION               60
AREA 1
        ALLOCATION              5
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             5
        EXTENSION               5

AREA 2
        ALLOCATION              18
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             3
        EXTENSION               6

KEY 0
        CHANGES                 no
        DATA_AREA               0
        DATA_FILL               100
        DATA_KEY_COMPRESSION    no
        DATA_RECORD_COMPRESSION no
        DUPLICATES              no
        INDEX_AREA              1
        INDEX_COMPRESSION       no
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       1
        NAME                    "NUM"
        NULL_KEY                no
        PROLOG                  3
        SEG0_LENGTH             8
        SEG0_POSITION           0
        TYPE                    bin8
```

Example 4–2 Cont'd. on next page

Example 4–2 (Cont.) Sample FDL File

```
KEY 1
        CHANGES                 yes
        DATA_AREA               2
        DATA_FILL               100
        DATA_KEY_COMPRESSION    yes
        DUPLICATES              yes
        INDEX_AREA              2
        INDEX_COMPRESSION       yes
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       2
        NAME                    "NAME"
        NULL_KEY                yes
        NULL_VALUE              0
        SEG0_LENGTH             39
        SEG0_POSITION           9
        TYPE                    string
```

4.1.2.2 Designing an FDL File

When you want to create an FDL file, you invoke EDIT/FDL with a DCL command in the following form:

```
EDIT/FDL/CREATE fdl-filespec
```

The /CREATE qualifier specifies that you want to create an FDL file with the name entered in the **fdl-filespec** parameter. When EDIT/FDL displays the main menu, select the INVOKE command. In response to the INVOKE command, EDIT/FDL prompts you for a script. The only appropriate scripts for creating a file are INDEXED, RELATIVE, and SEQUENTIAL.

As discussed previously, you can enter a script directly by specifying the /SCRIPT qualifier on the DCL command line. For example, enter the following command to create an indexed FDL file:

```
$ EDIT/FDL/CREATE/SCRIPT=INDEXED MY_FDL_FILE
```

When you select the script, EDIT/FDL prompts you for information about the data file. Each prompt consists of a short question, a range of acceptable values (for example, 50-100) or the value type (for example, Keyword, YES/NO, and so forth) in parentheses, and the default answer in brackets. One of the questions in the INDEXED script is shown as follows:

```
Number of Keys to Define (1-255)[1]       :
```

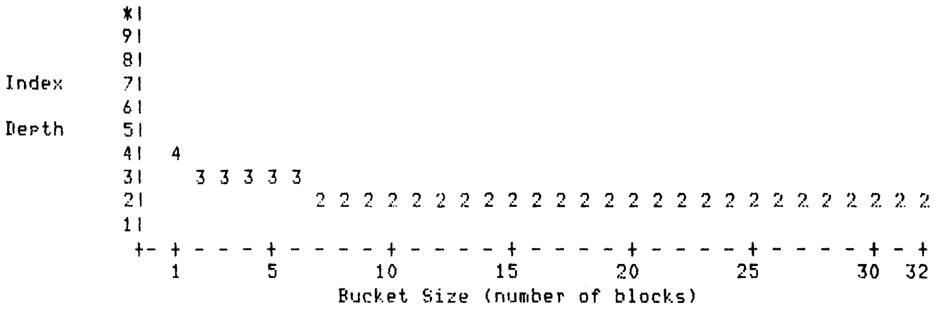
In this example, EDIT/FDL prompts you for the number of keys you want to define for an indexed data file. EDIT/FDL accepts any number from 1 to 255. If you do not specify a value, it assumes that you want to define one key only, the primary key. To accept the default value, press the RETURN key.

If EDIT/FDL requires that you enter a value (that is, no default value is specified for the response), it includes a dash within brackets [-].

When you specify the SEQUENTIAL script or the RELATIVE script, EDIT/FDL returns you to the main menu level after finishing the dialog. When you specify the INDEXED script, one of the prompts requests your choice of a design graphics display: a Line_Plot graph or a Surface_Plot graph. After finishing the dialog, EDIT/FDL displays the selected graph to help you make your final design choice.

The Line_Plot graph plots bucket size against index depth. All things equal, the size of the buckets determines the number of levels in the index, and the number of levels has a direct effect on the run-time performance of an indexed file. Fewer levels generally reduce the average number of keys searched when the index tree is traversed. However, fewer levels imply more records per data bucket and may cause longer data bucket search times. Thus, the Line_Plot graph helps you decide on the best bucket size for your application. Figure 4–1 shows a Line_Plot graph.

Figure 4–1 A Line_Plot Graph



```
        *|
        9|
        8|
Index   7|
        6|
Depth   5|
        4|  4
        3|    3 3 3 3 3
        2|              2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
        1|
         +- + - - - + - - - - + - - - - + - - - - + - - - - + - - - - + - +
            1       5         10        15        20        25        30  32
                          Bucket Size (number of blocks)
```

ZK-980-82

As shown in Figure 4–1, a bucket size of 1 block results in an index with five levels. Increasing the bucket size to 2 blocks reduces the number of index levels to four, but an increase to 5 blocks does not reduce the number of index levels at all. A bucket size of 7 blocks, however, reduces the number of index levels to three.

When you choose the bucket size, remember that the graphs do not display the data level. For example, if you want three levels in the file, then you must limit the number of index levels to two.

The Surface_Plot graphics mode lets you choose a range of values to see their effects. EDIT/FDL prompts you to enter a lower and upper bound for one of the following values:

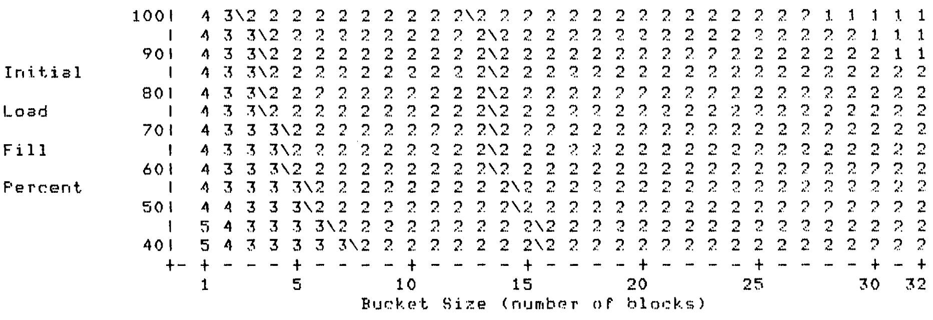
- Load fill percent
- Key length
- Record size
- Initial load record count
- Additional record count

The selected range is displayed along the graph's vertical axis.

The variable on the graph's horizontal axis is bucket size. The numbers in the field portion of the graph show the number of levels at each bucket size for each of the other values.

Figure 4–2 is a Surface_Plot graph that shows a range of values for initial fill factors ranging from 100% to 40%.

Figure 4–2 A Surface_Plot Graph



```
          100| 4 3\2 2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1
             | 4 3 3\2 2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1
           90| 4 3 3\2 2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1
Initial      | 4 3 3\2 2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
           80| 4 3 3\2 2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Load         | 4 3 3\2 2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
           70| 4 3 3 3\2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Fill         | 4 3 3 3\2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
           60| 4 3 3 3\2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
Percent      | 4 3 3 3 3\2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
           50| 4 4 3 3 3\2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
             | 5 4 3 3 3 3\2 2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
           40| 5 4 3 3 3 3 3\2 2 2 2 2 2 2 2\2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
            +- + - - - + - - - - + - - - - + - - - - + - - - - + - - - - + - +
               1       5        10        15        20        25        30  32
                                 Bucket Size (number of blocks)
```

ZK-949-82

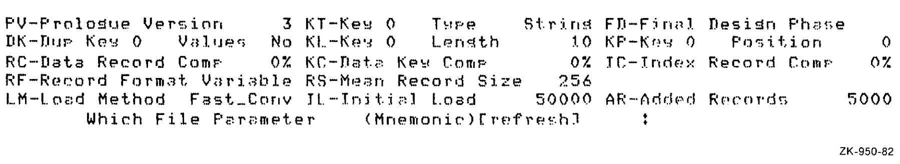
The area on the graph within the slash marks represents combinations that VMS RMS finds acceptable. In Figure 4–2, a fill factor of 70% and a bucket size of 10 blocks is the optimum combination. A fill factor of 70% and a bucket size of 15 blocks is a relatively poor combination because it falls outside of the slash boundaries.

If you are sure the information you supplied to EDIT/FDL is valid, the best values are those that lie along the left-hand boundary next to the slash marks. If you are not sure that your information is valid, you should choose a value that lies more to the right of the slash boundary.

When you complete the dialog and EDIT/FDL presents the graph, you can make changes to certain attributes of the proposed data file. The design is not complete until you specify "FD" for "Finish Design," at which point EDIT/FDL asks a few more questions. You then have the opportunity to return to the main menu to view the file attributes that EDIT/FDL has created.

Figure 4–3 shows the attributes that you can alter when EDIT/FDL displays the graph. Note that each attribute has a 2-letter mnemonic. To alter an attribute, you specify the corresponding mnemonic. To refresh the display, press the RETURN key. To begin the final design phase, enter "FD."

Figure 4–3 Design Mnemonics



```
PV-Prologue Version      3 KT-Key 0   Type     String FD-Final Design Phase
DK-Dup Key 0   Values   No KL-Key 0   Length       10 KP-Key 0   Position     0
RC-Data Record Comp     0% KC-Data Key Comp        0% IC-Index Record Comp   0%
RF-Record Format Variable RS-Mean Record Size    256
LM-Load Method  Fast_Conv IL-Initial Load       50000 AR-Added Records     5000
        Which File Parameter    (Mnemonic)[refresh]     :
```

ZK-950-82

During the final design phase, EDIT/FDL gives you an opportunity to supply values for such attributes as TITLE, an optional primary that allows you to label the FDL file. (Most of these questions are also applicable to designing sequential and relative files.) When you have answered the questions, EDIT/FDL assigns the values to the FDL attributes and returns you to the main menu level to display the resulting FDL file.

At the main menu, you can select the ADD command to assign values to any attribute the script omitted. Remember that if an attribute does not appear in the FDL file, it assumes the default value. (For a list of the default values for each attribute, see the *VMS File Definition Language Facility Manual*.) To modify an attribute, use the MODIFY command, and to delete an attribute, use the DELETE command.

To create the displayed FDL file, select the EXIT command. To abort the session without creating an FDL file, select the QUIT command.

4.1.2.3 Setting Characteristics For FDL Files

The FDL file that you use to create data files has certain characteristics of its own. For instance, by default FDL files have variable-length records, and the default carriage control for FDL files is a carriage return. Other values such as protection come from the VAX RMS system and process defaults.

You can create an FDL file with characteristics other than the defaults by first creating an FDL file to specify the characteristics you want to apply to the other FDL files. Then, use DCL to assign this FDL file the logical name EDF$MAKE_FDL. EDIT/FDL uses this file to determine the file characteristics for any FDL files it subsequently creates.

Note that the contents of the FDL files are still determined by the attribute values assigned during the EDIT/FDL session; only the external characteristics of the FDL files are affected by the EDF$MAKE_FDL file.

For example, use the following procedure to customize the protection for your FDL files.

1 Create the FDL file OUTSPEC.FDL containing only the following attributes:

```
FILE
        PROTECTION (SYSTEM=RWED,OWNER=RWED,GROUP=R)
```

2 Assign this file the logical name EDF$MAKE_FDL with the following DCL command:

```
$ ASSIGN OUTSPEC.FDL EDF$MAKE_FDL
```

From this point on, FDL files created by EDIT/FDL allow READ, WRITE, EXECUTE, and DELETE access to SYSTEM and to OWNER, allow READ access only to GROUP members, and deny access to WORLD.

4.1.3 Using the FDL Routines

You can also define file-creation characteristics with the FDL utility routines. The FDL routines provide you with the functions of the File Definition Language, and they allow you to set file creation characteristics from within your application.

There are four FDL routines:

FDL$CREATE Creates a file from an FDL specification and then closes the file. See Section 4.2.4 for more information.

FDL$GENERATE Produces an FDL specification by interpreting a set of VMS RMS control blocks. It then writes the FDL specification either to an FDL file or to a character string.

FDL$PARSE Parses an FDL specification, allocates control blocks, and then fills in the relevant fields.

FDL$RELEASE Deallocates the virtual memory used by the VMS RMS control blocks created by FDL$PARSE. You must use FDL$PARSE to fill in (to populate) the control blocks if you plan to release the memory with FDL$RELEASE later.

Because the FDL$GENERATE, FDL$PARSE, and FDL$RELEASE routines allow you to use the run-time, as well as the creation-time, features of VMS RMS, you must call them from a language that can access the VAX RMS control block fields that specify the CONNECT options. This may be difficult from a high-level language.

Example 4–3 shows how to call the FDL$PARSE and FDL$GENERATE routines from a Pascal program.

Example 4–3 Using FDL Routines in a Pascal Program

```
[INHERIT ('SYS$LIBRARY:STARLET')]
PROGRAM example2 (input,output,order_master);

(* This program fills in its own FAB, RAB, and  *)
(* XABs by calling FDL$PARSE and then generates *)
(* an FDL specification by calling FDL$GENERATE.*)
(* It requires an existing input FDL file       *)
(* (TESTING.FDL) for FDL$PARSE to parse.        *)

TYPE
(*+                                             *)
(* FDL CALL INTERFACE CONTROL FLAGS             *)
(*-                                             *)
        $BIT1 = [BIT(1),UNSAFE] BOOLEAN;
```

Example 4–3 Cont'd. on next page

Example 4–3 (Cont.) Using FDL Routines in a Pascal Program

```
        FDL2$TYPE = RECORD CASE INTEGER OF
        1: (FDL$_FDLDEF_BITS : [BYTE(1)] RECORD END;
           );
        2: (FDL$V_SIGNAL : [POS(0)] $BIT1;
                (* Signal errors; don't return          *)
            FDL$V_FDL_STRING : [POS(1)] $BIT1;
                (* Main FDL spec is a char string       *)
            FDL$V_DEFAULT_STRING : [POS(2)] $BIT1;
                (* Default FDL spec is a char string    *)
            FDL$V_FULL_OUTPUT : [POS(3)] $BIT1;
                (* Produce a complete FDL spec          *)
            )
        END;

    mail_order =  RECORD
                  order_num : [KEY(0)] INTEGER;
                  name : PACKED ARRAY[1..20] OF CHAR;
                  address : PACKED ARRAY[1..20] OF CHAR;
                  city : PACKED ARRAY[1..19] OF CHAR;
                  state : PACKED ARRAY[1..2] OF CHAR;
                  zip_code : [KEY(1)] PACKED ARRAY[1..5]
                       OF CHAR;
                  item_num : [KEY(2)] INTEGER;
                  shipping : REAL;
                  END;
    order_file  = [UNSAFE] FILE OF mail_order;
    ptr_to_FAB  = ^FAB$TYPE;
    ptr_to_RAB  = ^RAB$TYPE;
    byte = 0..255;

VAR
    order_master : order_file;
    flags        : FDL2$TYPE;
    order_rec    : mail_order;
    temp_FAB     : ptr_to_FAB;
    temp_RAB     : ptr_to_RAB;
    status       : integer;

FUNCTION FDL$PARSE
    (%STDESCR FDL_FILE : PACKED ARRAY [LL.:INTEGER]
         OF CHAR;
    VAR FAB_PTR : PTR_TO_FAB;
    VAR RAB_PTR : PTR_TO_RAB) : INTEGER; EXTERN;
FUNCTION FDL$GENERATE
    (%REF FLAGS : FDL2$TYPE;
    FAB_PTR : PTR_TO_FAB;
    RAB_PTR : PTR_TO_RAB;
    %STDESCR FDL_FILE_DST : PACKED ARRAY [LL.:INTEGER]
         OF CHAR) : INTEGER;
    EXTERN;

BEGIN

    status := FDL$PARSE ('TESTING',TEMP_FAB,TEMP_RAB);
    flags::byte := 0;
    status := FDL$GENERATE (flags,
                            temp_FAB,
                            temp_RAB,
                            'SYS$OUTPUT:');
```

Example 4–3 Cont'd. on next page

Example 4–3 (Cont.) Using FDL Routines in a Pascal Program

```
END.
```

For more information about FDL routines, see the *VMS Utility Routines Manual.*

4.2 Creating a File

After you select the creation characteristics for your file, you use the selected characteristics to create the file. You can create the file using one of the following:

- VMS RMS Create service
- Create/FDL Utility
- Convert Utility
- FDL$CREATE routine

4.2.1 Using the VMS RMS Create Service

The VMS RMS Create service creates a new data file assigning it the attributes you specify in the FAB and any applicable XABs. Note that where there is a conflict, the XAB fields override the FAB fields.

When you use the Create service to create a file, the file remains open until you explicitly close it.

If you set the create-if (CIF) bit in the FOP (file-processing options) field of the FAB, you can open an existing file with the VMS RMS Create service. If the file you try to create has the same name as an existing file, the Create service opens the existing file instead of creating the new file.

The Create service allows you to set file-creation characteristics and to create the file directly from your application program.

For more information about the Create service, see the *VMS Record Management Services Manual.*

4.2.2 Using the Create/FDL Utility

Unlike the Create service, using FDL to create a file is a two-step process. You must first create the FDL file using EDIT/FDL, and then use another VMS RMS utility or your application program to create the data file.

One of the utilities you can use to create a file is the Create/FDL Utility (CREATE/FDL). CREATE/FDL creates an empty data file from the specifications in an existing FDL file. This feature allows you to use EDIT/FDL to create standard FDL files that describe commonly needed data files and then to use CREATE/FDL to create the data files as they are needed

For example, to create an empty data file called CUSTRECS.DAT from the specifications in an FDL file called INDEXED.FDL, enter the following DCL command:

```
$ CREATE/FDL=INDEXED.FDL CUSTRECS.DAT
```

4.2.3 Using the Convert Utility

Another VMS RMS utility that creates an output data file from the specifications in an FDL file is the Convert Utility (CONVERT). However, instead of being empty, the new output file generally contains data records from the input file unless the input file was also empty.

If you want to use CONVERT to change the characteristics of a particular file, you can use a DCL command of the following form:

```
CONVERT/FDL=fdl-file input-file output-file
```

The CONVERT/FDL command creates a new file named by the **output-file** parameter and assigns the new file the characteristics specified in the FDL file.

For more information about populating data files with CONVERT, see Section 4.4.

4.2.4 Using the FDL$CREATE Routine

You can also create data files according to your specifications with the FDL$CREATE routine. FDL$CREATE is the FDL routine most likely to be called from a high-level language. It creates a file from an FDL specification and then closes the file.

The FDL$CREATE routine performs the same function as the Create/FDL Utility, but it allows you to create data files from your application. However, it allows you to use only the creation-time features of VMS RMS.

Example 4–4 shows how to call the FDL$CREATE routine from a FORTRAN program.

Example 4–4 Using the FDL$CREATE Routine in a FORTRAN Program

```
*       This program calls the FDL$CREATE routine.  It
*       creates an indexed output file named NEW_MASTER.DAT
*       from the specifications in the FDL file named
*       INDEXED.FDL.  You can also supply a default file name
*       and a result name (which receives the name of the created
*       file).  The program also returns all statistics.
*
        IMPLICIT        INTEGER*4       (A - Z)
        EXTERNAL        LIB$GET_LUN,    FDL$CREATE
        CHARACTER       IN_FILE*11      /'INDEXED.FDL'/,
        1               OUT_FILE*14     /'NEW_MASTER.DAT'/,
        1               DEF_FILE*11     /'DEFAULT.FDL'/,
        1               RES_FILE*50
        INTEGER*2       FIDBLK(3)       /0,0,0/
        I = 1
        STATUS = FDL$CREATE (IN_FILE,OUT_FILE,
                 DEF_FILE,RES_FILE,FIDBLK,,)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
*
        STATUS=LIB$GET_LUN(LOG_UNIT)
        OPEN (UNIT=LOG_UNIT,FILE=RES_FILE,STATUS='OLD')
        CLOSE (UNIT=LOG_UNIT, STATUS='KEEP')
*
```

Example 4–4 Cont'd. on next page

Example 4–4 (Cont.) Using the FDL$CREATE Routine in a FORTRAN Program

```
        WRITE (6,1000) (RES_FILE)
        WRITE (6,2000) (FIDBLK (I), I=1,3)
*
1000    FORMAT  (1X,'The result filename is: ',A50)
*
2000    FORMAT  (/1X,'FID-NUM: ',I5/,
        1         1X,'FID-SEQ: ',I5/,
        1         1X,'FID-RVN: ',I5)
*
        END
```

Example 4–5 shows how to call the FDL$CREATE routine from a COBOL program.

Example 4–5 Using the FDL$CREATE Routine from a COBOL Program

```
*       FDLCR.COB
*
*       This program calls the FDL$CREATE routine.  It creates
*       an indexed output file named NEW_MASTER.DAT from the
*       specifications in the FDL file named INDEXED.DAT.  You
*       can also supply a default file name and a result name
*       (that receives the name of the created file).  The
*       program also returns the FDL$CREATE statistics.
*
*       DATA NAMES:
*
*       OUT-REC    defines the output record
*       STATVALUE  receives the status value from the routine
*                  call
*       NORMAL     receives the value from SS$_NORMAL
*       FIDBLOCK   receives the FDL$CREATE statistics.  There
*                  are three:
*                  (1) file identification number (FID-NUM)
*                  (2) file sequence number       (FID-SEQ)
*                  (3) relative volume number     (RVN)
*       RESNAME    receives the name of the file that is created
*                  (the result file name)
*
IDENTIFICATION DIVISION.
PROGRAM-ID. FDL-CREATE-EXAMPLE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-780.
OBJECT-COMPUTER. VAX-780.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OUT-FILE ASSIGN TO 'NEWMASTER.DAT'.

DATA DIVISION.
FILE SECTION.
FD      OUT-FILE
        DATA RECORD IS OUT-REC.
```

Example 4–5 Cont'd. on next page

Example 4–5 (Cont.) Using the FDL$CREATE Routine from a COBOL Program

```
01      OUT-REC.
        02      OUT-NUM     PIC X(4).
        02      OUT-NAME    PIC X(20).
        02      OUT-COLOR   PIC X(4).
        02      OUT-WEIGHT  PIC X(4).
        02      SUPL-NAME    PIC X(20).
        02      FILLER      PIC X(28).
WORKING-STORAGE SECTION.
01      MORE-DATA-FLAGS     PIC XXX     VALUE 'YES'.
        88      THERE-IS-DATA           VALUE 'YES'.
        88      THERE-IS-NO-DATA        VALUE 'NO '.

01      STATVALUE           PIC S9(9)   COMP.

01      FIDBLOCK            USAGE IS COMP.
        02      NUM         PIC S9(9)   VALUE 0.
        02      SEQ         PIC S9(9)   VALUE 0.
        02      RVN         PIC S9(9)   VALUE 0.

01      RESNAME             PIC X(50).

PROCEDURE DIVISION.
MAIN.
        PERFORM CREATE-FILE THRU DISPLAY-STATS.
        STOP RUN.

CREATE-FILE.
        CALL 'FDL$CREATE' USING BY DESCRIPTOR 'INDEXED.FDL'
                                BY DESCRIPTOR 'NEWMASTER.DAT'
                                BY DESCRIPTOR 'DEFAULT.DAT'
                                BY DESCRIPTOR RESNAME
                                BY REFERENCE FIDBLOCK
                                BY VALUE 0
                                BY VALUE 0
                                BY VALUE 0
                                BY VALUE 0
                                BY VALUE 0
                          GIVING STATVALUE.

        IF STATVALUE IS FAILURE
        CALL 'LIB$STOP' USING BY VALUE STATVALUE.

DISPLAY-STATS.
        DISPLAY 'The result filename is: ',RESNAME CONVERSION.
        DISPLAY 'FID number:             ',NUM CONVERSION.
        DISPLAY 'FID sequence:           ',SEQ CONVERSION.
        DISPLAY 'Volume number:          ',RVN CONVERSION.
```

4.3 Defining File Protection

After you have created a file, you want to protect it against accidental or unauthorized access. You can protect a disk file in two ways:

- User identification codes (UICs)
- Access control lists (ACLs)

Magnetic tape files can be protected only with UICs.

4.3.1 UIC-Based Protection

You can protect both disk and magnetic tape files with UICs. This type of protection is made up of two parts: an owner UIC and a protection code.

The owner UIC is normally the UIC of the person who created the file. The protection code indicates who is allowed access and what type of access they are permitted.

When you try to open a file, your UIC is compared to the owner UIC of the file. Depending on the relationship of the UICs, you may be classified under one or more of the following categories:

- SYSTEM
- OWNER
- GROUP
- WORLD

Depending on your classification, you may be allowed or denied the following types of access:

READ	Can examine, print, or copy a disk or tape file
WRITE	Can modify or write to a disk or tape file
EXECUTE	Can execute a disk file that contains executable program images
DELETE	Can delete a disk file

You can specify the UIC-based protection value you need when the file is created if you use either an FDL specification or VMS RMS directly.

After you create a file, you can change its UIC-based protection with the DCL command SET PROTECTION. For more information about the SET PROTECTION command, see the *VMS DCL Dictionary*.

The previous list omits CONTROL access because it is never specified in the standard UIC-based protection code. However, CONTROL access can be specified in an ACL and is automatically granted to certain user categories when UIC-based protection is evaluated.

CONTROL access grants the accessor all the privileges of the object's actual owner. For more information, see the *Guide to VMS System Security*.

4.3.2 ACL-Based Protection

You can also protect disk files with access control lists (ACLs). (ACLs cannot be used with files stored on magnetic tape.)

An ACL is a list of people or groups who are allowed to access a particular file. ACLs offer more scope than UICs in determining what action you want taken when someone tries to access your file. You can provide an ACL on any file to permit as much or as little access as you want.

You can specify the ACL for a file when you create it if you use VMS RMS directly. You cannot specify an ACL in an FDL specification, and ACLs are not supported over DECnet.

After a file is created, you can define the access control list for it with the ACL Editor. You can invoke this editor with either of the following DCL commands:

- EDIT/ACL
- SET FILE/ACL

For more information about how to invoke, modify, and display ACLs, see the *VMS Access Control List Editor Manual.* For additional information about VMS security features, see your system or security manager, or consult the *Guide to VMS System Security.*

4.4 Populating a File

The next two sections explain how to use the Convert Utility to populate a file.

4.4.1 Using the Convert Utility

The Convert Utility allows you to create and populate a file.

To create a file, you need an input data file and an FDL file that describes the output file you want to create. You issue a DCL command in the following form:

```
CONVERT/CREATE/FDL=fdl-file  input-file  output-file
```

As with the CREATE/FDL command, the CONVERT/CREATE/FDL command creates a file named by the **output-file** parameter and having characteristics specified in your FDL file. Unlike the CREATE/FDL command, CONVERT populates the output file with the records from the input file.

For example, to create the file CUST.IDX from the specifications in the FDL file STDINDEX.FDL and copy the records from the input file CUST.SEQ into CUST.IDX, you enter the following command:

```
$ CONVERT/CREATE/FDL=STDINDEX.FDL CUST.SEQ CUST.IDX
```

VMS RMS assigns the records in CUST.IDX the characteristics specified in the file STDINDEX.FDL.

4.4.2 Using the Convert Routines

You can invoke the functions of the Convert Utility from your application program by calling the following series of convert routines:

CONV$PASS_FILES
Names the files to be converted. You can also specify an FDL file.

CONV$PASS_OPTIONS
Indicates the CONVERT qualifiers that you want to use. You may specify any legal CONVERT option, or you may accept the defaults.

CONV$CONVERT
Copies records from one or more source data files to an output data file. The output file is not required to have the same file organization and format as the source files.

The routines must be called in this order.

Example 4–6 shows how to call the CONVERT routines from a FORTRAN program.

Example 4–7 shows how to call the CONVERT routines from a COBOL program.

Example 4–6 Using the CONVERT Routines in a FORTRAN Program

```
*            This program calls the routines that perform the
*            functions of the Convert Utility.  It creates an
*            indexed output file named CUSTDATA.DAT from the
*            specifications in an FDL file named INDEXED.FDL.
*            The program then loads CUSTDATA.DAT with records
*            from the sequential file SEQ.DAT.  No exception
*            file is created.  This program also returns all
*            CONVERT statistics.
*            Program declarations

        IMPLICIT        INTEGER*4 (A - Z)

*            Set up parameter list: number of options, CREATE,
*            NOSHARE, FAST_LOAD, MERGE, APPEND, SORT, WORK_FILES,
*            KEY=0, NOPAD, PAD CHARACTER, NOTRUNCATE,
*            NOEXIT, NOFIXED_CONTROL, FILL_BUCKETS, NOREAD_CHECK,
*            NOWRITE_CHECK, FDL, and NOEXCEPTION.
*
        INTEGER*4       OPTIONS(19),
        1  /18,1,0,1,0,0,1,2,0,0,0,0,0,0,0,0,0,1,0/

*            Set up statistics list as an array with the
*            number of statistics that requested.  There are
*            four: number of files, number of records, exception
*            records, and good records, in that order.
        INTEGER*4       STATSBLK(5) /4,0,0,0,0/

*            Declare the file names

        CHARACTER       IN_FILE*7 /'SEQ.DAT'/,
        1               OUT_FILE*12 /'CUSTDATA.DAT'/,
        1               FDL_FILE*11 /'INDEXED.FDL'/

*            Call the routines in their required order.

        STATUS = CONV$PASS_FILES (IN_FILE, OUT_FILE, FDL_FILE)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        STATUS = CONV$PASS_OPTIONS (OPTIONS)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        STATUS = CONV$CONVERT (STATSBLK)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

*            Display the statistics information.

        WRITE (6,1000) (STATSBLK(I),I=2,5)
1000    FORMAT (1X,'Number of files processed: ',I5/,
        1       1X,'Number of records: ',I5/,
        1       1X,'Number of exception records: ',I5/,
        1       1X,'Number of valid records: ',I5)

        END
```

Example 4–7 Using the CONVERT Routines in a COBOL Program

```
*       CONV.COB
*
*       This program calls the routines that perform the
*       functions of the Convert Utility.  It creates an
*       indexed output file named CUSTDATA.DAT from the
*       specifications in an FDL file named INDEXED.FDL.
*       The program then loads CUSTDATA.DAT with records
*       from the sequential file SEQ.DAT.  No exception
*       file is created.  This program also returns all
*       of the CONVERT statistics.
*
*       DATA NAMES:
*
*       IN-REC     defines the input record
*       OUT-REC    defines the output record
*       STATVALUE  receives the status value from the
*                  routine call
*       NORMAL     receives the value from SS$_NORMAL
*       OPTIONS    defines the CONVERT parameter list
*       STATSBLK   receives the CONVERT statistics.  The
*                  first data field (NUM-STATS) contains
*                  the total number of statistics requested.
*                  There are four:
*                  (1) number of files processed   (NUM-STATS)
*                  (2) number of records processed (NUM-FILES)
*                  (3) number of exception records (NUM-RECS)
*                  (4) number of valid records     (NUM-VALRECS)
*
IDENTIFICATION DIVISION.
PROGRAM-ID. PARTS.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-780.
OBJECT-COMPUTER. VAX-780.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT IN-FILE ASSIGN TO SEQ.
        SELECT OUT-FILE ASSIGN TO CUSTDATA.
DATA DIVISION.
FILE SECTION.
FD      IN-FILE
        DATA RECORD IS IN-REC.

01      IN-REC.
        02      IN-NUM        PIC X(4).
        02      IN-NAME       PIC X(20).
        02      IN-COLOR      PIC X(4).
        02      IN-WEIGHT     PIC X(4).
        02      SUPL-NAME     PIC X(20).
        02      FILLER        PIC X(28).

FD      OUT-FILE
        DATA RECORD IS OUT-REC.
```

Example 4–7 Cont'd. on next page

Example 4–7 (Cont.) Using the CONVERT Routines in a COBOL Program

```
01      OUT-REC.
        02      OUT-NUM       PIC X(4).
        02      OUT-NAME      PIC X(20).
        02      OUT-COLR      PIC X(4).
        02      OUT-WGHT      PIC X(4).
        02      SUPL-NAME      PIC X(20).

WORKING-STORAGE SECTION.
01      MORE-DATA-FLAGS       PIC X(3)      VALUE 'YES'.
        88      THERE-IS-DATA               VALUE 'YES'.
        88      THERE-IS-NO-DATA            VALUE 'NO '.

01      STATVALUE             PIC S9(9)     COMP.

01      OPTIONS               USAGE IS COMP.
        02      NUM-OPTS      PIC S9(9)     VALUE 18.
        02      CREATE        PIC S9(9)     VALUE 1.
        02      NOSHARE       PIC S9(9)     VALUE 0.
        02      FASTLOAD      PIC S9(9)     VALUE 1.
        02      NOMERGE       PIC S9(9)     VALUE 0.
        02      NOPPEND       PIC S9(9)     VALUE 0.
        02      XSORT         PIC S9(9)     VALUE 1.
        02      XWORKFILES    PIC S9(9)     VALUE 2.
        02      KEYS          PIC S9(9)     VALUE 0.
        02      NOPAD         PIC S9(9)     VALUE 0.
        02      PADCHAR       PIC S9(9)     VALUE 0.
        02      NOTRUNCATE    PIC S9(9)     VALUE 0.
        02      NOEXIT        PIC S9(9)     VALUE 0.
        02      NOFIXEDCTRL   PIC S9(9)     VALUE 0.
        02      NOFILLBUCKETS PIC S9(9)     VALUE 0.
        02      NOREADCHECK   PIC S9(9)     VALUE 0.
        02      NOWRITECHECK  PIC S9(9)     VALUE 0.
        02      FDL           PIC S9(9)     VALUE 1.
        02      NOEXCEPTION   PIC S9(9)     VALUE 0.
01      STATSBLK              USAGE IS COMP.
        02      NUM-STATS     PIC S9(9)      VALUE 4.
        02      NUM-FILES     PIC S9(9)      VALUE 0.
        02      NUM-RECS      PIC S9(9)      VALUE 0.
        02      NUM-EXCS      PIC S9(9)      VALUE 0.
        02      NUM-VALRECS   PIC S9(9)      VALUE 0.
PROCEDURE DIVISION.
MAIN.
      PERFORM CONVERT-FILE THRU DISPLAY-STATS.
      OPEN INPUT IN-FILE.
      READ IN-FILE
              AT END MOVE 'NO ' TO MORE-DATA-FLAGS.
      CLOSE IN-FILE.
      STOP RUN.

CONVERT-FILE.
      CALL 'CONV$PASS_FILES' USING BY DESCRIPTOR 'SEQ.DAT'
                                   BY DESCRIPTOR 'CUSTDATA.DAT'
                                   BY DESCRIPTOR 'INDEXED.FDL'
                             GIVING STATVALUE.
      IF STATVALUE IS FAILURE
      CALL 'LIB$STOP' USING BY VALUE STATVALUE.
```

Example 4–7 Cont'd. on next page

Example 4–7 (Cont.) Using the CONVERT Routines in a COBOL Program

```
      CALL 'CONV$PASS_OPTIONS' USING BY CONTENT OPTIONS
                               GIVING STATVALUE.
      IF STATVALUE IS FAILURE
      CALL 'LIB$STOP' USING BY VALUE STATVALUE.

      CALL 'CONV$CONVERT' USING BY REFERENCE STATSBLK
                          GIVING STATVALUE.
      IF STATVALUE IS FAILURE
      CALL 'LIB$STOP' USING BY VALUE STATVALUE.

DISPLAY-STATS.
      DISPLAY 'Number of files processed:   ',NUM-FILES CONVERSION.
      DISPLAY 'Number of records:           ',NUM-RECS CONVERSION.
      DISPLAY 'Number of exception records: ',NUM-EXCS CONVERSION.
      DISPLAY 'Number of valid records:     ',NUM-VALRECS CONVERSION.
```

For more information about calling the Convert routines, see the *VMS Utility Routines Manual.*

4.5 Summary of File-Creation Options

This section summarizes the file-creation options that are available using VMS RMS. File-creation options may be available as qualifiers or keywords to the OPEN statement and include various aspects of file creation, including file disposition, file characteristics, file allocation, and file positioning.

Note that the run-time options for opening files in conjunction with creating files are not included here, but they are described in Chapter 9.

4.5.1 File-Creation Options

The following chart lists the creation-time options that apply to specifying how an application uses a file.

Name of Option	Function
Create-if	Creates the file only if the directory does not contain a file with the same name. If a file with the same name exists in the directory, VMS RMS opens the existing file instead of creating a new file. FDL: FILE CREATE_IF VMS RMS: FAB$L_FOP FAB$V_CIF
Maximize version	Creates the file with the specified version number or a version number one greater than a file of the same name in that directory. FDL: FILE MAXIMIZE_VERSION VMS RMS: FAB$L_FOP FAB$V_MXV
Supersede version	Supersedes the file with the same name, type, and version number in the current directory. FDL: FILE SUPERSEDE VMS RMS: FAB$L_FOP FAB$V_SUP

Name of Option	Function
Temporary	Creates a temporary file (which has no directory entry) that is retained when the file is closed. The file can only be accessed if its internal file identifier is known (requires the use of a name block). Name blocks provide additional fields for extended file specifications. FDL: FILE DIRECTORY_ENTRY VMS RMS: FAB$L_FOP FAB$V_TMP
Temporary, delete	Creates a temporary file (which has no directory entry) marked for deletion. The file is deleted automatically when the file is closed. FDL: FILE TEMPORARY VMS RMS: FAB$L_FOP FAB$V_TMD

4.5.2 File Characteristics

The creation-time options that define file characteristics are described in the following chart:

Name of Option	Function
Block size	Defines the number of bytes to be used in each block (unit of I/O) throughout the life of this file. This file characteristic applies only to magnetic tape files. FDL: FILE MT_BLOCK_SIZE VMS RMS: FAB$W_BLS
Bucket size	Defines the number of blocks to be used in each bucket (unit of I/O) throughout the life of this file. This file characteristic applies only to relative and indexed files. FDL: FILE BUCKET_SIZE VMS RMS: FAB$B_BKS or XAB$B_BKZ
Date information	Specifies the date and time values for file backup, file creation, file expiration, and file revision. Can also set the number of file revisions. FDL: DATE attributes and FILE REVISION VMS RMS: Date and Time XAB fields
File organization	Defines the file organization: sequential, relative, or indexed. FDL: FILE ORGANIZATION VMS RMS: FAB$B_ORG
File protection	Defines the file protection for the file being created. FDL: FILE OWNER, FILE PROTECTION, FILE MT_PROTECTION VMS RMS: Protection XAB fields

Name of Option	Function
Fixed-length control field size	Defines the number of bytes in the fixed-length control field of a VFC record. FDL: FILE CONTROL_FIELD_SIZE VMS RMS: FAB$B_FSZ
Key characteristics	Defines the characteristics of a key in an indexed file, including key size, starting position, key type, bucket fill size, and key options. FDL: KEY attributes VMS RMS: Key Definition XAB fields
Maximum record number	Defines the maximum number of records for the file. Applies only to relative files. FDL: FILE MAX_RECORD_NUMBER VMS RMS: FAB$L_MRN
Maximum record size	Defines the maximum record size for all records in the file. Maximum record size refers to the size of all records in a file with fixed-length records, the size of the largest record with variable-length records, or the size of the variable-length portion of VFC records. A value of 0 with variable-length records means that there is no limit on the record size, except for magnetic tape files, for which a value of 0 sets an effective maximum record size equal to the block size minus 4. Variable-length records and VFC records must conform to certain physical limitations (see the *VMS Record Management Services Manual*). FDL: RECORD SIZE VMS RMS: FAB$L_MRS
Record attributes	Defines the type of record control information associated with each record. Records can be prevented from crossing block boundaries (FDL attribute RECORD BLOCK_SPAN) and can use one of the following carriage control conventions: • Each record is preceded by a line feed and terminated by a carriage return. • Each record contains a FORTRAN carriage return. • Each record is in print format where the two-byte fixed-length control field (VFC record format) of each record contains the carriage return. FDL: RECORD BLOCK_SPAN or RECORD CARRIAGE_CONTROL VMS RMS: FAB$B_RAT

Name of Option	Function
Record format	Defines the record format: • Fixed-length record format • Variable-length record format • VFC record format • Stream record format • Undefined record format (sequential files only) FDL: RECORD FORMAT VMS RMS: FAB$B_RFM

4.5.3 File Allocation and Positioning

You can specify file-allocation and positioning options with either the FAB control block or an allocation XAB (XABALL) control block. Note that any value specified in the XABALL control block overrides the corresponding value in the FAB. The creation-time options described below apply to file allocation and positioning.

Name of Option	Function
Allocation quantity	Allocates the file or area using the number of blocks specified by this value, rounded up to the nearest even multiple of the volume's cluster size. FDL: FILE ALLOCATION or AREA ALLOCATION VMS RMS: FAB$L_ALQ or XAB$L_ALQ
Areas	Allocates the file using single or multiple areas. Applies only to indexed files; sequential and relative files are always contained in a single area. Indexed files can be placed in specific areas, for example, to separate the data area from the index area. FDL: AREA number VMS RMS: XAB$B_AID
Contiguous	Allocates the file or area using a single extent. If the disk's unallocated space does not permit the file to be allocated contiguously, an error is returned. FDL: FILE CONTIGUOUS or AREA CONTIGUOUS VMS RMS: FAB$L_FOP FAB$V_CTG or XAB$L_AOP XAB$V_CTG

Name of Option	Function
Contiguous best try	Attempts to allocate the file or area using a minimum number of extents. If the file cannot be allocated contiguously, an error is not returned. FDL: FILE BEST_TRY_CONTIGUOUS or AREA BEST_TRY_CONTIGUOUS VMS RMS: FAB$L_FOP FAB$V_CBT or XAB$L_AOP XAB$V_CBT
Cylinder boundary	Allocates the file or area at the beginning of a cylinder boundary. FDL: AREA POSITION ANY_CYLINDER VMS RMS: XAB$B_AOP XAB$V_ONC
Cylinder position	Positions the file or area at the beginning of the specified cylinder number. FDL: AREA POSITION CYLINDER VMS RMS: XAB$B_ALN XAB$V_CYL and XAB$L_LOC
Default extension	Defines the minimum number of blocks for a file extension (extent) when additional disk space is needed. For EDIT/FDL file extension sizes, see Appendix A. FDL: FILE EXTENSION VMS RMS: FAB$W_DEQ or XAB$W_DEQ
Hard positioning	Directs VMS RMS to return an error if the requested file or area positioning or alignment cannot be performed. FDL: AREA EXACT_POSITIONING VMS RMS: XAB$B_AOP XAB$V_HRD
Logical block position	Positions the file or area at the beginning of the specified logical block. FDL: AREA POSITION LOGICAL VMS RMS: XAB$B_ALN XAB$V_LBN and XAB$L_LOC
Related file position	Positions the file or area as close as possible to a related file, at the specified virtual block. FDL: AREA POSITION FILE_ID or AREA POSITON FILE_NAME VMS RMS: XAB$B_ALN XAB$V_RFI and XAB$L_LOC
Virtual block position	Positions the file or area at the beginning of the specified virtual block. FDL: AREA POSITION VIRTUAL VMS RMS: XAB$B_ALN XAB$V_VBN and XAB$L_LOC

Name of Option	Function
Truncate end of file	Truncates a nonshared sequential file at its logical end to release the space between the logical end of the file (end of file data) and the physical end of the file (allocated file space) for other use. FDL: FILE_TRUNCATE_ON_CLOSE VMS RMS: FAB$V_TEF
Volume number	Indicates the volume set where the file or area is placed when it is created. FDL: AREA VOLUME VMS RMS: XAB$W_VOL

For the list of the run-time options that are common to creating and opening a file, see Chapter 9.

For more information about the options listed above, see Chapter 2. For more detailed information about the programming aspects of these options, refer to the *VMS Record Management Services Manual*.

5 Locating and Naming Files

When creating or opening a file, your program must provide the appropriate file specification. Typically, VAX languages require a file specification argument for an OPEN statement that names a file being created or locates a file being opened.

There are several ways to locate a file using a VMS file specification. The most direct way is to provide the complete file specification, which is often used when creating a new file. Another way to locate a file is to have the program supply the defaults so that the user enters only the file name component of a file specification to access the file.

Unlike small computer systems, which might have only one mass storage device, a VAX system may have many disk and magnetic tape devices. To eliminate having to always specify the device and directory in specifying a file, VMS RMS uses as defaults the process default device and directory.

5.1 Understanding File Specifications

A file specification on a VMS operating system consists of up to seven components, several of which assume a default value when they are not specified. To allow VMS RMS to identify the boundaries of each component, certain characters separate the components in a file specification. These characters mark the beginning or the end of a file specification component and must be supplied if a subsequent component is present. A complete file specification takes the following form:

node::device:[root.][directory-name]filename.type;version

The following table lists the characters that separate (begin or end) each component of a file specification.

Component	Separator Character(s)
Node	Double colon (::) ends a node name.
Device	Single colon (:) ends a device name.
Root	Square brackets ([ ]) or angle brackets (<>) delimit the root name. Note that a period (.) must terminate the root name.
Directory	Square brackets ([ ]) or angle brackets (<>) delimit the directory name. Within the directory component, a period (.) separates each directory and subdirectory name.
Filename	Period (.) begins the type component and ends the file name.
Type	Period (.) begins the type component and a semicolon (;) or a period (.) ends the type component.
Version	Period (.) or semicolon (;) following the the type component begins the version component.

Some examples of valid file specifications follow:

```
DISK1:[MYROOT.][MYDIR]FILE.DAT
DISK1:[MYDIR]FILE.DAT
[MYDIR]FILE.DAT
FILE.DAT;10
NODE::DISK5:[REMOTE.ACCESS]FILE.DAT
```

The maximum length of a file specification string is 255 characters, including all separator characters. The following table lists the length limits for each of the component parts of a file specification. Note that although the collective limit exceeds 255 characters, the overriding limitation is on the length of the file specification. For example, if you use the maximum number of characters allowed for a logical device name (255 characters), you cannot specify any other file specification component because the length of the file specification string exceeds the 255-character limit.

VMS RMS supports up to eight directory levels each for the root component and the directory component in the file specification.

Component	Number of Characters
Node	Up to 59 characters including an access control string (physical node names are up to 6 characters; logical node names are up to 15 characters)
Device	Up to 15 characters for a physical device name; up to 255 characters for a logical device name
Root	Up to 39 characters for each root name
Directory	Up to 39 characters for each directory and subdirectory name
Filename	Up to 39 characters
Type	Up to 39 characters
Version	Up to 5 digits, which optionally may be preceded by a hyphen (-)

Be careful when naming files that will be copied or accessed by remote systems. File name restrictions are generally determined by the file naming capabilities of the remote systems that require access to them. These restrictions must be considered part of the overall application design when network access is required.

5.1.1 File Specification Formats

Selecting a file specification format depends in part on whether you confine file activity to the local node or you conduct file activity on remote nodes. For example, to locate a file on the local node or VAXcluster, you do not have to include the node name in the file specification. Conversely, to locate a file on a remote node, the name of the remote node must be present either as the physical node name or as a logical name whose translation contains the physical node name. A logical node name can also contain access control information used to log in to the remote system.

The device can be identified with either a physical name or a logical name. You can terminate a physical device name or a logical device name with a colon and place one or more file specification components (directory name, file name, file type, and version) after it.

A logical device name may translate to another logical name, a physical device name, or a physical device name with additional file specification components. The logical name can be a search list, which specifies multiple file locations where the file can be found (see Section 5.2.3).

You only have to include the device name when specifying a record-oriented device, such as a terminal. However, if you choose to include other file specification components, you must follow the naming conventions described previously.

A logical name can be the file name component if it is the only component specified in the file specification. Refer to the *VMS DCL Concepts Manual* for additional information on defining logical names.

File specification formats for locating local and remote files are described in the remainder of this section.

5.1.1.1 Local Node

The following file specification format does not include a node name:

```
device:[root.][directory-name]filename.type;version
```

This is the general format of a file specification used to locate a file on the local node or VAXcluster.

The following file specification format is used only for ANSI-formatted magnetic tape volumes:

```
device:[directory-name]"quoted-ascii-a-string".;nn
```

Note that a null node name of the form "::" specifies the local node; this form overrides any default node names.

5.1.1.2 Remote Node

The following file specification formats are used for accessing files on remote nodes:

```
node::filespec
```

```
node"access-control-string"::filespec
```

The second file specification format includes an access control string. If an access control string is specified or if the process seeking to gain access to a remote file has a proxy login account on the remote node, the specified remote process uses its access rights to locate the file. If an access control string is not specified and a proxy account does not exist on the remote system, the local process may use the default DECnet account to locate the file.

The following file specification format is used to locate files on remote nodes:

```
node::"foreign-filespec"
```

The only action VMS RMS takes with the foreign file specification is to translate the logical node name, if applicable. This format is especially useful when the remote system is not a VMS operating system and the file specification does not conform to VMS file specification syntax conventions. Refer to the *VMS Networking Manual* for more information.

The following file specification format does not specify a file directly. Instead, it specifies a task on the remote system.

```
node::"task-spec-string"
```

For more information about specifying a logical node name or using any of the file specification formats and their associated syntax rules, refer to the *VMS DCL Concepts Manual*. The *VMS DCL Concepts Manual* also lists the characters from the DEC Multinational character set and the ASCII "a" characters that can be used in a quoted string for naming ANSI-labeled magnetic tape files.

5.1.2 Using File Specification Defaults

When you omit file specification components (except for the node name and root name), VMS RMS supplies default values for the missing components. The file specification to which defaults are applied is called the *primary file specification*. Your program can supply default values for all primary file specification components using either the *default file specification* or the *related file specification*. The process executing the program can supply specific default values for device and directory components.

Where applicable, VMS RMS substitutes the translated logical name to the primary file specification before it applies default values. After translating the primary file specification, VMS RMS first applies the defaults from the default file specification, then it applies the defaults from the related file specification, if relevant. VMS RMS then applies the process default values, where applicable, for the device and directory to obtain the full file specification it uses to locate the file.

VMS RMS applies process defaults to the device and directory components when a file specification does not include these components. Therefore, you must explicitly specify the device and directory if you want to access a file outside of your process-specified device and directory. At login, the process device and directory is set to the value established by the system-defined logical name SYS$LOGIN. VMS RMS obtains the current device by translating the logical name SYS$DISK, and it maintains the current directory in your process context.

For more information about the application of defaults, refer to Section 6.1.

5.2 Logical Names and Parsing

VMS RMS translates any logical name present in a file specification at run time. The use of logical names can be desirable for several reasons, including program simplification, device independence, file independence, and ease of use.

You can specify the file specification at compile (or assembly) time, or the program can prompt for it at run time. By specifying a logical name when you compile a program, you eliminate having to program a terminal input request, and you preserve the flexibility of being able to specify the input file before run time.

Device independence is more readily attainable if a logical name is used for the device name component. By using a logical name rather than explicitly specifying a physical device, an alternate device (usually containing a recent backup copy of the device) can be substituted by changing the definition of the logical name. Typically, device independence can reduce or eliminate the downtime caused by media failure or scheduled preventive maintenance.

Similarly, when you use a logical name and the current copy of a file is not available, an alternate file can be used. To locate several files in a defined search order, you can use a search list, which is a form of logical name. Alternatively, you can use wildcard characters to locate several files using one file specification; however, wildcard characters do not allow you to specify a search order.

Using a logical name to represent a complex file specification or a file specification component reduces keystrokes to save time and reduces the chance of error. For example, you could define a logical node name that translates to an actual node name and access control string for use when locating remote files. To keep the password a secret when you use this technique, the logical name should be defined interactively rather than in a command procedure.

5.2.1 Image Activation Using Logical Names

When VMS activates an image, it uses VMS RMS to open the image file. If the program specifies the image file with a logical name, VMS RMS uses the equivalence name to look up the image in the *known file* list, unless the file specification includes a version number delimiter (a semicolon (;) or a period (.)). Known files are files that are installed using the Install Utility, and the known file list provides a listing of these files by name and by number.

If VMS RMS finds the file in the known file list, it uses the file number to access the file directly on disk and bring it into memory for execution. If the specified image file is not in the known file list, VMS RMS must go through the time-consuming process of looking through the disk directories to find the file.

If you create a new version of an image but do not install it as a known image and do not remove the old version of the image from the known file list, the new image will not run.

Similarly, when you use a search list to specify the image, the known file lookup takes precedence. Until a lookup is successful or until the search list is exhausted, VMS RMS executes a known file lookup for each element on the search list that does not include a file version delimiter. If it exhausts the search list, VMS RMS uses the search list again, this time trying to locate and open the image file on disk.

If an older version of the image is included in the search list and if VMS RMS finds the older version first, it will execute the older version and never look for the new version. Be sure to consider this when using search lists.

5.2.2 Example Use of Logical Names

Regardless of the programming language, you can use a logical name to provide components of a file specification. The following program example shows how to access a remote file. You access a remote file in the same way that you access a local file, except that the remote file specification includes a node name.

Example 5–1 is a simple FORTRAN program that transfers a remote file on node TRNTO to the line printer on node BOSTON, using the logical names SRC and DST. You must define the logical name for the process before you run the program, using the following sequence of commands:

```
$ DEFINE SRC TRNTO::USER:[STOCKROOM.PAPER]INVENTORY.DAT
$ DEFINE DST BOSTON::LPAO:
$ RUN TRANSFER
```

In Example 5–1, standard I/O calls transfer the file's records from one device to another. Note the use of the VMS file specification format with a remote node name. (If the remote node is other than VMS, the format of the file specification may differ.)

After opening the files and copying all the records, the program closes the channels, thereby terminating network operations. These operations are similar for applications in the other high-level VAX languages.

Example 5–1 Using Logical Names for Remote File Access

```
        PROGRAM TRANSFER
C
C       This program creates a sequential file with variable-length
C       records from a sequential input file. The input and output
C       files are identified by the logical names SRC and DST,
C       respectively.
C
        CHARACTER BUFFER*132
C
100     FORMAT (Q,A)
200     FORMAT (A)
C
C       Open the input and output files.
C
        OPEN (UNIT=1,NAME='SRC',TYPE='OLD',ACCESS='SEQUENTIAL',
     1      FORM='FORMATTED')
        OPEN (UNIT=2,NAME='DST',TYPE='NEW',ACCESS='SEQUENTIAL',
     1      FORM='FORMATTED',CARRIAGECONTROL='LIST',
     2      RECORDTYPE='VARIABLE')
C
C       Transfer records until end-of-file or other error condition.
C
10      READ (1,100,END=20,ERR=20) NCHAR,BUFFER(:NCHAR)
        WRITE (2,200) BUFFER(:NCHAR)
        GOTO 10
C
C       Close the input and output files.
C
20      CLOSE (UNIT=2)
        CLOSE (UNIT=1)
        END
```

5.2.3 Types of Logical Names

When a logical name is defined, you can assign it various translation attributes including the *concealed* attribute and the *terminal* attribute. By default, a logical name is neither concealed nor terminal.

To specify a logical name as either concealed or terminal, use the /TRANSLATION_ATTRIBUTES qualifier for the DCL commands DEFINE or ASSIGN.

The terminal attribute indicates to VMS RMS that the related logical name is the final name in the translation process. That is, no further translation is to be performed.

The concealed attribute ensures that VMS RMS uses the device logical name when communicating with the application program. If the device logical name does not have the concealed attribute, any file specification information returned to the application program includes the device's physical name rather than its logical name. To illustrate, enter the following command sequence:

```
$ DEFINE/SYSTEM USERDISK DUA5:
$ SET DEFAULT USERDISK:[JONES]
$ DIRECTORY
```

The system responds with the following display, which identifies the device by its physical name (DUA5):

```
DIRECTORY DUA5:[JONES]

FILE.TXT;1          FILE.TXT;2

Total of 2 files.
```

Now enter the following command sequence:

```
$ DEFINE/SYSTEM/TRANSLATE=CONCEALED USERDISK DUA5:
$ DIRECTORY
```

The system responds with the following display, which identifies the device by its logical name (USERDISK):

```
DIRECTORY USERDISK:[JONES]

FILE.TXT;1          FILE.TXT;2

Total of 2 files.
```

A search list is a logical name that contains more than one file specification. Typically a search list is used to search multiple file locations looking for a file. VMS RMS attempts to locate the file by using the first file specification in the search list, then the next, and so forth until the file is found or the search list is exhausted. Like other logical names, a search list is usually defined using the ASSIGN or DEFINE commands; however, in a search list logical name, the multiple file specifications (equivalence names) must be separated by commas.

Any of the equivalence names in the search list may be specified individually as being terminal or being concealed. Section 6.2 describes the use of search lists and wildcard characters for multiple file processing and parsing. For general information about using logical names, refer to the *VMS DCL Concepts Manual.*

5.2.4 Introduction to File Parsing

VMS RMS allows an application program to specify defaults for the device and directory components of a file specification as well as other components of a file specification. The method VMS RMS uses to apply defaults and translate any logical names present is called *file parsing.* In effect, VMS RMS merges the various default strings (after translating any logical names) to generate the file specification used to locate the file.

One of the functions of file parsing is to determine when a logical name is present and whether the file specification describes a file on the local node. If a node name is not present in the file specification (the file is located on the local system), VMS RMS first translates any logical names, applies defaults to any missing components, and then attempts to locate the file.

If a node name is present, VMS RMS does not process the file specification on the local node. Instead, it merges any program-specified defaults without translation and passes the defaulted, untranslated file specification to the file access listener (FAL) at the remote node; the operating system on the remote node interprets it.

With advanced file parsing, a *single* file specification can be used to locate a single file or multiple files. To locate a single file, multiple file locations or file names can be searched to ensure that the file is found. The multiple file locations or file names can be located in the same or in different directories, on different devices, on different nodes, or a combination thereof. Using wildcard characters and search lists, you can locate multiple files with a single file specification.

When a wildcard character or a search list is included in a file specification, the application program may need to preprocess the file specification before attempting to locate the file. A VMS RMS file service that operates on an unopened file (such as the Create service and the Open service) performs the following file-parsing tasks:

- Examines a file specification for validity
- Translates any logical names present
- Applies defaults
- Attempts to locate the file

If a name block is present, the service may also do the following file-parsing tasks:

- Returns the actual complete file specification used to access the file and its associated file identifier
- Returns the length of each component of a file specification as well as other information about the file specification

Some file services, including the Open and Create services, cannot process a file specification that contains wildcard characters. If a file specification contains wildcard characters, you must use the Search service to resolve the wildcard characters before you invoke the Open service or the Create service.

The Parse service determines whether wildcard characters or search lists are present, and it initializes control block fields that are necessary to search for multiple files using the Search service. To use the Search service, a NAM (name) block must be present when the Parse service is invoked.

Alternatively, you can use the SYS$FILESCAN system service (scan string for file specification) to scan a file specification for validity and optionally return the lengths of the individual file specification components without translating logical names or applying defaults. Two Run-Time Library routines, LIB$FIND_FILE and LIB$FILE_SCAN, perform functions that are similar to the Parse service and the SYS$FILESCAN system service.

For more information about how VMS RMS parses a file specification, see Section 6.1. For additional information about using directory specifications, including directory syntax conventions, see Section 6.3.

5.3 Using One File Specification to Locate Many Files

Five services can translate and apply defaults to a file specification to produce a fully parsed file specification: the Create, Open, Erase, Parse, and Rename services. Other file services must be preceded by one of these services to parse the file specification and, in some cases, to open the file.

If a file specification contains one or more wildcard characters, it must be preprocessed using the Parse and Search services before the file can be located. The Parse service sets bit values in the name block file name status bits field (NAM$L_FNB). This field can be tested to determine whether a wildcard character or a search list logical name is present. The Search service locates a file and specifies its name (without wildcard characters). If wildcard characters are present, you must first invoke the Search service before processing (opening or creating) the file; if wildcard characters are not present, the file can be processed without invoking the Search service.

To process a single file, you need to invoke the Search service only once; to process many files, invoke the Search service as many times as needed to return the next full file specification. When no more files match the file specification, the Search service returns a no-more-files-found message (RMS$_NMF).

In summary, the Parse and Search services work together to provide a fully qualified file specification that the Search service uses to locate the file.

Your program can process a single file without using the Search service if neither the file specification nor the search list contain wildcard characters. If any of the file specifications in a search list contain wildcard characters, the Search service must be invoked before processing the file to prevent an invalid wildcard completion status error. If a wildcard character is present in the second or subsequent file specifications in a search list, VMS RMS does not set the wildcard bit in the file name status bits field.

If the Parse and Search services precede an Open service, an open-by-name-block operation should be performed by specifying the address of the name block in the name block address (FAB$L_NAM) field and setting the file-processing options (FAB$L_FOP) open-by-name-block (FAB$V_NAM) bit option.

Wildcard characters cannot be present in the file specification when the Create service is invoked. Sometimes the Parse service and the Search service precede a Create service.

When the create-if option bit (FAB$V_CIF) or the supersede option bit (FAB$V_SUP) is set in the file-processing options (FAB$L_FOP) field, the program may invoke the Parse service to check for wildcard characters or search lists in the file specification. If a search list or wildcard characters are found, the program must invoke the Search service before invoking the Create service.

The create-if option tries to open any file found in the search list. If the file is not found in the search list, VMS RMS creates it using the first file specification in the search list. If these options are specified and a wildcard character is present when the Create service is invoked, the file specification is invalid; if a search list is present, the file is created using the first file specification from the search list.

You can either call these services directly from a VAX MACRO procedure (or as part of a USEROPEN or USER_ACTION routine in a high-level language) or execute the calls from VAX language subroutines or functions that call the VMS RMS services. The Parse and Search services require that a name block be present. Unless your language supports a means of setting values in a name block (and other control blocks) and invoking VMS RMS services, you should use a VAX MACRO procedure. FDL does not support the use of a name block.

In addition to a name block, you usually need a file access block (FAB) and a record access block (RAB). To perform file services, a FAB (and, if needed, extended attribute blocks (XABs)) must be present; to perform record services, a RAB must be present.

The following program shows how to use the LIB$FIND_FILE routine to locate the desired file, which the interactive user enters. Because LIB$FIND_FILE is used with the supplied arguments, the file specification may contain wildcard characters, a search list, and a search list that assumes the program will allow the use of "sticky" defaults, as in DCL command line parsing. The routine is called by the following VAX BASIC program USEROPEN option for the BASIC OPEN statement.

```
100 MAP (REC.1) SURNAME$ = 20%, REST$ = 60%
110 OPEN " " FOR OUTPUT AS FILE #1%, ORGANIZATION RELATIVE, &
         MAP REC.1, USEROPEN LOCATE
120 CLOSE #1%
130 END
```

The BASIC program allocates the control blocks before control is given to the USEROPEN routine; it also passes the address of the FAB as the first argument and the address of the RAB as the second argument. These arguments enable the VAX MACRO routine to obtain the control block addresses because the argument pointer points to the longword count of arguments, followed by the longword-length arguments. Because the VAX MACRO macros $FAB and $NAM are not used, access to the symbolic offset values defined for these control blocks are not available; thus, the $FABDEF, $NAMDEF and $RABDEF macros define these symbols for the USEROPEN routine.

In addition to locating the file using any valid file specification, the called routine also connects to the file requesting 15 global buffers (or as many global buffers as system resources permit). This routine is linked with the BASIC program to form the executable image. Example 5–2 shows the routine.

Example 5–2 Selecting the USEROPEN Option to Call a Routine

```
        .TITLE  LOCATE
        .PSECT  DATA,WRT,NOEXE
        .EXTERNAL LIB$SIGNAL,LIB$STOP,LIB$GET_INPUT,LIB$PUT_OUTPUT
        .EXTERNAL STR$GET1_DX
        $FABDEF                              ; Define FAB symbols
        $RABDEF                              ; Define RAB symbols
;
IFILE:  .BLKB   80                           ; Input filespec
IFILED: .LONG   80                           ; Filespec descriptor
        .LONG   IFILE
;
OFILED: .WORD   255                           ; Filespec descriptor
        .BYTE   DSC$K_DTYPE_T                 ; Specify character text
        .BYTE   DSC$K_CLASS_D                 ; Specify descriptor class
OFILE:  .LONG   0                             ; Address set by STR$GET1_DX
;
DFILED:  .ASCID   /.DAT/                      ; Default filespec descriptor
;
PROMPT: .ASCID  /Enter the filespec: /        ; User prompt
LOC_P:  .ASCID  /*** NOTE: Global buffers unavailable ***/  ;
NULL_P: .ASCID  / /                           ; Blank line prompt

ARGS:   .LONG   7                             ; 7 arguments
        .ADDRESS IFILED                       ; Input filespec
        .ADDRESS OFILED                       ; Output filespec
        .ADDRESS CTEXT                        ; Context
        .ADDRESS DFILED                       ; Default filespec
        .ADDRESS NULL                         ; No related filespec
        .ADDRESS STV_L                        ; STV field
        .ADDRESS UFLAGS                       ; User flags
CTEXT:  .LONG   0                             ; Context work area
NULL:   .LONG   0                             ; No related filespec
STV_L:  .BLKL   1                             ; STV status return area
UFLAGS: .BLKL   1                             ; User flags
LEN:    .BLKB   255
                                              ;
        .PSECT  CODE,NOWRT,EXE
        .ENTRY  LOCATE,^M<>
                                              ;
        MOVL    4(AP),R6                      ; Move FAB address into R6
        MOVL    8(AP),R7                      ; Move RAB address into R7
        BISL2   #2,UFLAGS                     ; Set flag for sticky defaults
TERR:   PUSHAL  IFILED                        ; Get input length
        PUSHAL  PROMPT                        ; Prompt for input
        PUSHAL  IFILED                        ; Input descriptor
        CALLS   #3, G^LIB$GET_INPUT           ; Get input
        BLBC    R0,TERR                       ; Retry on error
        PUSHAL  OFILED                        ; Push descriptor address
        PUSHAL  LEN                           ; And length
        CALLS   #2, G^STR$GET1_DX             ; Allocate dynamic string
        BLBC    R0,ERR                        ; Branch on error
        CALLG   ARGS, G^LIB$FIND_FILE         ; Call RTL Find File Routine
        BLBC    R0,ERR                        ; Branch on error
        BRW     OPEN                          ; Skip on success
ERR:    PUSHL   STV_L                         ; Signal error status
        PUSHL   R0                            ; codes
        CALLS   #2, G^LIB$SIGNAL              ; Display error
        BRW     TERR                          ; Reenter filespec on error
```

Example 5–2 Cont'd. on next page

Example 5–2 (Cont.) Selecting the USEROPEN Option to Call a Routine

```
OPEN:
        PUSHAL   OFILED                      ; Display filespec
        CALLS    #1, G^LIB$PUT_OUTPUT        ; on screen
        MOVL     OFILE,R10                   ; Move filespec address to R10
        $FAB_STORE FAB=R6,FNA=(R10),FAC=GET,-
         FNS=OFILED,SHR=<GET,MSE>            ; Set read-sharing global buffer
        $OPEN    FAB=R6                      ; Open the file
        BLBS     R0,CONNECT                  ; Branch on success
        PUSHL    FAB$L_STV(R6)               ; Push STV and STS in reverse
        PUSHL    FAB$L_STS(R6)               ; order on stack to
        CALLS    #2, G^LIB$STOP              ; Signal error and stop
;
; This block of code attempts to Connect with global buffers if possible
; and uses local buffers if global buffer resources are not available.
; Because the global buffer value is set between the Open and Connect,
; all defaults are overwritten.
;
CONNECT:
        MOVL     #15,R9                      ; R9 contains global buffer count
        BRB      RETRY                       ; Skip local buffer handing
LOCAL:  MOVL     #0,R9                       ; Turn off global buffers
        $RAB_STORE RAB=R7,MBF=#6             ; Request 6 local buffers
        PUSHAL   LOC_P                       ; Inform user
        CALLS    #1, G^LIB$PUT_OUTPUT        ; No global buffers
RETRY:  $FAB_STORE FAB=R6,GBC=R9             ; Override default global buffer
        $CONNECT RAB=R7                      ; Connect the record stream
        BLBC     R0,RERR                     ; Branch on error
        BRW      DONE                        ; On success, return
RERR:   CMPL     R0,#RMS$_CRMP               ; Test if too many global buffers
        BNEQ     CERR                        ; Quit if other error
        CMPL     #4,R9                       ; Test if too few global buffers
        BLSS     LOCAL                       ; Use local buffers
        SUBL2    #3,R9                       ; Decrement R9 by 3
        BRW      RETRY                       ; Attempt Connect again
CERR:
        PUSHL   RAB$L_STV(R7)                ; Push STV and STS in reverse
        PUSHL   RAB$L_STS(R7)                ; order on stack to
        CALLS   #2, G^LIB$STOP               ; Signal and end on error
DONE:   RET                                  ; Return to main program
        .END
```

Example 5–2 also shows the proper way to signal errors. The RAB$L_STS (completion status) field and the RAB$L_STV (additional status values) field of the FAB or RAB are used so that secondary completion information is displayed, if appropriate, by the LIB$SIGNAL or LIB$STOP routines. The VAX MACRO program shown in Example 5–3 invokes the Parse service, determines whether a wildcard character or search list is present, and conditionally branches to a sequence of instructions that invoke the Search service followed by the Open service. The resultant string is displayed after the file is opened.

For more information about the LIB$ routines shown here and other routines in the VMS Run-Time Library, see the *VMS Run-Time Library Routines Volume*.

Example 5–3 Using the Parse, Search, and Open Services

```
        .TITLE  WILDFILE
;
;   BEGIN DATA PROGRAM SECTION  * * * * * * * * * * * * * * * * *
;
        .PSECT  DATA,NOEXE,WRT
MY_NAM: $NAM    RSA=RES_STR,-           ; Result buffer address
                RSS=NAM$C_MAXRSS,-      ; Result buffer size
                ESA=EXP_STR,-           ; Expanded buffer address
                ESS=NAM$C_MAXRSS,-      ; Expanded buffer size
MY_FAB: $FAB    FOP=NAM,-               ; Use NAM block option
                NAM=MY_NAM,-            ; Pointer to NAM block
                FNA=INP_STR             ; Address of file name string
EXP_STR:                                ; Expanded string buffer
        .BLKB   NAM$C_MAXRSS
RES_STR:                                ; Resultant string buffer
        .BLKB   NAM$C_MAXRSS
RES_STR_D:                              ; Resultant string descriptor
        .BLKL   1
        .LONG   RES_STR
INP_STR:                                ; Input string buffer
        .BLKB   NAM$C_MAXRSS
INP_STR_D:                              ; Input string descriptor
        .LONG   NAM$C_MAXRSS
        .LONG   INP_STR
INP_STR_LEN:                            ; Input string length
        .BLKL   1
PROMPT_D:                               ; User prompt string
        .ASCID  /Please enter the file specification: /
;
;       BEGIN CODE PROGRAM SECTION  * * * * * * * * * * * * * * * * * *
;
        .PSECT  CODE,EXE,NARRATE
        .ENTRY  WILDFILE, ^M<>          ; Save no registers
        PUSHAB  INP_STR_LEN             ; Address for string length
        PUSHAB  PROMPT_D                ; Prompt string descriptor
        PUSHAB  INP_STR_D               ; String buffer descriptor
        CALLS   #3,G^LIB$GET_INPUT      ; Get input string value
        BLBS    R0,MOVE                 ; Branch on success
        BRW     EXIT                    ; Quit on error
;
; Store user input string and perform parse
;
MOVE:   MOVB   INP_STR_LEN, -           ; Set string size
                 MY_FAB+FAB$B_FNS
```

Example 5–3 Cont'd. on next page

Example 5–3 (Cont.) Using the Parse, Search, and Open Services

```
PAR:     $PARSE  FAB=MY_FAB                ; Parse filespec in MY_FAB
         BLBC    R0,F_ERR                  ; Branch on error
         BBS     #NAM$V_WILDCARD,-
                  NAM$L_FNB+MY_NAM,WILD    ; Branch on bit set
         BBS     #NAM$V_SEARCH_LIST,-
                  NAM$L_FNB+MY_NAM,WILD    ; Branch on bit set
         BRB     OPEN                      ; OK to open file
WILD:    $SEARCH FAB=MY_FAB                ; Search for next file
         BLBC    R0,F_ERR                  ; Branch on error
OPEN:    $OPEN   FAB=MY_FAB                ; Open file
         BLBC    R0,F_ERR                  ; Branch on error
         MOVZBL  MY_NAM+NAM$B_RSL,-        ; Move resultant string
                  RES_STR_D                ; length to descriptor
         PUSHAB  RES_STR_D                 ; String buffer descriptor
         CALLS   #1,G^LIB$PUT_OUTPUT       ; Display resultant filespec
                                           ; Connect and process file
                                           ; .
                                           ; .
                                           ; .
                                           ; ... and
         $CLOSE FAB=MY_FAB                 ; Close file
         BLBS   R0,EXIT                    ; Branch on success
F_ERR:   PUSHL  MY_FAB+FAB$L_STV           ; Push STV and STS on stack
         PUSHL  MY_FAB+FAB$L_STS           ; in reverse order
         CALLS  #2, G^LIB$SIGNAL           ; Signal error
EXIT:    RET                               ; Exit with R0
         .END WILDFILE
```

Example 5–3 uses the VAX MACRO macros $FAB and $NAM that define the control blocks and specify the arguments for the Parse, Search, Open and Close services. It shows how to preprocess a file specification using the Parse and Search services. To process many files, you could add an unconditional branch instruction just before the symbolic address F_ERR to branch to the $SEARCH macro at the symbolic address WILD.

Refer to the *VMS Record Management Services Manual* and the *VAX MACRO and Instruction Set Reference Manual* for more VAX MACRO RMS examples and information about using VAX MACRO.

An application may also need to process either one file or many files, depending on the file specification that the terminal user enters or the logical name that is provided (if the program uses a logical name in its file specification). Each of these cases is discussed in the following sections.

5.3.1 Processing One File

When only a single file needs to be processed, but more than one location for the file may need to be searched, you can usually find the file by specifying a file specification that contains a search list.

For example, consider the case of a directory that contains the file PAY.DAT and a backup copy of the file named PAY_BUP.DAT. You could specify a file name of PAY*.DAT in the file specification and invoke the Parse service once and the Search service once to locate either of the two files; this method will locate PAY.DAT before PAY_BUP.DAT.

A potential problem arises if the file PAY.DAT has been deleted or renamed. In this case, unless the program determines that the file specification is one of several that are acceptable, any file named PAY that has the file type DAT could be accessed: for example, PAY_ACC.DAT. You can avoid such problems by defining a search list logical name that specifies that VMS RMS search for PAY.DAT and PAY_BUP.DAT. A search list named SEARCH could be defined as follows for the directory [SMITH]:

```
$ DEFINE SEARCH [SMITH]PAY.DAT,[SMITH]PAY_BUP.DAT
```

To locate the file, specify SEARCH as the primary file specification.

When the file locations to be searched reside in different directories of a directory tree, you can use the ellipsis wildcard character in the directory field to search all subdirectories. Alternatively, you could define a search list that searches for the file PAY.DAT in one directory, the same file name in a subdirectory, and PAY_BUP.DAT in any directory in the directory tree by using the following DEFINE command:

```
$ DEFINE SEARCH [SMITH]PAY,[SMITH.PAY]PAY,[SMITH...]PAY_BUP
```

You use the file specification SEARCH:.DAT to locate the desired file. In this example, note that one of the search list file specifications contains wildcard characters. Wildcard characters can be used in a search list if they are needed, just as with any other logical names and file specifications. However, the Parse and Search services must be used to locate the correct file.

When you need to locate files in different directory trees (or top-level directories), include complete directory specifications in your search list definition. For example, to locate the file TEST_DATA.DAT in the device/directory combinations of DISK1:[SMITH], DISK2:[STATS], or DISK2:[SMITH] you could use the following command to define the search list TST:

```
$ DEFINE TST DISK1:[SMITH],DISK2:[STATS],DISK2:[SMITH]
```

You can also use search lists to locate files on different devices. To locate this file, you specify TST:TEST_DATA.DAT.

To find the same directory and the same file name on different devices, you could use the following command to define TST:

```
$ DEFINE TST DISK1:,DISK2:,DISK3:
```

When you define the search list TST in this manner, you can locate the file by using the search list to specify the device name. In this way, you can use a single search list to locate files that would otherwise require multiple file specifications, even if wildcard characters were used.

5.3.2 Processing Many Files

To process many files using a single file specification, you always need to use the Parse and Search services to locate the files.

The application requirements and the directory location of the files generally determine whether one or more search lists, wildcard characters, or search lists containing wildcard characters are used in the file specification. When files must be accessed in nonalphabetical order, use a search list.

To process multiple files using a single file specification, invoke the Parse service (or its equivalent) once to interpret the file specification and to create the file specification pattern to be searched. After the file specification is parsed, you can invoke the Search service to locate each file that matches the original file specification. In some cases, you can examine (or display) the resultant file specification string returned by the Search service to determine if you (or the interactive user) want to process (open) the file.

If you want to list all file specifications that match a particular file specifcation and let the terminal user choose each file to be processed, wildcard characters can be used safely, possibly in a search list that contains wildcard characters in one or more of its file specifications. To reduce the number of files that the user might choose to process, use a search list without wildcard characters or rely less on wildcard characters. For example, to locate all files in a directory tree on different devices with a file type of DAT, you could define the search list TREE as follows:

```
$ DEFINE TREE DISK1:[MYDIR...],DISK2:[MYDIR...]
```

The primary file specification that would be used for the Parse service would be TREE:*.DAT. A great number of files might match this.

For applications that will need to locate certain files, search lists with limited use of wildcard characters might be needed. Consider a file that contains a prefix of RESULTS followed by the date for which the data applies. You could use the file name RESULTS*JUN*.DAT to locate a record that was entered in the month of June by executing a Search service followed by an Open service for each file, reading all records until the correct one is found, and invoking the Close service after processing each file.

A search list should be used when a predefined group of files is processed by a program that is not intended to be interactive. Using a search list is particularly desirable if the files have unrelated file names or if they are located on different directories or devices. A search list also minimizes processing time by searching for a definite group of files.

5.3.3 Processing One or Many Files

For general-purpose applications, when the user enters a file specification that may indicate one file or many files, there is a means of testing whether one file or many files are to be processed, or to explicitly disallow the use of wildcard characters for applications where only a single file should be processed. To test for wildcard characters or search lists, or both, invoke the Parse service and test the appropriate bits in the NAM$L_FNB field.

The presence of a wildcard character usually indicates that many files should be processed, depending on program conventions. If a search list is present, it may or may not indicate that only one file should be processed and a convention is needed for users of that program. Thus, by testing whether a wildcard is present, the program can either invoke the Parse service once and the Search service repeatedly for each file to be opened, or it can disallow wildcard characters and request that the file specification be reentered. In some cases, the program may need to disallow the use of a search list or allow one or many files to be accessed, depending upon application conventions.

If you want to disallow wildcard characters, invoke the Open service. The Open service fails when it encounters a wildcard character.

6 Advanced Use of File Specifications

This chapter is intended for readers who want to better understand how VMS RMS internally applies defaults, parses file specifications, and handles directory specifications. This chapter also describes the use of rooted-directory syntax and process-permanent files.

6.1 How VMS RMS Applies Defaults

This section describes how VMS RMS applies defaults when it tries to locate a file specified by your program.

The program-supplied file specifications are the primary file specification, the default file specification, and one or more related file specifications. Of these, the primary file specification is usually specified.

The default file specification contains a default for the type component (typically DAT to specify a data file, TXT to specify a text file, and so forth) or it supplies defaults for other file specification components. The related file specification is used when two files are involved in an operation, such as copying or merging files, in which the input file specification is the related file specification for the output file.

A final default mechanism ensures that if the device or directory components, or both, are missing, process defaults are used. Table 6–1 describes the defaults that VMS RMS uses to produce a complete file specification when file specification components are omitted.

Table 6–1 File Specification Defaults

File Specification	Description
Primary	If the device field is a logical name, VMS RMS translates the logical device name to its component parts. The resulting device name may be a physical device name, a process-permanent file name, or another logical name.
Default	If the device field is a logical name, VMS RMS translates it before defaults are applied. If any of the fields in the file specification from the previous step are missing, they are supplied from the corresponding fields in the translated default file specification, where applicable.

Table 6–1 (Cont.) File Specification Defaults

File Specification	Description
Related	If the device field is a logical name, VMS RMS translates it and applies the default values before it uses the related file specification to add missing component fields. If fields contain wildcard characters, the wildcard characters remain in the fields. When VMS RMS uses the related file specification to specify an output file, the file name field and the file type field are replaced by the corresponding related file specification fields, where applicable. For more information, including the use of multiple related file specifications, see Section 6.2.3.
Device and Directory	If the device name is omitted, the device field and, optionally, the directory field accept the system logical name SYS$DISK. If VMS RMS cannot translate the logical name SYS$DISK to a physical device name, an error occurs. If the directory field does not accept the logical name SYS$DISK, it accepts the name of the current process default directory.

Primary, default, and related file specifications can use logical names. VMS RMS translates the primary file specification before it applies defaults and missing components. VMS RMS also translates the default file specification before using the default values. Finally, VMS RMS translates the related file specification before it uses missing components supplied by the related file specification. If the file specification is still missing the device or directory name components, the process executing the program supplies default device and directory values.

The algorithm used in determining the appropriate translation is as follows:

```
if node name present
   then translate node name
else if device name present
   then translate device name
else if only file name present
   then translate file name
```

For the remainder of this description, the component parts of the file specification are referred to as strings. For example, the device component is referred to as the device string; the name component is the name string, and so forth. Furthermore, as components are added to a file specification, the expanded file specification is referred to as the expanded string. Finally, the resultant file specification is called the resultant string.

Table 6–2 shows the sequence in which defaults are applied to a file specification (primary file specification string) and the resulting file specification (resultant string). In Table 6–2, the program specifies the primary file specification string FILE, omitting all other components of the file specification. The default file specification string .DAT provides the file type component. The related file specification string does not provide any component strings, but the default device string (logically SYS$DISK) provides the device string DISK1: and the directory string, [INV_C], is provided by the default directory string. Finally, because the resultant string is used to specify a new file, VMS RMS applies the version number **1** to complete the new file specification.

Table 6–2 Example of Applying Defaults

String Name	String Applied	Expanded String
Primary file specification	FILE	FILE
Default file specification	.DAT	FILE.DAT;
Related file specification	None.	FILE.DAT;
Default device (SYS$DISK)	DISK1:	DISK1:FILE.DAT;
Default directory	[INV_C]	DISK1:[INV_C]FILE.DAT;
Resultant string		DISK1:[INV_C]FILE.DAT;1

VMS RMS appends the version number to the expanded string to convert it into the resultant string. The resultant string is the resultant file specification that VMS RMS uses to locate the file.

When coding the file specification information in a program, you can use the language keyword for the OPEN (or CREATE) statement. Then you use the FDL Editor to enter the file specification characteristics. Finally, you call the FDL$CREATE routine to create a file, or you call the FDL$PARSE routine and the FDL$RELEASE routine to open a file.

Alternatively, you can set the appropriate control block fields and call the VMS RMS services directly, perhaps as part of a USEROPEN routine or a USER_ACTION routine.

Consider a program that does not explicitly specify the device and directory in any of the file specifications and does not have a related file specification. VMS RMS adds the current process default device and the current process default directory to the expanded string after it applies components provided by the default file specification. However, if the program looks for a data file that is not in the current process default device and directory, it does not find the file. In this case, the solution is to specify the data file's device and directory either in the primary file specification, the default file specification, or the related file specification.

The program-supplied file specifications can be specified using the methods summarized in the following chart:

File Specification	How You Can Specify It
Primary	Use the FDL attribute FILE_NAME; use the file name or the name following the FILE, FILE_ID, or FILENAME keywords in the OPEN statement in certain VAX languages; or use the string pointed to by the FAB field FAB$L_DNA.
Default	Use the FDL attribute FILE DEFAULT_NAME; use the default file specification or the name following the DEFAULTNAME or DEFAULT_FILE_ID keyword in the OPEN statement in certain VAX languages; or use the string pointed to by the FAB field FAB$L_DNA.
Related	Use the name block (NAM) pointed to by the NAM$L_RLF field; the related name block must specify the location of a file specification, which must be pointed to by the NAM field NAM$L_RSA.

Specifying all components in the primary file specification explicitly decreases the chance of error. However, defaults are provided and can be very useful, especially for general-purpose applications and for applications in which the file specification is entered by the interactive user. Another option to consider is the use of logical names.

See the appropriate languages documentation for information about language statements and their keywords. Consult the *VMS File Definition Language Facility Manual* for information about the FDL Editor, and refer to the *VMS Utility Routines Manual* for information about the FDL$PARSE and FDL$RELEASE routines. For detailed information about VMS RMS control blocks and services, see the *VMS Record Management Services Manual.*

6.2 Understanding VMS RMS Parsing

In the following text, the term *expanded string* refers to the user-allocated string pointed to by the name block expanded string address (NAM$L_ESA) field; the term *equivalence string* refers to the internal area VMS RMS uses to store the result of a logical name translation.

As it processes each program-supplied file specification, VMS RMS translates the file specification into its component parts. Any component present in the primary, default, or related file specifications is used to form the resultant file specification, which VMS RMS uses to locate the file. If a name block is present and the address and size of the expanded string are specified, the file specification is copied into the expanded string, which is used to store the various intermediate forms of the file specification.

Note that the Parse service operates differently from other services with regard to the expanded string. With the Parse service, the expanded string contains all wildcard characters present in the file specification. VMS RMS does not generate the resultant string until the program invokes a related service, which uses the expanded string from the Parse service as input. When you use a search list, the expanded string contains the *first* location to be searched. VMS RMS stores internally the information that specifies the remaining search list equivalence strings. Note that the equivalence string from a $PARSE is not guaranteed to point to an actual file.

As different file locations are examined, VMS RMS updates the expanded string to reflect the current location, and the resultant string contains the actual file specification of the file.

With the Create, Display, Erase, Open, and Search services, defaults are applied to the expanded string to select the actual file used. The resultant string can be used by the program to indicate which file was located. When the file is located, the version number found (or created) is appended to the resultant file specification string (not the expanded file specification string). When a search list is used, the resultant string contains the file specification where the file was actually found.

The following sections describe the steps that VMS RMS uses to create a complete file specification.

6.2.1 Checking for Open-By-Name Block

If the open-by-name-block option is specified (FAB$V_NAM), VMS RMS examines the name block for a valid device identification (NAM$T_DVI field), directory identification (NAM$W_DID field), and file identification (NAM$W_FID field). If these fields are present, VMS RMS uses them to locate the file; all other components are ignored because they are not needed. If the open-by-name block succeeds, no expanded or resultant string is produced.

If these fields are not present in the name block or if an open-by-name block is not specified (for example, an Open service not preceded by a Parse service), VMS RMS performs the translation and application of defaults (see below). A file can also be created using the name block device and directory identification fields, but VMS RMS does not use the file identification.

If an open-by-name block is requested for remote DECnet file access between two VMS systems, VMS RMS does not check the device identification, directory identification or file identification to determine whether the requested open-by-name block operation can be performed. Instead, VMS RMS checks to see if a qualified resultant string is present. If a qualified resultant string is not present, VMS RMS translates logical names and applies defaults as if an open-by-name block operation was not requested (see Section 6.2.2).

6.2.2 File Specification Formats and Translating Logical Names

To form the file specification, VMS RMS examines and attempts to translate each program-supplied file specification, beginning with the primary file specification string indicated by the contents of the FAB$L_FNA and FAB$B_FNS fields.

A file specification may have one of three formats:

- The first file specification is in the following format:

  ```
  node::"foreign-filespec"
  node::"task-spec-string"
  ```

VMS RMS attempts to translate the node name to determine if a logical node name is present; only a logical or physical node name (including an access control string, if present) is allowed if the translation is successful. If a logical node name is found, the translation is repeated. When translation cannot be performed, the file specification is copied directly into the expanded string. The quoted string is not parsed except to determine if it refers to a file or a task on the remote system. For additional information about these formats, see the *VMS Networking Manual*.

- If the file specification contains only a name (without a terminating period or colon), VMS RMS attempts to translate it as a logical name. If the file name field is translated successfully, the entire translation operation restarts, using the equivalence string as input. If the file name field is not translated successfully, VMS RMS uses it as the file name component.
- If the file specification is not in either of the formats described previously, VMS RMS assumes it to be in the following file specification format:

  ```
  node::device:[root.][directory]filename.type;version
  ```

  Note that brackets do not imply optional file specification components. The only optional components are the node component and the root component.

  VMS RMS isolates the components, checks them for proper syntax, and copies them to the expanded string. If a node name is present, VMS RMS attempts to translate it as a logical node name as described previously. If a name in the device component is present and the node name is omitted, VMS RMS attempts to translate the device name as a logical name.

  After translating a logical name, VMS RMS determines whether the translation contains a duplicate component. If VMS RMS finds a duplicate component in the primary file specification translation, it signals an error. Conversely, if VMS RMS finds a duplicated component in the default string file specification translation or in the related string file specification translation, it ignores (discards) the duplicate component.

  If the node name is omitted and the device component does not translate successfully, VMS RMS treats the name in the device component as a device name.

  If the logical name translates successfully, VMS RMS performs one of the following actions:

  - Checks the equivalence string to determine whether it refers to a process-permanent file. If a process-permanent file is being referenced, VMS RMS copies the logical name to the expanded string and terminates processing the file specification (defaults are not needed). Process-permanent files are discussed in Section 6.4.
  - Checks the equivalence string to determine if the logical name is a concealed-device logical name. If the logical name is concealed, and if no concealed-device logical names have been encountered previously in the device file specification, the source string is used as the device name.
  - Restarts the translation operation using the equivalence string as input, if the equivalence string does not contain a process-permanent file and does not have the terminal attribute.

If a node name is present, VMS RMS passes the entire file specification (without the node name) to the remote node for interpretation, using the DECnet data access protocol (DAP) to communicate with the DECnet file access listener (FAL) at the remote node.

The logical name translation procedure reveals two conventions. First, if the file specification has been parsed previously by a VMS RMS file service, use the open-by-name-block option to save processing time. Second, a logical device name must be placed at the beginning of a file specification, unless it is preceded by a node name that indicates the node where the logical name should be translated.

6.2.3 Special Parsing Conventions

Additional parsing conventions for advanced file specifications include search lists, related file specifications, and the way VMS RMS handles directory specifications.

6.2.3.1 Parsing Conventions for a Search List

VMS RMS uses several conventions when processing a search list logical name.

- When VMS RMS encounters a search list, it searches internally for the file using search list file specifications previously specified. VMS RMS treats each file specification in the search list as a new file specification. That is, VMS RMS does not use components of one file specification element in the search list as the default for subsequent elements in the search list.

- When it uses search lists, VMS RMS ignores the following errors:

  Invalid device name (RMS$_DEV)
  Device not ready or not mounted (RMS$_DNR)
  Directory not found (RMS$_DNF)
  File not found (RMS$_FNF)
  Privilege violation (RMS$_PRV)

  All other errors terminate search list processing.

- When a search list is embedded (nested) in another search list, all file specifications of the nested search list are processed before the file specifications in the next-higher search list level. Therefore, VMS RMS permits iterative substitution in nested search lists as it does with other logical names. For example, consider the following search lists, X and Y:

  ```
  $ DEFINE X DISK1:[RED],DISK2:[WHITE]
  $ DEFINE Y X,DISK1:[BLUE]
  ```

  The following search order is derived from search list Y: is

  1. DISK1:[RED]
  2. DISK2:[WHITE]
  3. DISK1:[BLUE]

- When opening a file, VMS RMS tries all search list possibilities before it signals an error completion status. If VMS RMS cannot find the file, it displays, where applicable, the final search list file specification and the error message.

- When VMS RMS tries to locate a file using multiple search lists, it uses all combinations of the elements in the search lists. First it combines the first entry in the first list with the first entry in the second list. Then it combines the first entry in the first list with the second entry in the second list. After trying all combinations of the first entry in the first list with all entries in the second list, VMS RMS repeats the exercise using the entries in the second list with the second entry in the first list. This continues until VMS RMS locates the file or until it tries all combinations of all lists.

  For example, assume the program is looking for FILE.DAT, which may be in one of two directories, [BIG] or [BEST], on one of two disks, DISK1: or DISK2:. First, the program defines two search lists, a disk search list (PRIM) and a directory search list (DEF):

```
$ DEFINE PRIM DISK1,DISK2
$ DEFINE DEF [BIG],[BEST]
```

  Next, the program provides VMS RMS with a primary file specification that includes the search list (PRIM) for the disk together with the file name component:

```
PRIM:FILE
```

  Finally, the program must give VMS RMS the default specification that includes the search list (DEF) for the directory together with the file type component:

```
DEF:.DAT
```

  Given this information, VMS RMS looks for FILE.DAT using the file specification data in the following order:

Primary File Specification	Default File Specification	Expanded String
DISK1	[BIG]	DISK1:[BIG]TEST.DAT;
DISK2	[BIG]	DISK2:[BIG]TEST.DAT;
DISK1	[BEST]	DISK1:[BEST]TEST.DAT;
DISK2	[BEST]	DISK2:[BEST]TEST.DAT;

  Now, assume the program provides a related file specification with a search list for FILE.DAT.

  1 VMS RMS uses all combinations of the search list elements in the primary and default file specifications with the *first* component (device) of the related file specification.

  2 VMS RMS uses all combinations of the search list elements in the primary and default file specifications with the *second* component (directory) of the related file specification.

  3 VMS RMS repeats Steps 1 and 2 with each search list element in the related file specification.

6.2.3.2 Special Processing for a Related File Specification

This section describes the special processing needed to implement sticky defaults when a search list is used in a related file specification for an input file parse. The term *sticky default* means that file specification components from the first file specification are applied as defaults to the next file specification component, eliminating the need, for instance, to specify the device specification for each file specification when all the files are located on the same device.

The related file specification provides defaults when a related file name block is present. To use the related file specification, the file access block must specify the address of the primary file's name block (in the FAB$L_NAM field), and that name block must specify the address of the related file's name block (in the NAM$L_RLF field). The related file's name block must specify the address of a valid file specification in the resultant string (NAM$L_RSA and NAM$B_RSS) fields. Typically, a VMS RMS file service (other than Parse) places the file specification in the resultant string.

You can specify whether the related file is used as an input file specification or an output file specification by setting (output file specification parsing) or resetting (input file specification parsing) the output-file parse (FAB$V_OFP) bit in the file-processing options (FAB$L_FOP) field .

When an input file specification is being parsed, you can have multiple related name blocks by specifying the second related file's name block address in the NAM$L_RLF field of the first related name block, the address of the third related name block in the NAM$L_RLF field of the second name block, and so forth. The use of multiple related name blocks is especially useful for search lists; one related name block might contain a file type that can be used by any file specification in a search list, another might contain the full file specification that was produced by the first search list file specification, and another might contain the full file specification produced by the second search list file specification. This method allows the file specifications in a search list to provide sticky defaults, a characteristic associated with DCL command lines that contain multiple file specifications.

For a search list to be applied as a related file specification, the related file specification must not be a resultant string but must include the search list logical name. The related file specification in this case must describe the original primary file specification. For example, consider the following search list definition:

```
$ DEFINE WORK DISK1:[MINE],DISK2:[GROUP]
```

To process lists of input files—such as WORK:A,B,C,—your program must supply the string WORK:A as the related file specification, not DISK2:[GROUP]A.DAT. The routines LIB$FIND_FILE and LIB$FILE_SCAN are provided to perform this special processing; consult the *VMS Run-Time Library Routines Volume* for additional information; also refer to Example 5–2, which shows how to call the LIB$FIND_FILE routine.

6.2.3.3 Input File Specification Parsing

When the output-file parsing bit (FAB$V_OFP) is reset and the node name is omitted, VMS RMS processes the related file specification as an input file specification. This is shown in the following table. Note that the only wildcard character allowed is a single asterisk.

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Node	Use related file specification	Illegal
Device	Use related file specification	Illegal
Directory	Use related file specification	Remains wild
Filename	Use related file specification	Remains wild
Type	Use related file specification	Remains wild
Version	Remains null	Remains wild

When the FAB$V_OFP bit is reset and a node name is present, VMS RMS processes the related file specification as an input file specification as shown in the following table:

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Device	Remains null	Illegal
Directory	Remains null	Remains wild
Filename	Use related file specification	Remains wild
Type	Use related file specification	Remains wild
Version	Remains null	Remains wild

6.2.3.4 Output File Specification Parsing

When the FAB$V_OFP bit is set and a node name is not present, VMS RMS processes the related file specification as an output file specification as shown in the following table:

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Node	Remains null	Illegal
Device	Remains null	Illegal
Directory	Remains null	Substitute from related file specification with restrictions
Filename	Use related file specification	Substitute from related file specification
Type	Use related file specification	Substitute from related file specification
Version	Remains null	Substitute from related file specification

When the FAB$V_OFP bit is set and a node name is present, VMS RMS processes the related file specification as an output file specification, as shown in the following table:

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Device	Remains null	Illegal
Directory	Remains null	Substitute from related file specification with restrictions
Filename	Use related file specification	Substitute from related file specification
Type	Use related file specification	Substitute from related file specification
Version	Remains null	Substitute from related file specification

As shown in the previous table, a wildcard character in an output directory specification is subject to the following syntax restrictions:

- Only the asterisk and the ellipsis are permitted in the output directory specification. In the case of a related file specification, you may choose either the asterisk or the ellipsis (but not both) in the output directory specification unless you use the following form:

  [*...]

- A subdirectory specification that contains wildcard characters cannot be followed by a subdirectory specification that does *not* contain wildcard characters.
- Specifications in the UIC directory format may receive defaults only from directories in the UIC directory format.
- Specifications in the non-UIC directory format may receive defaults only from directories in the non-UIC directory format.

- Specifications in the non-UIC directory format that consist entirely of wildcard characters may receive related file specification defaults from directories in UIC or non-UIC format.

VMS RMS processes wildcard characters in an output directory specification as follows:

- If you specify an output directory using a specification that consists entirely of wildcard characters ([*] and [*...] only are allowed), VMS RMS accepts the complete directory component from the related file specification. This permits you to duplicate an entire directory specification.
- If you specify an output directory with a trailing asterisk (for example, [A.B.*]), VMS RMS substitutes the first "wild" subdirectory from the related file specification. This substitution permits you to move files from one directory tree to another directory tree that is not as deep as the first one.
- If you specify an output directory with a trailing ellipsis (for example, [A.B...]), VMS RMS substitutes the entire "wild" subdirectory from the related file specification. This substitution permits you to move entire subdirectory trees.
- The related name block must have the appropriate file name status bits set in the NAM$L_FNB field set according to the resultant string to allow VMS RMS to identify the "wild" portion of the resultant string.

6.3 Directory Syntax Conventions and Directory Concatenation

One of the components of a file specification is the directory specification. VMS RMS supports two conventions or types of directory specifications, one of which is used more often than the other.

When VMS RMS applies defaults to a directory specification in a file specification, the rules differ depending on what type of a directory specification is present. Two directory syntax conventions are available to access directories: normal and rooted. The default directory access is normal syntax. That is, you can specify the directory desired using the directory syntax described in the *VMS DCL Dictionary*.

6.3.1 Using Normal Directory Syntax

There is a master file directory (MFD) on each disk volume. Within each MFD, top-level directories are cataloged using the DCL command CREATE/DIRECTORY (or equivalent VMS RMS services). Beneath each top-level directory, you can create subdirectories referenced from the top-level directory.

Once the subdirectories are created, you can create subdirectories referenced from each subdirectory. You can create a maximum of seven levels of subdirectories beneath a top-level directory. The subdirectories created beneath a directory and the subdirectories within the subdirectories (and so forth) are called collectively a *directory tree*.

The base point for normal directory syntax access can be relative to the current position in the directory tree or an absolute reference that explicitly or by default states any higher-level directories or subdirectories needed to identify the appropriate directory or subdirectory. An absolute directory reference begins with a directory name; a relative directory reference begins with a hyphen (-) or a period (.). An absolute reference might include the name of the top-level directory and one or more subdirectories. A relative directory reference relies on the use of the process-default directory and device, which are set using the DCL command SET DEFAULT. Refer to the *VMS DCL Dictionary* for additional information and examples.

A relative directory reference can be in one of several forms. Assume the current directory position (process-default directory) is [SMITH.JONES].

- You can specify a lower level in the directory tree with a period (.) to indicate that the current directory position ([SMITH.JONES]) is prefixed to the specified directory as shown in the following command:

  ```
  $ SET DEFAULT [.DATA]
  ```

  This directory specification is combined with the current directory position to form [SMITH.JONES.DATA].

- You can specify higher levels in the directory tree by beginning the directory specification with a hyphen (-) to indicate that the directory specification is the next level up from the current directory level. If you are at currently at directory level [SMITH.JONES], the following command directs VMS RMS to use the directory SMITH:

  ```
  $ SET DEFAULT [-]
  ```

  If you include more than one hyphen, VMS RMS ascends the directory tree by a corresponding number of levels. For example, if you use the following command from directory level [RED.WHITE.BLUE], VMS RMS moves up the tree to level [RED]:

  ```
  $ SET DEFAULT [--]
  ```

- You can use combinations of hyphens and periods to traverse a directory tree. For example, assume the following directory tree structure:

  ```
          ONE
         /   \
        TWO  THREE
       /          \
      FOUR         FIVE
     /    \
  SIX    SEVEN
  ```

  Assume that your process is in directory [ONE.TWO.FOUR.SIX] and you want to access a file in [ONE.THREE.FIVE]. You can do this with the following DCL command:

  ```
  $ SET DEFAULT [---.THREE.FIVE]
  ```

- You can refer to the default directory explicitly by specifying an empty directory specification at the DCL prompt. This feature is useful when you to use a single DCL command to perform directory operations in your default directory and one other directory.

For example, assume you have a directory on device USERDISK named [CUSTOMERS.LOCAL] that contains three files: ABERCROMBIE, FITCH, and GOULD. Another directory named [CUSTOMERS.INTERNATIONAL] also contains three files: MERRILL, LYNCH, and PIERCE. Assume that your default directory is [CUSTOMERS.LOCAL] but you need a directory listing that contains the sizes of all customer files. You can list both directories using the following command line:

```
$ DIRECTORY/SIZE [CUSTOMERS.INTERNATIONAL],[]
```

DCL responds to this command with the following display:

```
Directory USERDISK:[CUSTOMERS.INTERNATIONAL]

MERRILL          1100
LYNCH             155
PIERCE            645

Directory USERDISK:[CUSTOMERS.LOCAL]

ABERCROMBIE       230
FITCH             100
GOULD             355

Total of 6 files, 2585 blocks
```

A directory name at the leftmost end of a directory specification is interpreted as a top-level directory, or an absolute directory reference. The syntax shown for the following specification refers to a top-level directory named GREEN, regardless of the current default directory:

[GREEN]

Conversely, a period or a hyphen before a directory name is always associated with a relative directory reference.

Note that because only one directory can be directly above any other directory, you can use a hyphen without explicitly naming the next higher directory. But, because many directories can be directly beneath the current directory, you must explicitly specify the directory at the next lower level by following the period with the name of the selected directory.

If the program omits either the device or the directory component in a file specification, VMS RMS accepts the value of the current device and directory from the logical translation of SYS$DISK. Therefore, any directory fields yielded by translation of SYS$DISK override the process default directory. If translation of SYS$DISK does not yield the directory element, VMS RMS uses the process default directory. Note that you can change the process default directory with the SET DEFAULT command or by invoking the SYS$SETDDIR system service.

6.3.2 Rooted-Directory Syntax Applications

Rooted-directory syntax allows you to refer to directory trees as logical devices and top-level directories. A reference to a top-level directory actually accesses existing subdirectories without program modification; thus, rooted-directory syntax extends the flexibility associated with logical names. Similarly, rooted-directory syntax can reduce the number of top-level directories needed for a disk volume. Rooted-directory syntax allows multiple VMS system directory trees to exist on a single system volume.

You specify rooted-directory syntax using a logical name in a program-specified file specification or in the device and directory for a SET DEFAULT command. If a program specifies a logical device name in the file specification, the logical device name can be redefined to specify a rooted-directory logical name. Redefining the logical device name to specify a rooted directory changes the directory (and the file or files) accessed by the program without modifying the program.

If a program does not specify a logical device name in the file specification, the user (or a command procedure) can issue DEFINE commands and the SET DEFAULT command before running the program to indicate the use of rooted-directory syntax and to specify the process-default device/directory. Using the SET DEFAULT command changes the directory accessed by the program without requiring that you modify the program. When the program finishes, use the SET DEFAULT command again to specify the new process-default device/directory and to resume the use of normal directory syntax (if desired).

Using rooted-directory syntax as illustrated in the preceding text provides several advantages. Because you did not modify the program, you avoided having to recompile (or reassemble), relink, or retest it. Another advantage is that because you were able to run the program from its resident directory, you eliminated the overhead of having to move the file several times.

6.3.3 Using Rooted-Directory Syntax

Rooted-directory syntax provides a way of making a directory or subdirectory appear to the user as the master file directory (MFD) for the logical disk volume. Subdirectories of the rooted directory appear as top-level directories (user file directories) for the logical disk volume.

The directory specified during logical name definition serves as a base from which directories beneath it can be accessed and is called the *root directory*. Root directories must be specified using alphanumeric UICs; octal numbering for group and member designations is not allowed.

A concealed-device logical name that defines a hidden root directory is called a *rooted-device logical name.*

When you define the rooted-device logical name for use in a program or in a SET DEFAULT command, you specify that the logical name is a concealed-device logical name by using the /TRANSLATION_ATTRIBUTES=CONCEALED qualifier with the DCL command DEFINE or ASSIGN. To define the concealed-device logical name as a rooted-device logical name, the root directory must contain a trailing period (.), such as DUA22:[ROOT.]. When specifying a directory, you can only use a trailing period for the root directory.

When you define a root directory, all directory references refer to the specified root directory or directories beneath it in the directory tree. A reference to a top-level directory refers to a subdirectory of the specified root directory when using rooted-directory syntax. This is consistent with the fact that the root directory appears as the MFD because a reference to directory [000000] refers to the root directory. When you execute the SHOW DEFAULT and other direct commands, the system displays the root directory as [000000]. Note that the directory specification form [0,0] for the MFD is not valid when using rooted-directory syntax.

For example, assume the top-level directory [ROOT1] on disk DUA7 contains a subdirectory [ROOT1.SUB]. The directory [ROOT1] is defined as the root directory associated with the logical name BASE as follows:

```
$ DEFINE BASE DUA7:[ROOT1.]
```

When you associate the root directory with the logical name base, you can refer to the logical top level directory [ROOT1.SUB] using the syntax BASE:[SUB]. The following chart provides a summary of the actual directory accessed and the equivalent rooted-directory syntax that applies to this example:

Actual Directory	Rooted Syntax	Comments
DUA7:[ROOT1]	BASE:[000000]	[ROOT1] appears as the MFD
DUA7:[ROOT1.SUB]	BASE:[SUB]	[ROOT1 SUB] appears as a top-level directory

The next example shows how to define the root logical name described in the previous chart and how to access a subdirectory of the specified root directory. Note that the trailing period [ROOT1.] indicates that a root directory is present.

```
$ DEFINE/TRANSLATION_ATTR=CONCEALED BASE DUA7:[ROOT1.]
$ SET DEFAULT BASE:[SUB]
$ DIRECTORY *.DIR,[-]*.DIR
```

The system responds with the following display:

```
BASE:[SUB]

SUBSUB.DIR

BASE:[000000]

SUB.DIR
```

In the preceding example, the SET DEFAULT command defines the process-default directory as [ROOT1.SUB] using the rooted-device logical name BASE. The DIRECTORY command looks for directory files in the current directory ([ROOT1.SUB]) and then in the directory directly above it ([ROOT1]). The directory [ROOT1.SUB] is listed (by the DIRECTORY command) as a top-level directory (BASE:[SUB]) and the root directory is listed using the syntax of the MFD, BASE:[000000].

6.3.4 Concatenating Rooted-Directory Specifications

When it concatenates specifications for rooted directories, VMS RMS uses different syntax rules than it uses when it concatenates directory specifications for normal directory syntax.

One difference between the two conventions is associated with the trailing period in the root directory definition. For example, consider how a top-level directory reference is handled. With rooted-directory syntax, the root directory's trailing period is implied as a leading period in subsequent rooted-directory references.

Directory concatenation *within the same file specification* occurs only with a rooted-device logical name. Normal directory concatenation occurs only through the application of defaults. Rooted-directory concatenation can occur within the same file specification and through the application of defaults. Rooted-device logical names specify a device name and a root directory. VMS RMS translates a rooted-device logical name into the device and root directory before it merges any other parts of a file specification (if present) into the equivalence file specification.

When you use a rooted-device logical name together with a directory specification, the following rules apply:

- You can refer to the root directory itself. The syntax of [000000] and relative directory references refer to the root directory.

  You can never refer to a directory *above* the specified root directory because the root directory is the logical MFD whenever a directory specification is used. When the process-default directory is the root directory, a reference to [-] results in an error, as shown in the following example:

  ```
  $ DEFINE/TRANSLATION_ATTR=CONCEALED BASE DUA7:[ROOT1.]
  $ SET DEFAULT BASE:[000000]
  $ DIRECTORY *.DIR
  ```

  The system responds to this command sequence with the following display:

  ```
  BASE:[000000]

  No files found
  ```

  The user then tries to check the contents of the next higher directory with the following command:

  ```
  $ DIRECTORY [-]*.DIR
  ```

  The system responds with the following messages:

  ```
  %DIRECT-E-OPENING, error opening [-]*.DIR as input
  -RMS-E-DIR, error in directory name
  ```

- You can refer to a specific subdirectory of the root directory in the same way that you refer to a top-level directory using normal directory syntax, as shown in the following example:

  ```
  $ DEFINE BASE DUA7:[ROOT1.]
  $ SET DEFAULT BASE:[SUBDIR]
  ```

- You can refer to any subdirectory beneath the root directory using wildcard characters to vertically traverse the directory tree. You can refer to all directories below the root directory [*...], all directories one level below the root directory [*], all directories two levels below the root directory [*.*], and other reference combinations, as shown in the following example:

  ```
  $ DEFINE/TRANSLATION_ATTR=CONCEALED BASE DUA7:[ROOT1.]
  $ DIR BASE:[*...]*.DIR
  ```

  The system responds with the following display:

  ```
  BASE:[SUBDIR]

      SUBSUBDIR.DIR

  BASE:[SUBDIR.SUBSUBDIR]

      SUBSUBSUBDIR.DIR

  BASE:[OTHERSUB]

      OTHERSUBSUB.DIR
  ```

Another difference between the conventions VMS RMS uses for rooted-directory syntax and standard directory syntax is the number of permissible nested directory levels. With rooted-directory logical names you can "hide" eight additional levels in the rooted-directory logical name and effectively nest 16 levels of directories.

Note that you must access these files using the rooted-directory logical name and that the system rejects any attempt to access these files using normal directory syntax. For example, you can legally define the rooted-directory logical name MYROOT to be DUA0:[D1.D2.D3.D4.D5.D6 .] and nest six subdirectories beneath it using the the following directory syntax:

MYROOT:[D7.D8.D9.D10.D11.D12]name.type

But if you try to access this file using the following directory syntax, VMS RMS returns a status code indicating the file specification is illegal:

DUA0:[D1.D2.D3.D4.D5.D6.D7.D8.D9.D10.D11.D12]name.type

Another problem occurs when you try to back up the directory tree using conventional directory syntax, because the Backup Utility only backs up the first eight levels of the directory tree.

Note: **This problem does not occur when you back up an entire disk at one time, that is, if you are using the BACKUP/IMAGE command or the BACKUP /PHYSICAL command.**

With rooted-directory snytax, VMS RMS uses the process-default device and directory *indirectly* as defaults. This occurs because VMS RMS uses the expanded rooted-device logical name device and root directory before applying the process-default device and directory.

With rooted-directory snytax, you can use relative directory syntax, such as the hyphen (-) and leading period (.name). When a directory component is missing, VMS RMS attempts to obtain the missing components from the process-default directory.

Consider the rooted-device logical name X defined as shown in the following DCL command:

```
$ DEFINE X DJB3:[SMITH.]
```

Now assume you set the default directory to JONES:

```
$ SET DEFAULT [JONES]
```

When the rooted-device logical name X is used with a directory specification, all directory references are relative to the root directory [SMITH.]. Most wildcard characters that apply to normal directory syntax also apply to rooted-directory syntax.

The following table lists the file specifications involving the rooted-device logical name X and the directory that is accessed with each specification:

File Specification	Directories Accessed
X:	[SMITH.JONES]
X:[000000]	Root directory, [SMITH.]
X:[ ]	[SMITH.JONES]
X:[-]	Root directory [SMITH.], listed as X:[000000]
X:[- -]	Invalid (error)
X:[name]	[SMITH.name]
X:[.name]	[SMITH.JONES.name]
X:[name.*...]	All directories in all directory trees below [SMITH.name]
X:[*]	All directories one level below [SMITH.]
X:[*...]	All directories in all directory trees below [SMITH.]
X:[...]	All directories in all directory trees below [SMITH.JONES]

Note that VMS RMS uses the default directory with relative directory references when the specified directory name contains a leading period or a hyphen, or if no directory name is specified.

6.3.5 An Example of Using a Rooted Directory

Consider an application made up of several programs that refer to the same file using a file specification IN:[INVENTORY]FILE.DAT. Assume that all of the programs invoke the following command procedure to define the logical name IN as device DUA29:

```
$    ON CONTROL_Y THEN GOTO ENDIT
$    DEFINE IN DUA29:
$    RUN XYZPROG
$ ENDIT:
$    EXIT
```

The programs show the current inventory level and the stockroom used for a particular item and are dispersed among many users in the company. As the business grows, the number of items in the inventory grows and the number of inventory records makes the file extremely large and difficult to access. Because the items can be classified as belonging to one of four groups, the data management department decides to split the file into four parts. A special-purpose program reads each record in the master file, determines the record type, and routes the record to the appropriate file group, each on a separate device. All output files are named FILE.DAT, but each is distinguished by putting it in a top-level directory associated with the appropriate group category. For example, computer supplies files are cataloged in the directory [COMPUTER.INVENTORY].

This is done by modifying the command procedure to conditionally define the value of IN to be a rooted-device logical name with four subdirectories. The modified command procedure is shown in Example 6–1.

Example 6–1 Example of Rooted-Directory Syntax

```
$ ON CONTROL_Y THEN GOTO END
$    GOTO ASK
$ RETRY:
$    WRITE SYS$OUTPUT "Enter a number from 1 to 4 for the type of part"
$ ASK:
$    WRITE SYS$OUTPUT -
       "Select Parts Group: 1-COMPUTER 2-TYPEWRITER 3-DESK 4-OTHER 5-END"
$    INQUIRE ANS
$    IF ANS .GT. 5 .OR. ANS .LT. 1 THEN GOTO RETRY
$    IF ANS .EQ. 5 THEN EXIT
$    IF ANS .EQ. 1 THEN DEFINE/TRANS=CONCEAL IN DUA29:[COMPUTER.]
$    IF ANS .EQ. 2 THEN DEFINE/TRANS=CONCEAL IN DUA29:[TYPEWRITER.]
$    IF ANS .EQ. 3 THEN DEFINE/TRANS=CONCEAL IN DUA29:[DESK.]
$    IF ANS .EQ. 4 THEN DEFINE/TRANS=CONCEAL IN DUA29:[OTHER.]
$    RUN XYZPROG
$ END:
$    EXIT
```

With the enhanced command procedure, none of the programs has to be modified, recompiled (or reassembled), relinked, or copied to a different directory.

6.4 Using Process-Permanent Files

A *process-permanent* file is a file that VMS RMS opens or creates in supervisor or executive mode code. This is initiated by setting the process-permanent file bit (FAB$V_PPF) in the file-processing options field (FAB$L_FOP). When the bit is set, VMS RMS-maintained internal data structures are allocated in the process control region of memory for the life of the process. Thus, process-permanent files can remain open across image activations. SYS$COMMAND, SYS$INPUT, SYS$OUTPUT, and SYS$ERROR are all opened in this manner by the LOGINOUT command image.

The DCL command OPEN also opens files in this manner. With user mode code, you can only access process-permanent files indirectly. VMS RMS provides a subset of the total available operations to the indirect accessor.

Indirect accessors gain access to process-permanent files through the logical name mechanism, as follows:

1 The LOGINOUT command image, or at a later point the command interpreter, opens or creates a file corresponding to the process's command, input, output, and error message streams. Logical names are created in the process logical name table for SYS$COMMAND, SYS$INPUT, SYS$OUTPUT, and SYS$ERROR, respectively. The equivalence string for the logical name has a special format that indicates

the correspondence between the logical name and the related process-permanent file. For more detail concerning the equivalence-string format for logical names, see the discussion of logical name services in the *VMS System Services Reference Manual*. For example, for an interactive user, these single process-permanent files are opened for the terminal.

2 When an indirect accessor opens or creates a file specifying a logical name that has one of these special equivalence strings, VMS RMS recognizes this and therefore does not open or create a new file. Instead, the returned value for the internal file identifier (and later the value for the internal stream identifier from a Connect service) is set to indicate that access to the associated process-permanent file is with the indirect subset of allowable functions.

The implications for the indirect accessor are described in the following list:

- A Create service for a process-permanent file becomes an Open service; the fields of the FAB are output according to the description of the Open service, not the Create service.
- The Open and Create services require no I/O operations.
- Any number of indirect Open and Create operations are allowed.
- There is only one position context for the file; each sequence of the Open or Create service accesses the same record stream, not an independent stream.
- If the process-permanent file was initially opened with the sequential-processing-only (FAB$V_SQO) bit set in the FAB$L_FOP field, neither random access nor the Rewind service is permitted. This is the case for SYS$COMMAND, SYS$INPUT, SYS$OUTPUT, and SYS$ERROR.
- Certain options to various services produce errors. For example, you cannot set the non-file-structured (FAB$V_NFS), process-permanent file (FAB$V_PPF), and user-file-open (FAB$V_UFO) bits of the FAB$L_FOP field for the Open and Create services. Other options are ignored, such as the spool (FAB$V_SPL), submit-command-file (FAB$V_SCF), and delete (FAB$V_DLT) bits of the FAB$L_FOP field for the Close service, the asynchronous (RAB$V_ASY) bit of the RAB$L_ROP field, and both the multiblock count and multibuffer count fields (RAB$B_MBC and RAB$B_MBF).
- If a name block is used and either an expanded or resultant file specification string is returned, the string consists solely of the process logical name followed by a colon (such as SYS$INPUT:).
- The file access (FAB$B_FAC) field is ignored by the Open service; instead, operations are checked against the FAB$B_FAC field specified for the original Open or Create service.
- Information from the record attributes field is saved on each Open service and subsequent Connect service in the values returned in the internal file identifier and internal stream identifier fields. If the output file is a print file (VFC record format and the FAB$V_PRN bit set in the record attributes field), mapping is performed for each Put service from the user-specified carriage control to the print file carriage control format. Thus, different carriage control types from different indirect Open services all work correctly.
- You cannot use the Erase service.

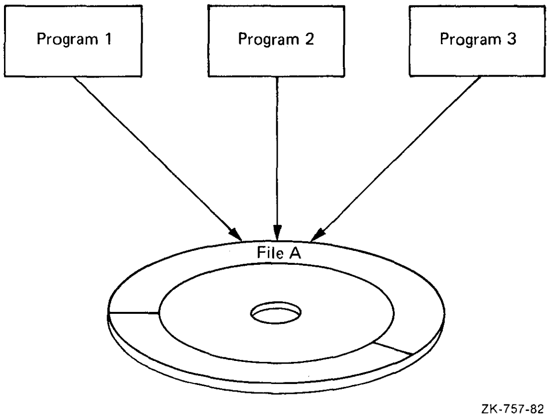
- Checking is performed for $DECK, $EOD, and other dollar-sign ($) records on the SYS$INPUT stream if the SYS$INPUT stream is from a file. Checking is not done if SYS$INPUT comes from a record-oriented device, such as a terminal or a mailbox. (See the *VMS DCL Dictionary.*)
- At image exit time, the VMS RMS run-down control routine ensures that the indirect I/O on process-permanent files terminates; however, these files are not closed.
- All file organizations may be opened directly as process-permanent files (for example, through the DCL command OPEN), but only those files with a sequential organization may be indirectly accessed.

7 File Sharing and Buffering

This chapter discusses the run-time options that are available when opening, connecting, and closing a shared file. These options are also implicit in creating a shared file because the Create service includes an implied file open, which uses the run-time options either explicitly or by default.

File sharing includes file accessing, record locking, and local and shared buffering. Figure 7–1 shows a typical shared file situation.

Figure 7–1 Shared File Access



See the *VMS Record Management Services Manual* for more information about accessing and sharing files.

7.1 File Accessing

VMS RMS file sharing allows multiple users to access a single file. Timely access to files sometimes requires that more than one active program be allowed to read, write, and modify records within the same file simultaneously.

Whether or not a file can be shared depends on the type of device it resides on and the explicit file-sharing information specified by the processes that access the file. Magnetic tape files cannot be shared because magnetic tape drives are sequentially-operated devices. However, disk files can be shared by any combination of readers and writers without restriction. Your program provides the information that enables file sharing. You control the degree of sharing by providing VMS RMS with an explicit file-sharing specification when your program opens or creates a file. This specification indicates the types of file operations that are permitted for application programs that share the file.

When a program creates or opens a disk file, it gives VMS RMS two pieces of information needed to determine if and how the file may be shared. First, it states the types of operations it intends to perform on the file, such as read, write, or update. VMS RMS later checks this information to protect against unauthorized file access.

Second, the program specifies the types of operations other concurrently active programs can perform on the file. When the sharing specification of one program is compatible with the sharing specification of another, both programs can gain access to the file simultaneously. To ensure that multiple programs can access the file simultaneously, you may have to do some schedule planning.

7.1.1 Types of File Sharing and Record Streams

A single process can access the same file using multiple *record streams.* A record stream is the access environment in which file records may be read, written, deleted, or updated. Important elements of the access environment are the current record position (if any), the access mode established for the current record, the sequential next record position, and the state of locks on other records in the file.

The Connect service creates a record stream and associates it with a file previously opened or created by the appropriate VMS RMS service. The connection between a record stream and a file is explicitly terminated by the Disconnect service or is implicitly terminated by closing the file. Record streams are connected to a file in one of three ways:

- Within one process or across several processes, multiple FABs can be connected to a shared file. One or more record streams may then be connected to each FAB. This form of sharing is known as *interlocked interprocess file sharing* and is associated with reading or writing records, not blocks.
- Within one process, multiple record streams can be associated with one FAB to read and write records, not blocks. This form of sharing is known as *multistreaming.*
- Within one process or across several processes, multiple FABs can be connected to a file. One record stream (RAB) is connected to each FAB, and users provide their own synchronization outside of VMS RMS. This form of file sharing is known as *user-interlocked interprocess file sharing.* (User-interlocked interprocess file sharing usually applies only to block I/O processing and to record processing for nonshared sequential files residing on disk devices.)

Two options that are especially important for shared files are the file-access and file-sharing options. These options specify the type of record access that the sharing processes can perform. They are specified by the FDL attributes ACCESS and SHARING and the VMS RMS FAB fields identified by the symbolic offsets FAB$B_FAC and FAB$B_SHR. When creating or opening a file, VMS RMS compares the values of these fields to determine whether or not the requesting process may have access to the file.

In this manual, the term *accessor* refers either to a process that accesses a file or a record stream that a accesses a record. The first process to access a file determines which operations other accessors can perform on the file, which in practice determines whether or not subsequent users are allowed to access the file. For example, if your process requests a certain type of access that the first accessor (since the file was last closed) has disallowed, your process cannot access the file.

When choosing the access other processes may have to the file, you can specify the type of file sharing to be used and indicate whether or not other processors can access the file simultaneously. In a VAXcluster environment, processes can access shared files on the same or different nodes of the same VAXcluster (see Section 3.6).

A single file can be accessed by both interlocked interprocess file sharing and multistreaming. DIGITAL does not recommend the simultaneous use of interlocked interprocess file sharing and user-interlocked interprocess file sharing on the same file if the process that requests user-interlocked interprocess file sharing intends to modify the file. The reason is that record locking is not done or checked for the processes using user-interlocked interprocess file sharing.

You must define your process access based on planned record operations. The record operations with associated FDL and VMS RMS options are summarized in Table 7–1.

Table 7–1 File Access Record Operations

Function (VMS RMS service)	FDL and VMS RMS Options
Read records (Get)	ACCESS GET specified or FAB$B_FAC field FAB$V_GET set
Locate records (Find)	ACCESS GET specified or FAB$B_FAC field FAB$V_GET set
Delete records (Delete)	ACCESS DELETE specified or FAB$B_FAC field FAB$V_DEL set
Add new records (Put)	ACCESS PUT specified or FAB$B_FAC field FAB$V_PUT set
Truncate file (Truncate)	ACCESS TRUNCATE specified or FAB$B_FAC field FAB$V_TRN set
Modify records (Update)	ACCESS UPDATE specified or FAB$B_FAC field FAB$V_UPD set
Access blocks (see text)	ACCESS BLOCK_IO specified or FAB$B_FAC field FAB$V_BIO set; under certain conditions, ACCESS RECORD_IO or FAB$B_FAC FAB$V_BRO

The record-access functions you request are compared with the protection on the specified file. If your process is limited to reading and locating records, it should have read access to the file. If your process is deleting, adding, truncating, or updating records, it must have write access to the file. VMS RMS permits any process that may delete, add, truncate, or modify records to also locate and read records because write access to a file also implies read access.

With VMS RMS, you can perform block I/O operations using the Read, Space, and Write services. Block I/O is usually only used by applications written in VAX MACRO or other low-level languages. Note that when ACCESS BLOCK_IO is specified, the application program must also specify either SHARING USER_INTERLOCK or SHARING PROHIBIT.

Different types of record operations can be specified to define the type of access to be allowed for other processes, as shown in Table 7–2.

Table 7–2 File Sharing Record Operations

Function (VMS RMS Service)	FDL and VMS RMS Options
Read records (Get)	SHARING GET specified or FAB$B_SHR field FAB$V_SHRGET set
Locate records (Find)	SHARING GET specified or FAB$B_SHR field FAB$V_SHRGET set
Delete records (Delete)	SHARING DELETE specified or FAB$B_SHR field FAB$V_SHRDEL set
Add new records (Put)	SHARING PUT specified or FAB$B_SHR field FAB$V_SHRPUT set
Modify records (Update)	SHARING UPDATE specified or FAB$B_FAC field FAB$V_SHRUPD set
No access	SHARING PROHIBIT or FAB$B_SHR field FAB$V_NIL set
User interlocking	SHARING USER_INTERLOCK or FAB$B_SHR field FAB$V_UPI set
Multistreaming	SHARING MULTISTREAM or FAB$B_SHR field FAB$V_MSE set

If other processes are limited to reading and locating records, they are unable to modify or add records, and record-lock checking is not performed. If other processes are allowed to delete, add, or modify records, they are also able to read records; however, record-lock checking occurs. All record-access functions use interlocked interprocess file sharing.

No access denies access to all accessors except the accessor who specifies this option. The no-access option might be used when a file is shared infrequently or to perform a major file update. When using this option, close the file promptly if other users may need to access the file. Choose this option or the user-interlocking option when using block access; to use the Queue I/O Request system service, specify the FILE USER_FILE_OPEN attribute (FAB$L_FOP field FAB$V_UFO set). The no-access option does not allow file sharing and requires that your process have write file protection access.

User interlocking permits the user, not VMS RMS, to maintain interlocking protection (including maintaining the end-of-file mark). For any other form of file sharing, VMS RMS controls the reading and writing of I/O buffers to ensure the integrity of file and record structures. This option is useful for nonshared sequential files and for block I/O access using VMS RMS or the Queue I/O Request system service.

Multistreaming allows your process to access the same file using more than one record stream and allows other users to access the file using interlocked interprocess file sharing (unless SHARING PROHIBIT is also specified). When you select this option, select the appropriate SHARING record operations, such as SHARING GET. When multiple streams are connected, the buffers allocated for each stream become part of a buffer cache for the entire process. (A buffer cache is a common shared buffer pool intended to minimize I/O.) A record operation on one stream may use cached buffers from a previous record operation on a different stream that referenced the same buckets.

When you open or create a file, you must specify the file access and file sharing you want for it. When using FDL or VMS RMS, the default is to read records from the file (ACCESS GET) and to allow others accessors to read records from the file (SHARING GET). Typically, an application program may want to read records (ACCESS GET) while allowing other accessors to add records (SHARING PUT). You might want to modify records (ACCESS UPDATE) while allowing other accessors to add new records to the file (SHARING PUT).

When you create a file, the default for FDL and VMS RMS to add records to the file (ACCESS PUT) and to not allow others to access the file (SHARING NONE). When you create a file with the create-if option, it is especially important to specify the access and sharing values. In this instance, you have denied yourself access if the file already exists because you have specified SHARING NONE and you are not the initial accessor. One way to avoid this when you create a file is to allow most operations for other users (such as SHARING GET, SHARING PUT, SHARING UPDATE, and SHARING DELETE).

Combinations of file access and file sharing that specify a mixture of interlocked interprocess file access and user-interlocked interprocess file sharing allow the application program to access the file without record locking protection. Such combinations are not recommended for general use; they should be used only for application programs that require read-only access to a file. Other combinations may cause an error, such as requesting ACCESS BLOCK_IO without specifying SHARING NONE or SHARING USER_INTERLOCK.

7.1.2 Interlocked Interprocess File Sharing

Interlocked interprocess is the most common form of file sharing. This method allows the connection of one or more record streams (RABs) to one or more processes (FABs), either within a single process or across several processes. When using this form of file sharing, the values specified for file sharing and file access by the initial accessor determine the type of access permitted for subsequent processes.

The initial accessor must consider the restrictions that result from the values specified for file sharing and file access. Typically, the initial accessor denies all write access to subsequent processes. Such a restriction occurs when the initial accessor specifies some type of write access for file access without specifying write access for file sharing.

If the initial accessor specifies read-only file access and file sharing, subsequent accessors can only read the file. If the appropriate type of write access is not specified, then subsequent accessors cannot perform the corresponding write operations to the file.

If the initial accessor specifies one or more values for file sharing, subsequent processes can access the file only if they specify compatible file access values. For example, if the initial accessor specifies SHARING GET and SHARING PUT, subsequent accessors must specify ACCESS GET to read the file, and and ACCESS PUT to write new records to the file (read access is implied by all four types of write access).

Table 7–3 presents the values that the initial accessor of a file can specify for file sharing to permit access to subsequent accessors.

Table 7–3 Initial File Sharing and Subsequent File Access

Initial Accessor Sharing	Subsequent Accessor Access
SHARING PROHIBIT	No access allowed
SHARING GET[1]	ACCESS GET[1]
SHARING DELETE	ACCESS DELETE
SHARING PUT	ACCESS PUT
SHARING UPDATE	ACCESS UPDATE

[1] Implied related operation

Because the initial accessor can specify multiple SHARING values, a subsequent accessor whose ACCESS values match one, some, or all of the initial accessor's SHARING values is allowed access; however, when the subsequent accessor specifies an ACCESS value that the initial accessor did not specify as a SHARING value (an exception is SHARING GET, which is implied), access is denied to the subsequent accessor.

In addition to comparing the file access values that subsequent accessors specify with the file-sharing values specified by the initial accessor, the values that subsequent accessors specify must be compatible with values specified by the initial accessor. Table 7–4 shows the file-sharing values that subsequent accessors must specify to access the file.

Table 7–4 Initial File Access and Subsequent File Sharing

Initial Accessor Access	Subsequent Accessor Sharing
ACCESS GET[1]	SHARING GET[1]
ACCESS DELETE	SHARING DELETE
ACCESS PUT	SHARING PUT
ACCESS UPDATE	SHARING UPDATE

[1] May be implied a related operation

Because the initial accessor can specify multiple ACCESS values, a subsequent accessor whose SHARING values match all of the initial accessor's ACCESS values is allowed access; however, when the subsequent accessor specifies a SHARING value that the initial accessor did not specify as an ACCESS value (an exception is ACCESS GET, which is implied), access is denied.

7.1.3 User-Interlocked Interprocess File Sharing

User-interlocked interprocess file sharing allows one or more application programs to write records to a sequential file residing on a disk device or to a file on a disk device that is open for block I/O processing. It cannot be used with relative and indexed files currently opened for record access. (For record access to relative and indexed files, VMS RMS transparently controls the reading and writing of buffers to the file and always maintains current end-of-file information.)

All sequential files that reside on disk devices may be write shared with user-provided interlocks. To use this feature, you must specify SHARING USER_INTERLOCK (set the FAB$B_SHR field FAB$V_UPI bit). Note that when this option is specified, VMS RMS does not attempt to control the reading and writing of I/O buffers across processes, nor does it maintain end-of-file information. Thus, you must use the Flush service (or VAX language equivalent, if any) to force the writing of modified I/O buffers and to rewrite the record attributes (including end-of-file information) in the file header. Processes that open the file after that point obtain the new end-of-file information. Note also that record attributes are rewritten whenever a file is closed. The last write accessor to close the file must also be the last accessor to have extended the file. If not, end-of-file information is written by another write accessor. Read accessors of a shared sequential file can update their internal end-of-file context by closing and reopening the file.

No form of record locking is supported for this type of file sharing. Although record locking is not checked using user-interlocked interprocess file sharing, file locking is checked. For instance, if you or another user specify SHARING NONE, it is likely that one of you will be denied access to the file.

If a process tries to implement the truncate service when closing a *sequential* file, it must have sole *write access* to the file. If other processes have *write access* to the file, VMS RMS does not close it and it remains accessible to other processes. If other processes have the file open for *read access*, VMS RMS defers the truncation until the final process having *read access* closes the file.

Similarly, if a process tries to implement the truncate-on-put option when inserting a record into a *sequential* file, it must have sole access to the file. If other processes have access to the file, VMS RMS does not insert the record.

7.2 Record Locking

Synchronized access to records is required in a shared file environment where record streams may compete for access to records. The VMS operating system implements synchronized access using record locking. That is, record access conflicts are resolved by locking the record until the final competing record stream processes the record. This ensures that a program may add, delete, or modify records without interference and that when a record operation is finished, the data is consistent.

Note: **VMS RMS record locking differs from RMS Journaling record locking. If your application program uses Recovery Unit Journaling, see the *VAX RMS Journaling Manual* for details.**

The VMS operating system allows you to determine whether the application program provides record locking or whether VMS RMS provides record locking. Processes accessing the file make this choice by specifying appropriate sharing attributes and access attributes in the FAB as described in Section 7.1. In general, VMS RMS enables record locking when record modifications are permitted in a shared file environment.

VMS RMS provides record locking for all file organizations and uses the VMS lock manager to keep conflicting record streams from updating a record simultaneously. The rest of Section 7.2 describes VMS RMS record locking.

7.2.1 Default Record Locking

You can specify various record locking options in the RAB when you access a record by way of a record stream. If you do not explicitly specify any record locking options when you access a record, VMS RMS uses default record locking to automatically and transparently lock and unlock shared records. Default record locking does not require special handling of locks in the application program.

In a typical record locking scenario, an application program calls a VMS RMS service to access and lock a record. The application program then processes the locked record. When it finishes processing the record, the application program calls the appropriate VMS RMS service to finish processing and unlock the record.

The following scenario illustrates processing an existing record:

1. The application program invokes the Get service to access the record, lock the record for exclusive access, and return the record to the application program.
2. The application program modifies the locked record. Other record streams that try to access the record using default record locking get a record-locked error. This prevents the locked record from being accessed and modified before the application program finishes modifying it.
3. The application program invokes the Update service to store the modified record in the file and remove the lock on the modified record, thereby making the record available to other record streams.

When VMS RMS provides record locking, the Get, Find and Put services apply locks. The Get service and the Find service normally return with a record locked, but the Put service returns with the record unlocked unless you specify the manual-unlocking option.

When the application program uses default record locking, VMS RMS automatically unlocks the locked record when one of the following events occurs:

- Another record is accessed (Get service and Find service).
- The current record is updated (Update service).
- The current record is deleted (Delete service).
- The record stream is disconnected (Disconnect service).
- The file is closed (Close service).
- The record stream is positioned to the beginning of the file (Rewind service).
- A new record is added to the file (Put service).

- The record lock is explicitly removed (Release service or Free service).
- An error occurs during a record operation.

Note that a sequential Get service immediately following a Find service does not unlock the record because it accesses the same record.

7.2.2 Record Locking Options

VMS RMS record locking options can be divided into three groups:

- Options that specify the access allowed by other record streams
- Options that control record conflicts between record streams
- Miscellaneous options

All record locking options are specified by RAB input to the accessing service. All record locking options apply to the Get service and the Find service, and most record locking options apply to the Put service. You can specify a different set of record locking options each time the record stream accesses a record.

This section describe the types of record access allowed by each record locking option. It also provides some examples of when an application program might select a particular record locking option. The following four record locking options control record access by other record streams:

- Exclusive locking
- Write locking
- Read locking
- No locking

To update or delete a record, a record stream must have an exclusive lock or a write lock on the record.

7.2.2.1 Exclusive Locking

By default, VMS RMS performs exclusive-locking. With exclusive locking, only the initial record stream is permitted to access the record for reading or writing until the lock is released. Any other record stream that tries to read or write the record by applying a lock is denied access. When a record stream is denied access because of a locked record, the requesting service returns a locked-record status (RMS$_RLK).

The only way a record stream can read an exclusively locked record is by using the read-regardless option (see Section 7.2.3.3).

Most application programs use exclusive locking because it requires minimal programming and provides maximum protection when modifying and reading records. Note, however, that contention is apt to be greatest when a record stream uses the exclusive-locking option.

See Section 7.2.1 for an example of how VMS RMS uses exclusive locking for an application program that is modifying a record.

7.2.2.2 Write Locking

The write-locking option allows the record stream that locks a record to modify the record. This option explicitly prohibits other record streams from having write-lock access or exclusive lock access, both of which imply an intent to modify the record. The write-locking option also denies read-lock access to other record streams because a read-lock access is incompatible with a record stream that is modifying the record.

Contending record streams can read the record using the no-locking option or the read-regardless option (see Section 7.2.3.3). When a contending record stream successfully reads a write-locked record using the no-locking option, the accessing service returns a success status.

Typically, an application program uses the write-locking option when it wants the record to remain in a consistent state while the application program is modifying the record.

7.2.2.3 Read Locking

The read-locking option permits other record streams to access the record for reading but denies access to any record stream that attempts to access the record for making modifications.

No record stream is allowed to access a read-locked record for making modifications to the record until all record streams that have a read lock release the record. Any record stream that attempts to access a read-locked record using either the exclusive-locking option or the write-locking option are denied access. The requesting service returns a completion status record to the application program indicating that the record was locked (RMS$_RLK) and the requesting record stream was denied access.

Contending record streams can read the record using the read-locking option, the no-locking option or the read-regardless option (see Section 7.2.3.3). When a contending record stream successfully accesses a read-locked record using the read-locking option or the no-locking option, the accessing service returns a success status.

Typically, an application program uses the read-locking option when it wants the record to remain in a consistent state while reading the record but does not intend to modify the record.

7.2.2.4 No Locking

The no-locking option specifies that the requesting record stream does not want to lock the record. This is the most compatible locking option because it permits the requesting record stream to have access to all locked records except for records that are locked for exclusive access. It also permits other record streams to apply any type of lock to the record. Using this option minimizes contention and may improve application program performance.

By implication, a record stream that uses the no-locking option can only access the record for reading. When a record stream uses the no-locking option to access a record, the invoked service returns with the record unlocked.

Note that when a record stream selects the no-locking option, VMS RMS momentarily locks the record to determine whether or not the record is already locked by another record stream. This is required in order to determine if access is allowed. If the record is not locked, the requesting service returns a completion status indicating a successful access. If the record has an exclusive lock, the access is denied and the requesting service returns a completion status indicating the record is locked (RMS$_RLK). If the record has a write lock or a read lock, the requesting service reads the record and returns a completion status indicating that the record was locked but a read was permitted (RMS$_OK_RLK).

If you specify the no-locking option together with the manual-unlocking option, the no-locking options takes precedence. That is, if you specify both options to the service that accesses the record, the service returns control to the application program with the record unlocked. See Section 7.2.4.1 for a description of the manual-unlocking option.

7.2.2.5 Put Service Considerations

Since the Put service adds a new record, the application program does not have to access an existing record. However, because adding a record is a multistep process, the record that is being added must be locked until the entire process is finished.

The scenario for adding a record to a file begins with the application program moving a record into its buffer. Next, the application program calls the Put service, which locks the record while it moves it from the application program buffer to the file. When the record is in the file, the Put service typically unlocks the record, making it available to other record streams. The locking process is transparent at the program level unless the application program selects the manual-unlocking option.

If a record stream tries to add a record using the no-locking option, the Put service ignores the option and adds the record.

7.2.2.6 Summary

This section provides two tables to summarize the information described in Section 7.2.2.1 through Section 7.2.2.5.

The record locking options that control record access exhibit varying degrees of compatibility. Table 7–5 summarizes access control locking compatibility by comparing the type of access being requested by a record stream with the current lock held by another record stream. The table does not take into account miscellaneous record locking options, notably the read-regardless option.

Table 7–5 Compatibility of Record locking options

	Current Lock Held by Another Record Stream			
Requested Access	EXCLUSIVE	WRITE	READ	None
EXCLUSIVE	NO	NO	NO	YES
WRITE	NO	NO	NO	YES
READ	NO	NO	YES	YES
NO LOCK	NO	YES[1]	YES[1]	YES

[1] RMS$_OK_RLK is returned.

The next table lists record locking options that control record access and how you select each option through the FDL and VMS RMS interfaces.

Option	How to select	
Exclusive locking	FDL: VMS RMS:	This is the default when you do not select write locking, read locking or no locking.
Write locking	FDL: VMS RMS:	CONNECT LOCK_ON_WRITE RAB$L_ROP RAB$V_RLK
Read locking	FDL: VMS RMS:	CONNECT LOCK_ON_READ RAB$L_ROP RAB$V_REA
No locking	FDL: VMS RMS:	CONNECT NOLOCK RAB$L_ROP RAB$V_NLK

7.2.3 Handling Record Locking Conflicts

Application programs that use shared files must be able to handle record locking conflicts that may occur when two or more record streams try to access the same record. VMS RMS provides three options for handling record locking conflicts:

- You can have the application program handle the record-locked error status (RMS$_RLK) returned by VMS RMS when a record stream is denied access to a record.
- You can have the requesting service wait for access (wait-if-locked option).
- You can have the requesting service ignore the lock (read-regardless option).

The following table lists the options for having VMS RMS handle record locking conflicts and how you select each option through the FDL and VMS RMS interfaces.

Option	How to select	
Wait if locked	FDL: VMS RMS:	CONNECT WAIT_FOR_RECORD RAB$L_ROP RAB$V_WAT
Wait timeout period	FDL: VMS RMS:	CONNECT TIMEOUT_ENABLE and CONNECT TIMEOUT_PERIOD RAB$L_ROP RAB$V_TMO and RAB$B_TMO
Read regardless	FDL: VMS RMS:	CONNECT READ_REGARDLESS RAB$L_ROP RAB$V_RRL

The following sections describe each of these options.

Example 7–1 Designing a Pause between Attempts to Access a Record

```
   .
   .
   .
10$:    $GET     RAB=INRAB              ; Get the record
        BLBS     R0,GOT_RECORD          ; Branch on success
        CMPL     R0,#RMS$_RLK           ; Record-locked error?
        BNEQ     ERROR                  ; Quit on other errors
        PUSHL    #^F1.0                 ; Pause for
        CALLS    #1,G^LIB$WAIT          ; one second
        BLBC     R0,ERROR               ; Quit on error
        BRB      10$                    ; Try again for record
   .
   .
   .
```

7.2.3.1 Handling the Record-Locked Error

When a VMS RMS service is denied record access because of a record conflict, it returns a record-locked error status (RMS$_RLK) that indicates the access attempt failed because the record was locked. One option is to have the application program pause briefly, and then try again to access the record.

Example 7–1 contains a program fragment written in VAX MACRO that demonstrates one method of implementing a short pause between attempts to access a locked record.

For more information about process control techniques, see the *VMS System Services Reference Manual.*

7.2.3.2 Waiting for Locked Records

Another option for handling record locking conflicts is to use the wait-if-locked option to wait for the locked record to be released. When you take this option, the accessing service does not return until the record is released or until a specified wait period expires.

The optional wait period is established using the wait-timeout-period option in conjunction with the wait-if-locked option. If the specified wait period expires before the requesting service obtains access to the locked record, the requesting service discards the request. The requesting service returns a completion status indicating that it waited for the locked record but was not granted access within the specified time period (RMS$_TMO).

If you select the wait-if-locked option and the requesting service must wait to access the record, it returns an alternate success status that indicates that it had to wait (RMS$_OK_WAT).

7.2.3.3 Reading Regardless of Lock

The third choice available to you for handling record locking conflicts involves using the read-regardless (of lock) option. This option allows the accessing service to ignore a lock that would normally prohibit read access. If a lock is granted under the specified record locking option, access is granted and the service returns with the specified lock. If the lock is denied, the read-regardless option allows the accessing service, Get or Find, to read the record, regardless of the lock. The service returns without a lock and with an alternate success status RMS$_OK_RRL.

An application program might use the read-regardless option to avoid record locking conflicts when a coordinated view of a record is not necessary. This option can also be used to continue sequential reads through a locked record.

Note that when you use the read-regardless option with the wait-if-locked option and a wait timeout period, VMS RMS acts on the read-regardless option only after the wait timeout expires.

7.2.4 Miscellaneous Record Locking Options

This section describes two miscellaneous record locking options—the manual-unlocking option and the lock-nonexistent-record option in a relative file.

7.2.4.1 Manual-Unlocking Option

The manual-unlocking option gives the application program explicit control over releasing a record lock established by the Get service, the Find service or the Put service as described in Section 7.2.1.

Even if you select the manual-unlocking option, VMS RMS unlocks affected records when a record stream is disconnected (Disconnect service), or when a file is closed (Close service). Other record operations, including operations that result in errors, do not unlock the record.

To manually release record locks, the application program can invoke the Free service to unlock all record locks currently held by a record stream, or it can invoke the Release service to selectively release record locks, using the record's RFA.

Manual unlocking is useful when you have to modify multiple records as part of a single transaction. For example, assume the application program must modify two related but separate records. Assume, too, that the modified first record must not be accessed by another record stream until modifications to the second record are completed.

While the program modifies the first record, it uses the manual-unlocking option to hold the lock on the modified first record. It then proceeds to modify the second record while still maintaining a lock on the first record. By using manual unlocking, the application program can restore the original contents of the first record if the update to the second record fails, thereby maintaining data integrity.

7.2.4.2 Lock-Nonexistent-Record Option

The lock-nonexistent-record option applies only to random accessing of relative files. Relative files have a static physical structure made up of record cells in contrast to sequential files and indexed files, which have a dynamic structure. The record cells may or may not contain records. A record may have been deleted from a cell, or the cell may be empty (that is, it never contained a record). In either case, the records cells are accessible to the application program.

Typically, if a record stream tries to access and lock an empty cell in a relative file using random access, the accessing service returns a record-not-found error status (RMS$_RNF). However, if the lock-nonexistent-record option is selected, the accessing service returns an alternative success status (RMS$_OK_RNF) indicating that the record stream successfully accessed a cell that never contained a record. If the cell contains a deleted record, VMS RMS returns the deleted record with an alternate success status (RMS$_OK_DEL) to indicate that a deleted record was accessed.

The lock-nonexistent-record option prevents other record streams from putting a record into an empty cell until the locking record stream puts a record in it or releases the record lock. Any other record stream that tries to access the cell to put data into it receives a record-locked status (RMS$_RLK). If the record stream that has the lock puts a record into the cell, VMS RMS returns an alternate success status (RMS$_OK_ALK) indicating that the cell was already locked. In general, the RMS$_OK_ALK status is returned when a VMS RMS service tries to lock a record that the current record stream has already locked. This also applies to the Put service, which locks and unlocks the record in one record operation.

The next table lists miscellaneous record locking options and how you select each option through the FDL and VMS RMS interfaces.

Option	How to select	
Manual unlocking	FDL: VMS RMS:	CONNECT MANUAL_UNLOCKING RAB$L_ROP RAB$V_ULK
Lock nonexistent record	FDL: VMS RMS:	CONNECT NONEXISTENT_RECORD RAB$L_ROP RAB$V_NXR

7.2.5 Record Locking Deadlocks

A deadlock occurs when there is a set of processes, and each process is waiting to access a record that is locked by another process in the set. The program stalls because none of the processes can acquire the record that it needs to complete its task and release its locks.

The VMS lock manager resolves the deadlock by arbitrarily denying one of the lock requests. When this occurs with a record lock, VMS RMS returns an RMS$_DEADLOCK status. The RMS$_DEADLOCK status is only returned if the wait-if-locked option is selected. If your application program does its own wait and retry handling, the deadlock will occur, but the lock manager will not be able to detect it.

The amount of time that lapses before VMS RMS takes action on the deadlock depends on the value specified in the DEADLOCK_WAIT system parameter. The default value for this system parameter is 10 seconds. For further details about how this parameter is set, see the *Guide to Setting Up a VMS System.*

For more information about the VMS lock manager, see the *VMS System Services Volume.*

7.3 Local and Shared Buffering Techniques

One of the key performance factors is record buffering, that is, the transfer of records between a storage device and an area of memory accessible to the application program. Between the storage device and the record buffer in the appliction program, however, is an intermediate buffer area that VMS RMS maintains. An intermediate buffer area is usually associated with each process; you can also specify a shared buffer area for a shared file.

7.3.1 Record Transfer Modes

For synchronous and asynchronous record operations, VMS RMS provides two record transfer modes: move mode and locate mode.

In move mode, VMS RMS copies a record from an I/O buffer into a buffer that you specify. For input operations, data is first read into the I/O buffer from a peripheral device (such as a disk), then moved to your application program buffer for processing. For output operations, you first build the record in your application program buffer; then VMS RMS moves the record to the I/O buffer that is used to transfer the record to disk.

In locate mode, VMS RMS allows the application program to access records directly in a VMS RMS I/O buffer by providing the address of the returned record as the internal VMS RMS buffer location instead of an application program buffer location (field RAB$L_RBF). Usually, this reduces program overhead because records can be processed directly within the I/O buffer. Locate mode is only available for input operations. Because it may not always be possible to use locate mode, you must supply an application program buffer for cases in which move mode must be used, even though you specify locate mode (see the *VMS Record Management Services Manual*).

Other VMS RMS facilities allow programs to control I/O buffer space allocation or to simply leave all space management to VMS RMS. The following sections briefly describe buffering.

7.3.2 Understanding Buffering

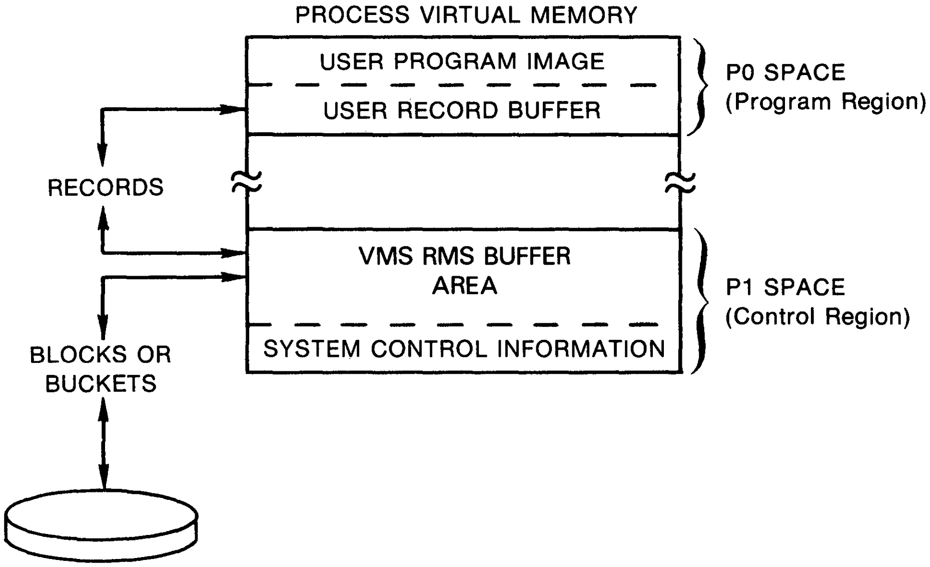
Record processing in VMS RMS appears to your program as the movement of records directly between a file and the program itself. In fact, VMS RMS uses internal memory areas called I/O buffers to read or write blocks or buckets of data. Transparent to your program, VMS RMS transfers blocks or buckets of a file into or from an I/O buffer. Records within the I/O buffer are then made available to the program when VMS RMS transfers the records between the I/O buffer and the application program's record buffer.

The unit of data transfer between a file and the I/O buffers depends on the file organization. For the sequential organization, VMS RMS reads and writes a block or series of blocks. For relative and indexed organizations, VMS RMS reads and writes buckets.

The relationship between the application program and the I/O buffers that VMS RMS maintains is shown in Figure 7–2. As illustrated, the application program resides in the P0 region of process address space. The VMS RMS-maintained buffer area, together with VMS RMS-maintained control information, resides in the P1 region.

Note that VMS RMS normally overflows into P0 space and that the linker provides options for controlling the overflow. Note, too, that linker options are available for allocating additional buffer space in the P0 region, if needed. See the *VMS Linker Utility Manual* for details.

Figure 7–2 VMS RMS Buffers and the Application Program



ZK-1993-84

The specified record buffer contains the record to be read or written, and VMS RMS maintains the rest of the block in application program process space in a VMS RMS-controlled area of the program.

For optimum performance, consider the number of buffers carefully. The defaults calculated by VMS RMS are few and may be adequate for access to small files. For example, it is not unusual to specify many buffers when processing a large indexed file, yet the default number of buffers VMS RMS provides is only two.

The CONNECT secondary attribute MULTIBUFFER_COUNT (VMS RMS field RAB$B_MBF) establishes the number of buffers, but the FILE secondary attribute GLOBAL_BUFFER_COUNT (VMS RMS field FAB$W_GBC) specifies the number of *global buffers* as described in Section 7.3.6.

Often the best way to achieve optimum buffering for a particular application program is to use combinations of buffer sizes and numbers of buffers. One approach is to time each combination and measure the number of I/O operations. Then consider the amount of memory used before you choose the one that improved application program performance the most.

With buffering, the goal is to use a buffer size and number of buffers that improves application program performance without exhausting the virtual memory resources of your process or system. Keep in mind the trade-offs between file I/O performance and exhausting memory resources. The buffers used by a process are charged against the process's working set. You should avoid allocating so many buffers that the CPU spends excessive processing time paging and swapping. For performance-critical application programs, consider increasing the size of the process working set and adding additional memory.

The system manager should monitor the paging and swapping activity of the application program's process and selected other processes to avoid improving the performance of the target application program at the expense of other application programs. Have your system manager consult the *Guide to VMS Performance Management* for system tuning information. For information about the resources needed for file applications, refer to Section 1.5.

When records are likely to be accessed sequentially, a large buffer (or buffers) should be used. Contiguous records in a file are read into memory in one or more blocks for sequential files or in buckets (multiblock units) for relative and indexed files. After the blocks or buckets are read into the buffer area provided by VMS RMS, later access to adjacent records would access records in the same block or bucket in the buffer. This eliminates additional I/O and improves performance. When a record is needed that is not in the current buffer cache, one of the buffers is replaced by the blocks or the bucket that contains the new record.

When records in the file are repeatedly accessed, using more than one buffer can hold the previously accessed records in memory longer and eliminate an I/O operation when the program accesses the records again.

The buffers that the application program requests VMS RMS to allocate for its use are referred to as a *buffer cache* and can be thought of as a buffer pool for your process that VMS RMS uses to locate records first before attempting I/O to the target device. When many processes share a file, the program can use a shared global buffer cache (see Section 7.3.6.)

7.3.3 Buffering for Sequential Files

With sequential files, the number of buffers and the size of the buffers can be specified at run time. You specify the number of buffers with the FDL attribute CONNECT MULTIBUFFER_COUNT (VMS RMS control block field RAB$B_MBF) and you specify the buffer size with the FDL attribute CONNECT MULTIBLOCK_COUNT (VMS RMS control block field RAB$B_MBC).

Sequential files provide an option that alternately uses two buffers. One buffer holds records to be read from the disk or written to the disk. The other buffer awaits I/O completion. This is called *read-ahead and write-behind processing* and should be considered for sequential access to sequential files. The number of buffers (CONNECT MULTIBUFFER_COUNT) should be specified as 2. The length of the buffers used for sequential files is determined by the specified multiblock count (CONNECT MULTIBLOCK_COUNT). For sequential access to a sequential file, the optimum number of blocks per buffer depends on the record size, but a value such as 16 is usually appropriate.

To see the default buffer count for the current process, use the DCL command SHOW RMS_DEFAULT. To set the default buffer count for the current process, use the DCL command SET RMS_DEFAULT/SEQUENTIAL /BUFFER_COUNT=n, where *n* is the number of buffers.

7.3.4 Buffering for Relative Files

With relative files, buckets, not blocks, are the unit of transfer between the disk and memory. The bucket size is specified when the file is created, although the bucket size of an existing file can be changed by converting the file (see Chapter 10).

The bucket size is specified by the FDL attribute FILE BUCKET_SIZE (VMS RMS control block field FAB$B_BKS or XAB$B_BKZ). When choosing this value, you should consider whether or not the file is usually accessed randomly (small bucket size), sequentially (large bucket size), or both (medium bucket size), as described in Chapter 2.

The number of buffers (CONNECT MULTIBUFFER_COUNT, RAB$B_MBF) is specified at run time. The type of record access to be performed determines the best use of buffers. The two extremes of record access are that records are processed completely randomly or completely sequentially. Also, there are cases in which records are accessed randomly but may be reaccessed (random with temporal locality) and cases in which records are accessed randomly but adjacent records are likely to be accessed (random with spatial locality).

For completely random or sequential access, a single buffer should be specified. In a processing environment in which the program processes records randomly and sometimes reaccess records, use multiple buffers to keep the reaccessed records in the buffer cache.

When records are accessed randomly and adjacent records are apt to be accessed, you should specify a single buffer. However, if your program is processing a file with small bucket sizes, you should consider specifying more buffers. When the file is likely to be accessed by several methods, you should consider a compromise of the number of buffers and bucket sizes.

When adding records to a relative file, consider choosing the deferred-write option (FDL attribute FILE DEFERRED_WRITE; FAB$L_FOP field FAB$V_DFW). With this option, the buffer (memory-resident bucket) into which the records have been moved is not written to disk until the buffer is needed for other purposes or until the file is closed. Note that if you use the deferred-write option, there is a risk that data may be lost if a system crash occurs before the records are written to disk.

To see the current process-default buffer count, use the DCL command SHOW RMS_DEFAULT. To set the process-default buffer count, use the DCL command SET RMS_DEFAULT/RELATIVE/BUFFER_COUNT=n, where *n* is the number of buffers.

7.3.5 Buffering for Indexed Files

With indexed files, buckets (not blocks) are the units of transfer between the disk and memory. The bucket size is specified when the file is created, although the bucket size of an existing file can be changed by converting the file (see Chapter 10).

The bucket size is specified by the FDL attribute FILE BUCKET_SIZE (VMS RMS control block field FAB$B_BKS or XAB$B_BKZ), as described in Chapter 2.

When accessing indexed files, it is important to remember that the index portion of the file must be read by VMS RMS to locate the desired record. The algorithm used by VMS RMS places a higher priority for the higher-level buckets of the index in the buffer cache. Thus, the highest levels of the index remain in the buffer cache, while the buffers that may have contained the actual data buckets and the lower-level index buckets are reused to contain other buckets. That is, the buffers that are reused first contain either data or lower-level index buckets, which are the first to be discarded from the buffer cache.

When accessing indexed files, the number of buffers (CONNECT MULTIBUFFER_COUNT, RAB$B_MBF) is specified at run time and recommended values can vary greatly for different application programs. When records are processed randomly, use as many buffers as your process working set can support to cache additional index buckets. When records are accessed sequentially, even after locating the first record randomly, use a small multibuffer count, such as the default of 2 buffers.

Many application programs access files using a mixture of completely random and completely sequential processing. For such application programs, a compromise of the above number of buffers is recommended.

When adding records to an indexed file, consider choosing the deferred-write option (FDL attribute FILE DEFERRED_WRITE; FAB$L_FOP field FAB$V_DFW). With the deferred-write option, the buffer into which the records have been moved is not written to disk until the buffer is needed for other purposes or until the file is closed. This option, however, may cause records to be lost if a system crash should occur before the records are written to disk.

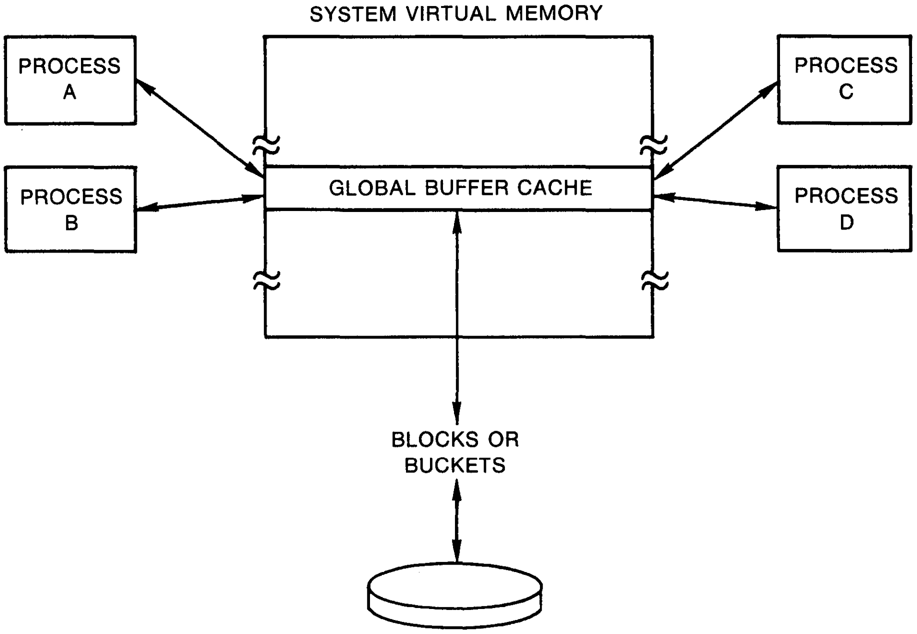
To see the current process-default buffer count, use the DCL command SHOW RMS_DEFAULT. To set the process-default buffer count, use the DCL command SET RMS_DEFAULT/INDEXED/BUFFER_COUNT=n, where *n* is the number of buffers.

7.3.6 Using Global Buffers for Shared Files

Two types of buffer caches are available using VMS RMS: local and global. Local buffers reside within process (program) memory space and are not shared among processes, even if several processes access the same file and read the same records. Global buffers, which are designed for application programs that access the same files and perhaps the same records, do not reside in process memory space.

If several processes share a file, you should specify that the file uses global buffers. A global buffer is an I/O buffer that two or more processes can access in conjunction with file sharing. If two or more processes request the same information from a file, each process can use the global buffers instead of allocating its own process-local buffers. Figure 7–3 illustrates the use of global buffers.

Figure 7–3 Using Global Buffers for a Shared File



ZK-1994-84

Unlike local buffers, global buffers can be accessed by multiple processes accessing the same file. When a record requested by one process is located in a global buffer, the record can be transferred directly from the global buffer to the program, eliminating an I/O read operation. Note that if the previous accessor modified the record, VMS RMS writes the buffer to disk before returning the record to the new accessor. This ensures that the modified bucket in memory matches its counterpart on the disk.

There are two situations in which global buffers cannot be used for shared files. When a process permanent file is being accessed, VMS RMS does not use global buffers (no error is returned). When an image is linked using the LINK option keyword IOSEGMENT=NOP0BUFS (rarely used), VMS RMS does not use global buffers.

Even if global buffers are used, a minimal number of local buffers should be requested, because, under certain circumstances, VMS RMS may need to use local buffers. When attempting to access a record, VMS RMS looks first in the global buffer cache for the record before looking in the local buffers; if the record is still not found, an I/O operation occurs. When using the deferred-write option with global buffering enabled, the number of buckets that can be buffered without I/O is equal to the number of local buffers; thus, the use of more than the minimum number of local buffers should be considered.

You can specify the number of global buffers two ways: by using a preset file default or by having the first process that accesses the file specify the value at run time. To set the file default (maintained in the file header), use the DCL command SET FILE/GLOBAL_BUFFERS=n where *n* is the number of buffers.

To set the global buffer value at run time, the first process to connect to the file with the FILE GLOBAL_BUFFER_COUNT attribute (VMS RMS control block field FAB$W_GBC) greater than 0 can set this value. The default value returned in the FAB$W_GBC field following an Open (or Create) service may be altered if unacceptable before invoking the Connect service. When a previous or subsequent application program attempts to open and connect to the file, the global buffer count determines whether or not that process uses global buffers. If the value is 0, that process uses only local buffers; if the value is greater than 0, that process uses global buffers along with other processes. Refer to the *VMS Record Management Services Manual* for additional information on the use of the FAB$W_GBC field and Connect service. An example of a routine that sets the global buffer count after opening a file is provided in Example 5–2.

To request that the global buffer cache be read-only, specify SHARING GET and SHARING MULTISTREAMING attributes (FAB$B_SHR field FAB$V_SHRGET and FAB$V_MSE).

When modifying an application program to use global buffers, consider using more global buffers and slightly larger bucket sizes if records are processed randomly. For application programs with many users, consider allocating a number of global buffers equal to the number of local buffers used previously, multiplied by number of users (if resources permit):

No. Global Buffers = No. Local Buffers x Average No. Users

When using an indexed file, if the index structure is small and the number of users is many, consider allocating enough global buffers to keep the entire index structure in memory.

8 Record Processing

This chapter describes record processing to help you use the run-time record operations described in Chapter 9. This chapter provides information about the following subjects:

- Record operations appropriate to high-level languages
- Accessing records for file organizations
- Record environment as it relates to record positioning
- Synchronous versus asynchronous record operations

8.1 Record Operations

Record operations are performed by VMS RMS services, which are classified as primary or secondary services. Primary services have functional equivalents in high-level language record operations, whereas secondary services are specific to VMS RMS functions.

Section 8.2 describes the five primary services. For a brief description of the secondary services, refer to Section 8.3, and for more detailed descriptions of the secondary services, refer to the *VMS Record Management Services Manual.*

8.2 Primary VMS RMS Services

This section describes the five VMS RMS services that are functionally similar to related high-level language operations. The following list provides a brief description of each of these services and cites the similarities to high-level languages.

Find — The Find service locates an existing record in the file. It does not return the record to your program; instead it establishes the record's location as the current-record position in the record stream. The Find service, when applied to a disk or magnetic tape file, corresponds to the FIND statement in BASIC and FORTRAN, the START statement in COBOL, the FIND or LOCATE statements in Pascal, and the READ statement with the SET keyword for PL/I.

Get — The Get service returns the selected record to your program. The Get service, when applied to a disk or magnetic tape file, corresponds to (is used by) the GET statement in BASIC; the READ statement in COBOL, FORTRAN, and PL/I; and the GET statement (and others) in Pascal.

Put — The Put service inserts a new record in the file. The Put service, when applied to a disk or magnetic tape file, corresponds to the PUT and PRINT statements in BASIC; the WRITE statement (and others) in COBOL; the WRITE statement in FORTRAN and PL/I; and the PUT and WRITELN statements in Pascal.

Update The Update service modifies an existing disk file record. The Update service corresponds to the UPDATE statement in BASIC and Pascal and to the REWRITE statement in COBOL, FORTRAN, and PL/I.

Delete The Delete service erases records from relative disk files and indexed disk files. The Delete service corresponds to the DELETE statement in BASIC, COBOL, FORTRAN, Pasçal, and PL/I.

A single statement in a VAX language may correspond to one or several VMS RMS record-processing service calls. For example, the COBOL statement DELETE uses the VMS RMS Delete service during sequential record access, but it uses the Find and Delete services during random record access.

File organization in part determines the types of record operations that a program can perform. Table 8–1 shows the major record operations that VMS RMS permits for each file organization.

Table 8–1 Record Operations and File Organizations

		File	
Record Operation		**Organization**	
Permitted	**Sequential**	**Relative**	**Indexed**
Get	Yes	Yes	Yes
Put	Yes[1]	Yes	Yes
Find	Yes	Yes	Yes
Delete	No	Yes	Yes
Update	Yes[2]	Yes	Yes

[1] In a sequential file, VMS RMS allows records to be added at the end of the file only. (Records can be written to other points in the file by using a Put service with the update-if option.)

[2] When performing an Update service to a sequential file containing fixed-length records, you cannot change the length of the record. The Update service is only allowed on disk devices.

The remainder of this section briefly describes the record retrieval (Find and Get) services, the record insertion (Put) service, the record modification (Update) service, and the record deletion (Delete) service. Note that all references to VMS RMS services imply applicability to similar functional capabilities found in high-level languages.

8.2.1 Locating and Retrieving Records

You can use the Find and Get services to locate and retrieve a record. The Find service locates a record and establish its location as the current-record position in a record stream but does not return the record to a buffer. The Get service locates the record, establishes its location as the current-record position in the record stream, and returns it to the buffer area you specify.

If you use the Get service, you must allocate a buffer area in the data portion of your program to store the retrieved record by defining an appropriate variable or multivariable record structure in the program.

Note: **When you invoke the Get service, VMS RMS takes control of the record buffer and may modify it. VMS RMS returns the record size but it can only guarantee record integrity from the access point to the end of the record.**

In addition to retrieving the record, VMS RMS returns to your program the length of the record (in control block field RAB$W_RSZ, record size) and the file address of the record (in control block field RAB$L_RBF, record buffer). If you direct VMS RMS only to locate the record, it does not write the record into your buffer. Instead, it sets the RAB$W_RSZ and RAB$L_RBF fields to point to an internal buffer where the record is located.

When using indexed files, you may need to allocate a buffer for the desired key and to specify its length. When using high-level VAX languages, the language's compiler may automatically handle the allocation and size specification of the record buffer and the key buffer.

In some applications, you can minimize record I/O and improve performance by using the Find service instead of the Get service. For example, a process does not have to retrieve a record when it is preparing to invoke the Update, Delete, Release, or Truncate service. If a process intends to update a record that is accessible to other processes, it should lock the record until it completes the update.

For interactive applications where the user verifies that the appropriate record is being accessed before deleting it or updating it, the program should use the Get service instead of the Find service.

In some situations, a process may use two services and two types of record access to retrieve a set of records. For example, the process might use the Find service and random access mode to locate the first record in the set, then switch to the Get service for sequentially retrieving the records in the set.

An efficient use of the Find service is to create a table of RFAs (record file addresses) to be used for rapidly accessing the records in the same file.

Record retrieval operations are typically used to repetitively read and process a set of records. As part of this type of operation, your program should check for an end-of-file condition after each Find or Get service.

For more information about the Find and Get services, refer to the *VMS Record Management Services Manual.*

8.2.2 Inserting Records

The Put service adds a record to the file. Within the data portion of your program, you must provide a buffer for the record that is to be added. The program must also supply the length of each record to be written when calling VMS RMS directly. This is a constant value with fixed-length records but varies from record to record when adding variable-length or VFC records. When using high-level VAX languages, however, the language's compiler may automatically handle record buffer size specification or supply a means to simplify its specification.

The current-record position is especially important when adding records to a sequential file. VMS RMS establishes the current-record position at the end of file for any record stream associated with a file opened for adding records. To add records to a relative file or to an indexed file, use random access (by key or record number), unless the program adds records sequentially by a specified ordering of primary keys or by relative record number.

The update-if option replaces an existing record using the Put service when you choose random access mode. When superseding existing records, consider using this option to add records to a relative or indexed file. A program can use the update-if option to update a record in a sequential file that is being accessed randomly by relative record number.

Be careful with automatic record locking when you use this option for a shared file because the Put service briefly releases record locks applied by the Get or Find service before the Update operation begins. This could permit another record stream to delete or update the record between the time that the program invokes the Put service and the beginning of the Update service.

Consider using the Update service instead of the Put service with the update-if option to update an existing record in a shared file.

When a file contains alternate keys with characteristics that prohibit duplicate values, the application must be prepared to handle duplicate-alternate-key errors.

For more information about the Put service, refer to the *VMS Record Management Services Manual.*

8.2.3 Updating Records

The Update service modifies an existing record in a file. Your program must first locate the appropriate record and optionally retrieve the record itself, by either calling the Find service or the Get service. As with the Put service, your program must provide a buffer within the data portion of the program to hold the record that is to be updated.

The program must also supply the length of each record to be written when calling VMS RMS directly. This is a constant value when updating fixed-length records but varies from record to record when updating variable-length records or VFC records. Note that some high-level VAX language compilers may automatically handle record buffer allocation and size specification or may supply a means to simplify its specification.

Your program must establish the current-record position before it updates a record. If the file is shared, the service that establishes the record position should also lock the record.

When you update indexed file records, take care not to alter the value of any key field that has been specified as unchangeable, for example, the primary key. To change the value of a record's primary key, you must replace the existing record with a new record having the desired primary key value. You can do this using the Put and Delete services respectively; or, where applicable, you may use the Put service with the update-if (RAB$L_ROP RAB$V_UIF) option.

When updating records in an indexed file, a key of reference does not need to be specified.

For more information about the Update service and record-processing options, refer to the *VMS Record Management Services Manual.*

8.2.4 Deleting Records

The Delete service removes a record from the file. You cannot delete individual records from sequential files, but you can truncate sequential files using the Truncate service. As with the Update service, the Delete service must be preceded by a Find or Get service to establish the current-record position.

When deleting records from an indexed file with alternate indexes, you can specify the fast-delete option to reduce the amount of time needed to delete a record. When you invoke the Delete service and specify the fast-delete option, VMS RMS does not attempt to remove any of the pointers from alternative indexes to the deleted record.

You improve performance by postponing the processing needed to eliminate the pointers from alternative indexes to the record. However, there are disadvantages to using the fast-delete option. First, the unused pointers from the alternate indexes result in a corresponding waste of space. Second, if the program later tries to access the deleted record from an alternate index, VMS RMS must traverse the pointer linkage, find that the record no longer exists, and then perform the processing that was avoided originally with the Delete service.

You should consider the fast-delete option only if the immediate improvement in performance is worth the added space and overhead. Typically, you consider the fast-delete option for indexed files that implement alternate keys and require frequent maintenance.

Conversely, you should avoid the fast-delete option for most read-only indexed files and for indexed files that are infrequently updated.

For more information about the Delete service, refer to the *VMS Record Management Services Manual.*

8.3 Secondary VMS RMS Services

This section provides very brief descriptions of the VMS RMS secondary services. Note that each of the services performs a specialized function with few options.

Connect	Allows you to connect to a single record stream or to multiple record streams.
Disconnect	Allows you to disconnect a record stream. This is done implicitly when a file is closed, but when using multiple record streams, you may want to disconnect one record stream but not others.
Flush	Writes modified I/O buffers and file attribute information maintained in memory to the file.
Free	Releases all record locks established by the current record stream.
Next Volume	Continues the next volume of a magnetic tape volume set. This service applies only to sequential files.
Release	Releases the record lock on the current record.

Rewind	Positions the record stream context to the first record of the file.
Truncate	Truncates a file beginning with the current record, effectively deleting it and all remaining records. This service applies only to sequential files.
Wait	Awaits the completion of an asynchronous record operation (or Connect service).

In addition to the record processing services, a variety of file-processing services is also available. For more information about both types of processing services and the options that apply to each, see the *VMS Record Management Services Manual.*

8.4 Record Access for the Various File Organizations

To retrieve or insert a file record for a particular record stream, your program must specify either sequential or random access.

Sequential access can be used with all file organizations. For sequential files, sequential access implies that records are accessed according to their physical position in the file. For relative files, sequential access implies that records are accessed according to the ascending order of relative record numbers. In indexed files, sequential access implies that records are accessed according to a specified ordering of values for a particular key or keys.

Random access is defined as one of the following:

- Random access by key for indexed files implies that VMS RMS uses the specified key value (contained within the record itself) to locate the desired record.
- Random access by relative record number for relative files and for sequential files having fixed-length records implies that the specified relative record number is used to locate the desired record. The relative record number does not necessarily reside in the record.
- Random access by RFA implies that the specified RFA is used to locate the desired record. This access mode is supported for all three file organizations and is normally available only to programs written in VAX MACRO or similar low-level VAX languages.

Record access is specified using language statements or by establishing the appropriate control block field values (not offset values) in the RAB.

Note: No FDL attributes are provided for specifying record access.

The appropriate RAB values in the access mode specification field, identified by the symbolic offset RAB$B_RAC, are listed below.

- You specify sequential access by inserting the value RAB$C_SEQ in the RAB$B_RAC field.
- You specify either random access by key or random access by relative record number by inserting the value RAB$C_KEY in the RAB$B_RAC field. This access mode is used to randomly access records in indexed files using a specified key value. It is also used to randomly access records by record number in relative files and in sequential files having fixed-length records.

- You specify random access by RFA for all file organizations by inserting the value RAB$C_RFA in the RAB$B_RAC field.

Your program may also need to specify the key or other record identifier needed to access the records. For indexed files, there are additional key-related options.

The record access mode can be changed without reopening the file or reconnecting the record stream. For example, you can use random access by key to establish the current-record position in an indexed file and then retrieve records sequentially by a specified sort order. Note, however, that changing modes in this manner requires program access to the RAB$B_RAC control block field at run time.

The record access mode, in conjunction with the file organization, is what determines the manner in which a record is selected. In the following sections, the sequential and random access modes are discussed in the context of the applicable file organizations. Random access by RFA is discussed separately because it applies to disk files, regardless of file organization.

The following discussion of record-access modes is directed primarily toward services that insert records and services that retrieve records. For additional details about these services, see the *VMS Record Management Services Manual.*

8.4.1 Processing Sequential Files

A program can read sequential files on both tape and disk devices using the sequential record-access mode. If the file resides on disk, the random access by RFA mode can be used to read records; and if the file uses the fixed-length record format, the random access by relative record number mode is permitted.

You can add records only to the end of a sequential file.

All record access modes permit you to establish a new current-record position in a sequential file using the Find service. With sequential access, the Find service permits you to skip over records. With either random access by relative record number or random access by RFA, the Find service establishes a starting point for sequential Get services.

You cannot randomly delete records from a sequential file. However, you can randomly update records in a sequential file if the file is on disk and if the update does not change the record size.

The following sections discuss the use of sequential and random access modes with sequential files.

8.4.1.1 Sequential Access

The sequential access mode is supported for sequential files on all devices. It is the only record access mode that is supported for nondisk devices, such as terminals, mailboxes, and magnetic tapes.

With sequential access, VMS RMS returns records from sequential files in the order in which they were stored. When a program has retrieved all of the records from a sequential file, any further attempt to sequentially access records in the file causes VMS RMS to return an end-of-file (no more data) condition code.

In sequential access mode, you can add records *only* to the end of a sequential file, that is, the file location immediately following the current-record position.

8.4.1.2 Random Access

You can use the relative record number to randomly retrieve and insert records in sequential files having fixed-length records. Records are numbered in ascending order, starting with number 1.

In a sequential file, records are usually inserted at the end of the file. To insert records randomly within the current boundaries of the file at a relative record number less than or equal to the highest record number, set the update-if option (FDL attribute CONNECT UPDATE_IF; RAB$L_ROP bit RAB$V_UIF) to overwrite existing records.

When accessing a sequential file randomly by relative record number, your program must provide the record number at symbolic offset RAB$L_KBF and must specify a key length of 4 at symbolic offset RAB$B_KSZ, in the RAB.

8.4.2 Processing Relative Files

The relative file organization permits greater program flexibility in performing record operations than the sequential organization. A program can read existing records from the file using sequential, random access by relative record number mode, or random access by RFA mode. You can write new records either sequentially or randomly, as long as the intended record location (cell) does not already contain a record. You can also delete records.

All record access modes for relative files allow you to establish the current-record position using the Find or Get service. After finding the record, VMS RMS permits you to delete the record from the relative file. After deleting the record, the empty cell becomes available for a new record. In addition, your program can update records anywhere in the file. For variable-length records, the Update service can modify the record length up to the maximum size specified when the file was created.

When you insert a record into a relative file, the record is placed in a fixed cell within the file. A cell within a relative file can contain a record, can be vacant (never have contained a record), or can contain a deleted record.

The following sections discuss the sequential and random access modes for relative files.

8.4.2.1 Sequential Access

For relative files, the sequential access mode can be used to retrieve successive records in ascending record number. Vacant cells and cells that contain deleted records are skipped over automatically.

8.4.2.2 Random Access

You can directly read a record within a relative file by specifying the appropriate relative record number. If you attempt to read from a nonexistent cell—that is, a vacant cell or a cell containing a deleted record—VMS RMS returns an error message.

To position the record stream at a particular cell, regardless of whether or not it contains a record, use the nonexistent-record option (FDL attribute CONNECT NONEXISTENT_RECORD) or set the RAB$V_NXR bit in the RAB$L_ROP field.

You can use two key record-processing options to directly access records in relative files: the equal-or-next key option and the next-key option.

The equal-or-next-key option (FDL attribute CONNECT KEY_GREATER_EQUAL) directs VMS RMS to return a record having a record number equal to or greater than the specified record number. For example, when you specify record number 48, VMS RMS returns record number 48. If VMS RMS does not find record number 48, it returns the first record it encounters having a number greater than 48.

The next-key option (FDL attribute CONNECT KEY_GREATER_THAN) directs VMS RMS to return the record that has the next greater record number. For example, when you specify record number 48, VMS RMS returns record number 49, if record 49 exists.

You can also use random access mode to insert records into relative files. You can even overwrite cells that contain records by selecting the update-if option (FDL attribute CONNECT UPDATE_IF) or by directly setting the RAB$V_UIF bit in the RAB$L_ROP field.

To access a relative file randomly by record number, your program must specify the relative record number in the RAB at symbolic offset RAB$L_KBF and a key length value of 4 at symbolic offset RAB$B_KSZ.

8.4.3 Processing Indexed Files

Indexed files provide the most record-processing flexibility. Your program can read existing records from the file in sequential, random access by RFA, or random access by key modes. VMS RMS also allows you to write any number of new records into an indexed file if you do not violate a specified key characteristic, such as not allowing duplicate key values.

In random access by key mode, you can direct VMS RMS to use one of the key options in conjunction with one of four match options.

There are two key options:

- The equal-or-next-key option (FDL attribute CONNECT KEY_GREATER_EQUAL) returns a record with a key value that equals the specified key value. If VMS RMS cannot find a record with the specified key value, it returns a record with a key value that meets the requirements of the specified sort order.

For example, assume an indexed file has four records having keys G, K, R and V, respectively. If the program wants to retrieve a record with key M and has specified ascending sort order, VMS RMS returns the record with key value R. Conversely, with descending sort order specified in this situation, VMS RMS returns the record with key value K.

- The next-key option (FDL attribute CONNECT KEY_GREATER_THAN) returns a record having a key value that is not equal to the specified specified key value but meets the requirements of the specified sort order.

  For example, assume an indexed file has five records having keys G, K, M, R and V, respectively. If the program specifies the next-key option with key value M and ascending sort order, VMS RMS returns the record with key value R. Conversely, with descending sort order specified in this situation, VMS RMS returns the record with key value K. In both cases, VMS RMS does not return the record with key value M.

You can use the Find service (similar to the Get service), in sequential access mode, random access by RFA mode, or random access by key access mode. When finding records in random access by key access mode, your program can specify any one of the four types of key matches (exact, generic, approximate, generic/approximate) described in Sections 2.1.1.2 and 8.4.3.2.

In addition to reading, writing, and finding a record, your program can delete or update any record in an indexed file if the operation does not violate specified key characteristics. For example, if the program specifies that key values cannot be changed, any update that attempts to change a key value is rejected.

The next section describes how indexed files are used with the sequential and random access by key modes.

8.4.3.1 Sequential Access

You can use sequential record access mode to retrieve successive records in an indexed file. VMS RMS retrieves the records in successive order by the specified sort order for a key of reference. The key of reference (for example, primary key, first alternate key, second alternate key, and so forth) is established through one of the following services:

- The Connect service
- The Rewind service
- The Find service or the Get service using random access (Note that a Get or Put service specifying random access by RFA always establishes the key of reference as the primary key.)

When the sequential access mode is used with the Put service to insert records into an indexed file, successive records must be in the specified sort order by primary key.

8.4.3.2 Random Access

One of the most useful features of indexed files is that you can randomly retrieve records by the record's key value. A key value and a key of reference (such as a primary key, first alternate key, and so forth) can be specified as input to the record-processing service. VMS RMS searches the specified index to locate the record with the specified key value.

When reading records in random access by key mode, your program may specify one of four types of key matches:

- Exact key match
- Approximate key match
- Generic key match
- Approximate and generic key match

Exact match requires that the record's key value precisely match the key value specified by the program's Get service.

Approximate key match allows the program to select one of the following options:

- Equal-or-next-key option
- Next-key option

The advantage of using an approximate key match is that your program can retrieve a record without knowing its precise key value. VMS RMS uses the approximations in your program to return the record with the key value nearest the specified value.

If you elect to use a generic key match, your program need provide only a specified number of leading characters in the key, for example, the first 5 bytes (characters) of a 10-byte string data-type key. VMS RMS uses this information to return the first record with a key value that begins with these characters and meets the specified sorting order requirement. This is useful when attempting to locate a record when only part of the key is known or for applications in which a series of records must be retrieved when only the initial portions of their key values are identical. Generic key match is available for string keys only.

For example, if the program specifies the next-key option with a generic match on the three characters *RAM* using ascending sort order, VMS RMS returns records with key values *RAMA*, *RAMBO* and *RAMP* in that order. A record with key value *RAM* is not returned. If descending sort order is specified, VMS RMS returns records with key values *RAMP*, *RAMBO* and *RAMA* in that order.

When a generic key match is used with various approximate key match options, the results can vary, as shown in the following example. Consider using a key value of ABB to access records having key values of ABA, ABB and ABC, respectively.

- If the program elects to use the equal-or-next-key option with ascending sort order and a 3-character generic match, VMS RMS returns the record containing the key ABB.
- If the program uses the next-key option with ascending sort order and a 3-character generic match, VMS RMS returns the record with key value ABC.

- If the program uses the equal-or-next-key option with ascending sort order and a 2-character generic match, VMS RMS returns the record with key value ABA.

Now observe the effects of varying the key search option and the length of the generic string.

- If the program elects to use the equal-or-next-key option with ascending sort order and a 2-character generic match (AB), VMS RMS returns the record containing the key ABA.
- If the program uses the next-key option with descending sort order and a 3-character generic match, VMS RMS again returns the record with key value ABA.
- If the program uses the next-key option with descending sort order and a 2-character generic match (AB), VMS RMS returns a record-not-found condition because none of the records has a key that begins with the letters AA.

Now consider an example of how to return all the records in a file with key values that match the generic string AB.

1. Specify the generic string value of AB (2-byte key) in random access by key mode.
2. Use the Get service (or the Find services) to access the first record.
3. Change the record access mode to sequential.
4. Access the next record.
5. Compare the first two characters of the returned record's key with the first two characters of the specified key.
6. If the two key values are the same, process the record and return to step 4. If the two keys differ, do not process the record; instead, proceed to the next task (may require changing back to random access by key).

This procedure can be used to return all records that match a specified duplicate key for a key that allows duplicates. An alternative to checking the characters is to specify an ending key value and set the key-limit option when the record access mode is changed to sequential.

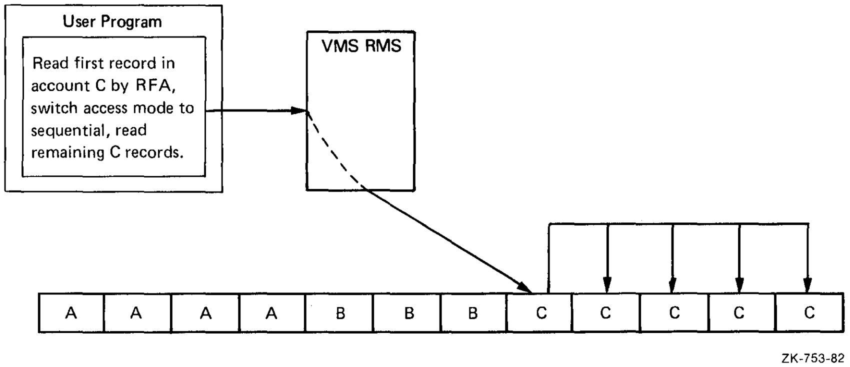
When accessing an indexed file randomly by key, the key value must reside in the area of memory identified by the control block offset RAB$L_KBF. When using string keys, you should specify the key length in the location identified by control block offset RAB$B_KSZ.

8.4.4 Access by Record File Address (RFA)

Random access by RFA is supported for all disk files. Whenever VMS RMS successfully accesses a record, an internal representation of the record's location is returned in the 6-byte RAB field RAB$W_RFA. When a program wants to retrieve the record using random access by RFA, VMS RMS uses this internal data to retrieve the record.

One way to use RFA access is to establish a record position for later sequential accesses. Consider a sequential file with variable-length records that can only be accessed randomly using RFA access. Assume the file consists of a list of transactions, sorted previously by account value. Because each account may have multiple transactions, each account value may have multiple records for it in the file. Instead of reading the entire file until it finds the first record for the desired account number, it uses a previously saved RFA value and random access by RFA to set the current-record position using a Find service at the first record of the desired account number. It can then switch to sequential record access and read all successive records for that account, until the account number changes or the end of the file is reached. Figure 8–1 shows how the file is accessed for account C.

Figure 8–1 Using RFA Access to Establish Record Position


User Program
Read first record in account C by RFA, switch access mode to sequential, read remaining C records.
VMS RMS
A
A
A
A
B
B
B
C
C
C
C
C
ZK-753-82

8.5 Block Input/Output

Block input/output (I/O) lets you bypass the VMS RMS record-processing capabilities entirely. In this manner, your program can process a file as a virtually contiguous set of blocks.

Block I/O operations provide an intermediate step between VMS RMS operations and direct use of the Queue I/O Request system service. Using block I/O gives your program full control of the data in the individual blocks of a file while being able to take advantage of the VMS RMS capabilities for opening, closing, and extending a file.

In block I/O, a program reads or writes one or more blocks by specifying a starting virtual block number in the file and the length of the transfer. Regardless of the organization of the file, VMS RMS accesses the identified block or blocks.

Because VMS RMS files contain internal information meaningful only to VMS RMS itself, DIGITAL does not recommend that you modify an existing file using block I/O if the file is also to be accessed by VMS RMS record-level operations. (Block I/O does not update any internal record information.) The block I/O facility, however, does allow you to create your own file organizations. This file structure must be maintained through specialized user-written programs and procedures; VMS RMS cannot access these structures with its record access modes.

For more information about using block I/O, see the *VMS Record Management Services Manual.*

8.6 Current Record Context

For each RAB connected to a FAB, VMS RMS maintains current context information about the record stream including the current-record position and the next-record position. Furthermore, the current context is different for the various VMS RMS services as shown in Table 8–2.

The current record context is internal to VMS RMS; you have no direct contact with it. However, you should know the context for each service in order to properly access records when you invoke a service.

Table 8–2 Record Access Stream Context

Service	Access Mode	Current	Next
Connect	N/A	None	First record
Connect with RAB$L_ROP RAB$V_EOF bit set	N/A	None	End of file
Get, when last service was not a Find	Sequential	Old next record	New current record+1
Get, when last service was a Find	Sequential	Unchanged	Current record+1
Get	Random	New	New current record+1
Put, sequential file	Sequential	None	End of file
Put, relative file	Sequential	None	Next record position
Put, indexed file	Sequential	None	Undefined
Put	Random	None	Unchanged
Find	Sequential	Old next record	New current record+1
Find	Random	New	Unchanged
Update	N/A	None	Unchanged
Delete	N/A	None	Unchanged
Truncate	N/A	None	End of file
Rewind	N/A	Unchanged	First record
Free	N/A	None	Unchanged
Release	N/A	None	Unchanged

Notes to Table 8–2:

1 Except for the Truncate service, VMS RMS establishes the current-record position before establishing the next-record position.

2 The notation "+1" indicates the next sequential record as determined by the file organization. For indexed files, the current key of reference is part of this determination.

3 The Connect service on an indexed file establishes the next record to be the first record in the index represented by the RAB key of reference (RAB$B_KRF) field.

4 The Connect service leaves the next record as the end of file for a magnetic tape file opened for Put services (unless the FAB$V_NEF option in the FAB$L_FOP is set).

8.6.1 Current-Record Position

For the Update, Delete, Release, and Truncate services, the current-record position reflects the location of the target record. The current-record position also facilitates sequential processing on disk devices for a stream.

The following list describes situations where the current-record position is undefined:

- When a RAB is first connected to a FAB
- When a record operation is unsuccessful
- Following the successful execution of a VMS RMS service other than a Get service or Find service.

When the current-record position is undefined, VMS RMS rejects the Update, Delete, Release, or Truncate services.

A Get service using sequential record access mode and immediately preceded by the Find service operates on the record specified by the current-record position. If the Find service does not lock the record (for relative and indexed files) and the current record is deleted, the Get service accesses the record at the next-record position.

Following successful execution of the Get service or the Find service, the current-record position is set to the target record's RFA. VMS RMS also places the target record's address in the RFA field of the related RAB. The results are as follows:

- After initialization, the current-record position reflects the RFA of the record that was the object of the most recent successful Get service or Find service (unless a failure occurs on a different VMS RMS service).
- Unless it is modified, the RAB$W_RFA field always contains the address of the target current record. (If the operation fails, the RFA is undefined.)

Table 8–2 summarizes the effect that each successful record operation has on the context of the current record.

8.6.2 Next-Record Position

VMS RMS uses the next-record position for doing sequential record access. For sequential record processing, the next-record position is the location of the target record for the next Find service (Get service where appropriate) or Put service. In a relative file, the target record is the record that occupies the next non-vacant cell.

The ability to look ahead significantly decreases access time for sequential processing. VMS RMS uses its internal knowledge of file organization and structures to determine the next-record position for each record service.

The Connect service initializes the next-record position to one of the following locations:

- The first record in a sequential file, or the first cell in a relative file.
- The first record in the collated sequence of the specified key of reference in an indexed file
- The end of a file on disk, if the RAB$L_ROP field RAB$V_EOF option is set
- The end of a write-accessed ANSI magnetic tape file, unless the FAB$V_NEF option is set in the FAB$L_FOP field

In any record access mode, the Get service establishes the next-record position as either the next record or the next record cell in the file. This is also true for the Find service in sequential access mode.

The Truncate service establishes the end of the file at the current-record position (effectively deleting the record at that location and all records following it) so you need only use Put services to extend the file. Note that you can only truncate sequential files.

In random access mode, the Find (or Get) service and the Put service do not affect the next-record position, unless these services are used to add a record with a primary key value or a record number that lies between the corresponding values of the current record and the next record. When this occurs, the current-record position is changed to reflect the location of the added record; that is, records are added *after the current record,* not before the next record.

In sequential access mode, the Put service initializes the next-record position to the end of the file in a sequential file. In a relative file, the Put service initializes the next-record position to the next record or record cell. For sequential accesses to an indexed file, the Put service does not define the next-record position.

Regardless of access mode, the Delete, Update, Free, and Release services have no effect on the next-record position. For sequential and relative files, the Rewind service establishes the next-record position as the first record or record cell in the file, regardless of the access mode. For indexed files, the Rewind service always establishes the next-record position as the location of the first record for the current key of reference.

Any unsuccessful record operation has no effect on the next record.

8.7 Synchronous and Asynchronous Operations

Your program can handle record operations on a file in one of two ways: synchronously or asynchronously. When operating synchronously, the program issuing the record-operation request regains control only when the request is completely satisfied. Most high-level languages support synchronous operation only. In asynchronous operations, the program can regain control before the request is completely satisfied. You can specify record operations and file operations to be either synchronous or asynchronous for each record stream.

For instance, when reading a record from a file synchronously, the program regains control only after the record is passed to the program. In other words, the program waits until the record returns; no other processing for this program takes place during this read-and-return cycle. On the other hand, when reading a record asynchronously, the program might be able to regain control before the record is passed to the program. The program can thus use the time normally required for the record transfer between the file and memory to perform some other computations. Another record operation cannot be started on the same stream until the previous record operation is complete. However, record operations on other streams can be initiated.

Whether the program regains control before the record operation finishes depends on several factors. For example, the required record may already reside in the I/O buffer, or the operating system may schedule another process, thus possibly allowing a necessary I/O operation to be completed before the original program is rescheduled.

One factor to consider in the use of asynchronous record operations is that you must include a separate completion routine or a VMS RMS wait request in the issuing program. This routine (or wait request) is required to determine when the record operation is completed because the results of the operation are not available, and the next record operation for that stream cannot be initiated until the previous operation is concluded.

8.7.1 Using Synchronous Operations

To declare a synchronous operation, you must clear the RAB$V_ASY option in the RAB$L_ROP field. Normally, you do not have to clear this option because it is already cleared (by default). However, if the RAB$V_ASY option had been set previously, then you must explicitly clear it.

Normally, you do not use success and error routines with synchronous operations. Instead, you test the completion status code for an error and change the flow of the program accordingly. However, if you use these routines, they are executed as asynchronous system traps (ASTs) before the VMS RMS service returns to your program (unless ASTs are disabled).

User-mode AST routines may be executed before the completion of a synchronous record operation (see the *VMS Record Management Services Manual*). If an AST routine attempts to perform operations on a record stream that is being called from a non-AST level, it must be prepared to handle stream-activity errors (RMS$_RSA or RMS$_BUSY).

8.7.2 Using Asynchronous Operations

To declare an asynchronous record operation, you must set the asynchronous (RAB$V_ASY) option in the RAB$L_ROP field. You can switch between synchronous and asynchronous operations during processing of a record stream by setting or clearing the RAB$V_ASY option on a per-operation basis.

You can specify completion routines to be executed as ASTs if success or error conditions occur. Within such routines, you can issue additional operations, but they should also be asynchronous. If they are not, all other asynchronous requests currently active in your program cannot have their completion routines executed until the synchronous operation completes.

If an asynchronous operation is not completed at the time of return from a call to a VMS RMS service, the completion status field of the RAB is 0, and a success status code of RMS$_PENDING is returned in Register 0. This status code indicates that the operation was initiated but is not yet complete.

Note: **Never modify the contents of a VMS RMS control block when an operation is in progress because the results are unpredictable.**

If you issue a second record operation request for the same stream before a previous request is completed, you receive an RMS$_RSA or RMS$_BUSY error status code, indicating that the record stream is still active. This can also occur when an AST-level routine attempts to use an active record stream; the original I/O request may be synchronous or asynchronous. An additional error (RMS$_BUSY) can be encountered by attempting an operation using the same record stream (RAB) from an error or success routine, when the main program is awaiting completion of the initial operation. In all cases, it is your responsibility to recognize this possibility and prevent the problem. Most problems can be prevented by using a Wait service. When the Wait service concludes, it returns control to your program.

Note that the Connect operation may be performed asynchronously. If the RAB$V_ASY option is set, a Wait service should follow the Connect service to synchronize with the completion of the Connect service. Another technique is to use the Connect service synchronously and set the RAB$V_ASY option at run time, after the Connect service.

9 Run-Time Options

This chapter describes the way you specify run-time options and it summarizes the run-time options available to you when opening files, connecting record streams, processing records, and closing files. The run-time options that apply to record processing and to opening and closing a file can usually be preset by file-open and record stream connection values. Some options can be selected after you open a file and connect a record stream.

Note that run-time options discussed in previous sections are only summarized in this chapter. Most of the material in this chapter relates to options not previously described in this document.

9.1 Specifying Run-Time Options

This section describes the way you use the FDL Editor to specify run-time options that are available to your program through the FDL$PARSE and FDL$RELEASE routines. It also describes the use of language statements and VMS RMS to specify control block values.

You select VMS RMS options by setting appropriate values in VMS RMS control blocks within the data portion of your program. In many cases, you can select these values by using keywords available to you in the VAX language OPEN statement for your application, or by taking suitable default values. The values may be selected using keywords in your record and file description statements or they may be selected directly within the OPEN statement.

If your application is written in a VAX language that does not provide keywords for the various features, you can usually select the options using the File Definition Language (FDL).

Predefined FDL attributes can be supplied to your program at run time using the FDL$PARSE routine. This routine also returns the address of the record access block (RAB) to let your program subsequently change RAB values. Some RAB options are not available in FDL and can be set only by directly accessing RAB fields and subfields at run time. To invoke options after record stream connection, your program must have direct access to VMS RMS control block fields using the address of the RAB and symbolic offsets into it.

9.1.1 Using the FDL Editor

You can use the FDL Editor to specify run-time attributes, such as adding a CONNECT attribute that is used to set a control block value when the FDL$PARSE and FDL$RELEASE routines are called by your program. These attributes preset the values available for opening a file and connecting a record stream. The following illustration is an original FDL file created with the FDL Editor:

```
IDENT   "19-JUL-1984 14:57:37   VAX-11 FDL Editor"

SYSTEM
        SOURCE                  VMS

FILE
        ORGANIZATION            indexed

RECORD
        CARRIAGE_CONTROL        carriage_return
        FORMAT                  variable
        SIZE                    0

AREA 0
        ALLOCATION              8283
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             18
        EXTENSION               2070

AREA 1
        ALLOCATION              18
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             18
        EXTENSION               18

KEY 0
        CHANGES                 no
        DATA_AREA               0
        DATA_FILL               100
        DATA_KEY_COMPRESSION    yes
        DATA_RECORD_COMPRESSION yes
        DUPLICATES              no
        INDEX_AREA              1
        INDEX_COMPRESSION       yes
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       1
        PROLOG                  3
        SEG0_LENGTH             9
        SEG0_POSITION           0
        TYPE                    string
```

Because the FDL Editor does not include run-time attributes, you must add them to the FDL definition. You can specify run-time attributes by specifying the ACCESS, CONNECT and SHARING attributes. For example, if you want to add the CONNECT secondary attribute LOCK_ON_WRITE, you use the EDIT/FDL ADD command. This is illustrated in Example 9–1.

Example 9–1 Specifying Run-Time Attributes

```
                      VAX-11 FDL Editor

  Add     to insert one or more lines into the FDL definition
  Delete  to remove one or more lines from the FDL definition
  Exit    to leave the FDL Editor after creating the FDL file
  Help    to obtain information about the FDL Editor

❶ Invoke  to initiate a script of related questions
  Modify  to change existing line(s) in the FDL definition
  Quit    to abort the FDL Editor with no FDL file creation
  Set     to specify FDL Editor characteristics
  View    to display the current FDL Definition
❷ Main Editor Function           (Keyword)[Help] : ADD

                      Legal Primary Attributes

  ACCESS  attributes set the run-time access mode of the file
  AREA x  attributes define the characteristics of file area x
  CONNECT attributes set various VMS RMS run-time options
  DATE    attributes set the data parameters of the file
  FILE    attributes affect the entire VMS RMS data file
❸ JOURNAL attributes set the journaling parameters of the file
  KEY y   attributes define the characteristics of key y
  RECORD  attributes set the non-key aspects of each record
  SHARING attributes set the run-time sharing mode of the file
  SYSTEM  attributes document operating system-specific items
  TITLE   is the header line for the FDL file
❹ Enter Desired Primary          (Keyword)[FILE] : CONNECT

                      Legal CONNECT Secondary Attributes

  ASYNCHRONOUS            yes/no  NOLOCK              yes/no
  BLOCK_IO                yes/no  NONEXISTENT_RECORD  yes/no
  BUCKET_CODE             number  READ_AHEAD          yes/no
  CONTEXT                 number  READ_REGARDLESS     yes/no
  END_OF_FILE             yes/no  TIMEOUT_ENABLE      yes/no
  FAST_DELETE             yes/no  TIMEOUT_PERIOD      number
  FILL_BUCKETS            yes/no  TRUNCATE_ON_PUT     yes/no
  KEY_GREATER_EQUAL       yes/no  TT_CANCEL_CONTROL_O yes/no
❺ KEY_GREATER_THAN        yes/no  TT_PROMPT           yes/no
  KEY_LIMIT               yes/no  TT_PURGE_TYPE_AHEAD yes/no
  KEY_OF_REFERENCE        number  TT_READ_NOECHO      yes/no
  LOCATE_MODE             yes/no  TT_READ_NOFILTER    yes/no
  LOCK_ON_READ            yes/no  TT_UPCASE_INPUT     yes/no
  LOCK_ON_WRITE           yes/no  UPDATE_IF           yes/no
  MANUAL_UNLOCKING        yes/no  WAIT_FOR_RECORD     yes/no
  MULTIBLOCK_COUNT        number  WRITE_BEHIND        yes/no
  MULTIBUFFER_COUNT       number
```

Example 9–1 Cont'd. on next page

Example 9–1 (Cont.) Specifying Run-Time Attributes

```
❻ Enter CONNECT Attribute        (Keyword)[-]    : LOCK_ON_WRITE

❼ CONNECT
          LOCK_ON_WRITE

❽ Enter value for this Secondary   (Yes/No)[-]    : YES

                  Resulting Primary Section

❾ CONNECT
           LOCK_ON_WRITE           yes

❿ Press RETURN to continue (^Z for Main Menu)
```

❶ This menu is the Main Editor Function menu. It displays the EDIT/FDL commands you can use.

❷ The ADD command displays the Legal Primary Attributes menu.

❸ The Legal Primary Attributes menu shows the primary attributes. You can either add a new primary attribute or add a secondary attribute to an existing primary attribute. Initially, the FILE primary attribute is the default.

❹ The selection of the CONNECT primary attribute displays the Legal CONNECT Secondary Attributes. You could similarly select the ACCESS, FILE, or SHARING options instead of the CONNECT primary attribute to display the Legal Secondary Attributes for the selected primary attribute.

❺ This menu shows all the CONNECT secondary attributes you can add to your FDL file.

❻ Select the proper CONNECT secondary attribute (in this case, LOCK_ON_WRITE).

❼ EDIT/FDL verifies that you have selected the secondary attribute.

❽ Enter the value that you want the secondary attribute to have (for instance, *yes*).

❾ EDIT/FDL verifies the value for the secondary attribute you have chosen.

❿ Return to the main menu. If you choose to add another secondary attribute, you will notice that CONNECT is now the default.

The FDL file containing the CONNECT primary attribute with the WRITE_BEHIND secondary attribute is shown in the following example:

```
IDENT    "19-JUL-1984 14:57:37   VAX-11 FDL Editor"

SYSTEM
          SOURCE                VMS

FILE
         ORGANIZATION           indexed

RECORD
         CARRIAGE_CONTROL       carriage_return
         FORMAT                 variable
         SIZE                   0
```

```
CONNECT
        WRITE_BEHIND             yes

AREA 0
        ALLOCATION               8283
        BEST_TRY_CONTIGUOUS      yes
        BUCKET_SIZE              18
        EXTENSION                2070

AREA 1
        ALLOCATION               18
        BEST_TRY_CONTIGUOUS      yes
        BUCKET_SIZE              18
        EXTENSION                18

KEY 0
        CHANGES                  no
        DATA_AREA                0
        DATA_FILL                100
        DATA_KEY_COMPRESSION     yes
        DATA_RECORD_COMPRESSION  yes
        DUPLICATES               no
        INDEX_AREA               1
        INDEX_COMPRESSION        yes
        INDEX_FILL               100
        LEVEL1_INDEX_AREA        1
        PROLOG                   3
        SEG0_LENGTH              9
        SEG0_POSITION            0
        TYPE                     string
```

9.1.2 Using Language Statements and VMS RMS

Language statements such as OPEN may contain keywords, clauses, or other modifiers that correspond to the run-time attributes that are appropriate for opening files, connecting record streams, processing records, and closing files. Some languages use system-defined procedures in place of keywords and clauses. Some languages allow you to call a user-supplied routine (USEROPEN or USERACTION) to set control block values before opening the file. For example, a user routine could be coded in VAX MACRO to take advantage of control block store macros (for an example of a VAX BASIC USEROPEN routine, see the Example 5–2). Consult the corresponding language documentation for additional information.

With VAX MACRO, VMS RMS control block macros allow you to establish control block values at assembly time and at run time using the same control block. (The assembly-time macros are placed in a data section of the program; the run-time macros are placed in a code section of the program.) Using VAX MACRO, control blocks are allocated within the program space at assembly time, and it may not be necessary to use the run-time macros because the program can move values to the control block fields using the VAX instruction set. Other languages, however, may not allocate the control blocks within program storage.

If your program has access to the starting location of the control block (a record access block, for instance), the VAX MACRO assembly-time control block macro or the corresponding symbol definition (DEF) macro provides your program with certain symbolic offsets (symbols) that can be used to locate and identify the various fields in the control block. Some VAX languages provide a means of making these symbols available to your program.

For additional information about using the control block macros and control block fields, refer to the *VMS Record Management Services Manual.*

9.2 Options Related to Opening and Closing Files

Before your program can access the records in a file, it must open the file and connect a record stream. When it finishes processing records and no longer requires access to that file, your program should close the file.

The options available for opening files, connecting record streams, and closing files include file access and file sharing options, file specification options, performance options, record access options, and options for:

- Adding records
- Acting on the file after it is closed (file disposition)
- Using indexed files
- Using magnetic tapes
- Performing nonstandard record processing
- Maintaining data reliability

9.2.1 File Access and Sharing Options

As described in Chapter 7, the program must declare the desired file-access and file-sharing values before opening an existing file or creating a new file and must specify record-locking and buffering strategies when the file is opened. These options are summarized in the following chart.

Option	Description
File access	Specifies the record operations that the current process performs: reading records, locating records, deleting records, adding new records, updating records, accessing blocks, and truncating the file. (For additional information, see Section 7.1.) You specify the file access values using the FDL ACCESS primary attribute or the VMS RMS FAB$B_FAC field.
File sharing	Specifies the types of record operations that the current process allows other file accessors to perform: reading records, locating records, deleting records, adding new records, and updating records. You can also use file sharing to enable the current process to use multiple record streams (or ensure a read-only global buffer cache), operate on the file without record interlocking, or disallow all other accessors from accessing the file. You specify file sharing values using the SHARING primary attribute or the VMS RMS FAB$B_SHR field.

Option	Description
Record locking	Allows you to provide record locking for a shared file under user control. By default, VMS RMS automatically locks records, depending on the file access and file sharing values specified. (For additional information, see Section 7.2.) You specify the record locking values using the CONNECT primary attribute or using the VMS RMS record-processing options (RAB$L_ROP) field [1].

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.2 File Specifications

As described in Chapters 4 and 6, the program should specify the specification for the file being opened (or created) and can also specify default file specifications. The file specifications are summarized in the following table:

File Specification	Description
Primary	Specifies the file specification to be used to locate the desired file(s). If any components of a file specification are omitted, VMS RMS applies defaults but you should specify the primary file specification. FDL: FILE NAME VMS RMS: FAB$L_FNA and FAB$B_FNS
Default	Specifies the default file specification to be used to fill any missing components not provided by the primary file specification. After applying these defaults, if any components are still missing, additional defaults are applied. FDL: FILE DEFAULT_NAME VMS RMS: FAB$L_DNA and FAB$B_DNS
Related	Specifies a related file specification that is used to provide additional defaults when a related file is used. If the device or directory components are missing, VMS RMS provides default values from the process-default device (SYS$DISK) and the current process-default directory. FDL: None. VMS RMS: FAB$L_NAM and NAM$L_RLF

9.2.3 File Performance Options

A number of run-time options that open files and connect record streams can collectively improve application performance. Such options include the buffering options discussed in Chapter 7.

Two run-time performance options not discussed previously are particularly important when adding records to a file: extension size and window size.

9.2.3.1 Extension Size

If you intend to add records to the file, specify a reasonable default extension size to reduce the number of times the file is extended.

Use the Edit/FDL Utility to calculate the correct extension size. EDIT/FDL uses your responses to assign an optimum value for the FDL attribute FILE EXTENSION. With multiple area files, EDIT/FDL assigns optimum values to the AREA EXTENSION attributes.

If you do not specify an extension size, VMS RMS computes the size; however, this size may not be optimum.

If you decide to create an FDL file for defining an indexed file without using EDIT/FDL, you can approximate the value of the EXTENSION attributes. You do this by multiplying number of records per bucket by the number of records that you intend to add to the file during a given period of time.

To see the current default extension size, use the DCL command SHOW RMS_DEFAULT. To set the default buffer count, use the DCL command SET RMS_DEFAULT/EXTEND_QUANTITY=n, where *n* is the number of blocks per extension. The corresponding VMS RMS field is FAB$B_DEQ.

9.2.3.2 Window Size

If the file is extended repeatedly, the extensions may be scattered on the disk. Each extension is called an *extent*—a pointer to each extent resides in the file header. For retrieval purposes, the pointers are gathered together in a structure called a *window*. The default window size is 7 pointers, but you can establish the window size to contain as many as 127 pointers. You can also set the window size to –1, which makes a window that is just large enough to map the entire file.

When you access an extent whose pointer is not in the current window, the system has to read the file header and fetch the appropriate window. This is called a *window turn*, and it requires an I/O operation.

Window size is a run-time option. Many high-level languages include a clause that sets window size when a file is opened. You can set the window size (FAB$B_RTV field) at run time with a VAX MACRO subroutine or with the FDL attribute FILE WINDOW_SIZE.

You can increase the default window size for a specific disk volume by using the DCL commands MOUNT and INITIALIZE. However, using additional window pointers increases system overhead. The window size is charged to your buffered I/O byte count quota, and indiscriminate use of large windows may result in exceeding the buffered I/O byte count quota or may exhaust the system's nonpaged dynamic memory.

You can use the Backup Utility (BACKUP) to avoid having too many extents. When you restore a file, BACKUP tries to write the file in one section of the disk. Although BACKUP does not necessarily create a contiguous copy of the file, it does reduce the number of extents. If you are regularly backing up the file, the number of extents is probably reasonable. For more information about BACKUP, see the *VMS Backup Utility Manual*.

Where disk space is available, you can reduce the number of extents by creating a new, contiguous version of the file using either the Convert Utility (CONVERT) or the DCL command COPY/CONTIGUOUS. If neither of these conditions apply, a larger window size is the only option to use. For file maintenance information, see Chapter 10.

9.2.3.3 Summary of Performance Options

The following list summarizes the run-time open and connect options that may affect performance.

Option	Description
Asynchronous record processing[1]	Specifies that record I/O for this record stream is done asynchronously. See Section 8.7. FDL: CONNECT ASYNCHRONOUS VMS RMS: RAB$L_ROP RAB$V_ASY
Deferred-write[1]	Allows records to be accumulated in a buffer and written only when the buffer is needed or when the file is closed. For use by all except non-shared sequential files. See Chapter 3. FDL: FILE DEFERRED_WRITE VMS RMS: FAB$L_FOP FAB$V_DFW
Default extension quantity	Specifies the number of blocks to be allocated to a file when more space is needed. FDL: FILE EXTENSION VMS RMS: FAB$W_DEQ
Fast delete[1]	Postpones certain internal operations associated with deleting indexed file records until the record is accessed again. This allows records to be deleted rapidly but may affect the performance of subsequent accessors reading the file. FDL: CONNECT FAST_DELETE VMS RMS: RAB$L_ROP RAB$V_FDL
Global buffer count	Specifies whether global buffers are used and the number to be used if the record stream is the first to connect to the file. See Section 7.3. FDL: CONNECT GLOBAL_BUFFER_COUNT VMS RMS: FAB$W_GBC
Locate mode[1]	Allows the use of locate mode, not move mode, when reading records. See Section 7.3. FDL: CONNECT LOCATE_MODE VMS RMS: RAB$L_ROP RAB$V_LOC
Multiblock count	Allows multiple blocks to be transferred into memory during a single I/O operation (for sequential files only). See Chapter 3 and Section 7.3. FDL: CONNECT MULTIBLOCK_COUNT VMS RMS: RAB$B_MBC
Number of buffers	Enables the use of multiple buffers for the buffer cache when used with indexed and relative files; when used with sequential files, enables the use of multiple buffers for the read-ahead and write-behind options. See Section 7.3. FDL: CONNECT MULTIBUFFER_COUNT VMS RMS: RAB$B_MBF

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
Read-ahead[1]	Alternates buffer use between two buffers when reading sequential files. See Chapter 2. FDL: CONNECT READ_AHEAD VMS RMS: RAB$L_ROP RAB$V_RAH
Retrieval window size	Specifies the number of entries in memory for retrieval windows, which corresponds to the number of extents for a file. FDL: FILE WINDOW_SIZE VMS RMS: FAB$B_RTV
Sequential access only	Indicates that a sequential file may only be accessed sequentially. FDL: FILE SEQUENTIAL_ONLY VMS RMS: FAB$L_FOP FAB$V_SQO
Write-behind[1]	Alternates buffer use between two buffers when writing to sequential files See Chapter 2. FDL: CONNECT WRITE_BEHIND VMS RMS: RAB$L_ROP RAB$V_WBH

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.4 Record Access Options

You can specify the record access for a record stream as sequential, random by key or record number, or random by RFA. (See Section 8.1.) The selected record access can be changed for each record processing operation. These options can be set using the VMS RMS RAB$B_RAC field, values RAB$C_SEQ, RAB$C_KEY, and RAB$C_RFA.

9.2.5 Options for Adding Records

When adding records to a file, consider the open and connection options in the following list:

Option	Description
Default extension quantity [1]	See Section 9.2.3
Deferred-write[1]	See Section 9.2.3
End-of-file	After the record stream is connected, the record context is positioned to the end of the file. FDL: CONNECT END_OF_FILE VMS RMS: RAB$L_ROP RAB$V_EOF
Retrieval window size [1]	See Section 9.2.3

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
Revision data	The revision date and time and the revision number can be specified to be a value other than the actual revision date and time and revision number when the file is closed. These options must be set while the file is open and thus cannot be set using FDL. FDL: Does not apply. VMS RMS: Revision Date and Time XAB
Truncate on Put[1]	When using sequential record access for sequential files only, the record to be written is the last record in the file, and VMS RMS truncates the file just beyond that record. FDL: CONNECT TRUNCATE_ON_PUT VMS RMS: RAB$L_ROP RAB$V_TPT
Update-if[1]	If you set this option and your program tries to replace an existing record while adding records randomly to a file, VAX RMS modifies the existing record instead of replacing it. When using this option for indexed files, note that the file must *not* allow duplicates for the primary key. Use this option carefully with a shared file (see Section 8.1). FDL: CONNECT UPDATE_IF VMS RMS: RAB$L_ROP RAB$V_UIF
Write-behind[1]	See Section 9.2.3

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.6 Options for Data Reliability

The following table lists the run-time *file open* options that apply to data reliability.

Option	Description
Read-check	Specifies that transfers from disk volumes are to be checked by a read-compare operation, which effectively doubles the amount of disk I/O performed. This option is not available for all devices (see the *VMS Record Management Services Manual*). FDL: FILE READ_CHECK VMS RMS: FAB$L_FOP FAB$V_RCK
Write-check	Specifies that transfers to disk volumes are to be checked by a read-compare operation, which effectively doubles the amount of disk I/O performed. This option is not available for all devices (see the *VMS Record Management Services Manual*). FDL: FILE WRITE_CHECK VMS RMS: FAB$L_FOP FAB$V_WCK

9.2.7 Options for File Disposition

The run-time file open options that apply to file disposition are listed in the following table. These options can only be selected while the file is open.

Option	Description
Delete on close	Deletes the file when it is closed. FDL: CONNECT DELETE_ON_CLOSE VMS RMS: FAB$L_FOP FAB$V_DLT
Submit command file	Submits a sequential file as a batch command procedure to SYS$BATCH when you close the file. FDL: FILE SUBMIT_ON_CLOSE VMS RMS: FAB$L_FOP FAB$V_SCF
Spool on close	Prints a sequential file on SYS$PRINT you close the file. FDL: FILE PRINT_ON_CLOSE VMS RMS: FAB$L_FOP FAB$V_SPL

9.2.8 Options for Indexed Files

The following table lists the run-time options that apply to indexed file processing For more information about processing indexed files, refer to Section 8.4.3.

Option	Description
Fast delete[1]	This option lets you postpone certain internal operations associated with deleting indexed file records until the record is next accessed. This allows records to be deleted rapidly, but it may degrade the performance of processes that read the file later. FDL: CONNECT FAST_DELETE VMS RMS: RAB$L_ROP RAB$V_FDL
Key equal or next[1]	If you select this option when locating or reading records, VMS RMS returns the first record with a key value equal to the specified key. If VMS RMS does not find a record with an equal key value, it returns the record with the next higher key value when ascending sort order is specified. When descending sort order is specified, VMS RMS returns the next record with the next lower key value. FDL: CONNECT KEY_GREATER_EQUAL VMS RMS: RAB$L_ROP RAB$V_EQNXT

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
Next key[1]	If you select this option when locating or reading records, VMS RMS returns the record with the next higher key value when you specify ascending sort order. When you specify descending sort order, VAX RMS returns the next record with the next lower key value. If you do not specify either this option or the equal-or-next-key option, VMS RMS tries for a key match. FDL: CONNECT KEY_GREATER_THAN VMS RMS: RAB$L_ROP RAB$V_NXT
Key of reference	When you process an indexed file with multiple keys, this option permits you specify which key to use for the current record stream. FDL: CONNECT KEY_OF_REFERENCE VMS RMS: RAB$B_KRF
Key buffer[1]	If you select this option when locating or reading records randomly, the specified key buffer must contain the selected record's key. FDL: None. VMS RMS: RAB$L_KBF
Key size[1]	If you select this option when locating or reading records with a string key type, you can specify that only a portion of the key be used to locate the selected record. FDL: None. VMS RMS: RAB$B_KSZ
Limit key[1]	This option directs VMS RMS, when locating or reading records sequentially, to return an alternate success status if the record key exceeds the specified key. FDL: CONNECT KEY_LIMIT VMS RMS: RAB$L_ROP RAB$V_LIM
Load buckets[1]	If you select this option when adding records to an index file, VMS RMS uses the fill factor specified when the file was created. By default, VMS RMS fills buckets completely. FDL: CONNECT FILL_BUCKETS VMS RMS: RAB$L_ROP RAB$V_LOA

[1] Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.9 Options for Magnetic Tape Processing

The run-time file open and close options that apply to magnetic tape processing are listed in the following table:

Option	Description
Not end-of-file	Use this option when you want to add a record to a location other than at the end of the file. FDL: FILE MT_NOT_EOF VMS RMS: FAB$L_FOP FAB$V_NEF
Current position	If you select this option when creating a file, the tape is positioned to the location immediately following the most recently closed file. FDL: FILE MT_CURRENT_POSITION VMS RMS: FAB$L_FOP FAB$V_POS
Rewind on Open	If you select this option, the VMS RMS directs that the tape volume be rewound before it opens or creates the file. The rewind-on-open option overrides the current-position option. FDL: FILE MT_OPEN_REWIND VMS RMS: FAB$L_FOP FAB$V_RWO
Rewind on Close	If you select this option, VMS RMS directs that the tape volume be rewound before it closes the file. FDL: FILE MT_CLOSE_REWIND VMS RMS: FAB$L_FOP FAB$V_RWC

9.2.10 Options for Nonstandard File Processing

The following table lists the run-time file open options that apply to nonstandard file processing.

Option	Description
Non-file-structured	Use this option when you want to process data from volumes created on non-DIGITAL systems. FDL: FILE NON_FILE_STRUCTURED VMS RMS: FAB$L_FOP FAB$V_NFS
User file open	Use this option if you want to use VMS RMS only to open the file and you intend to access the contents of the file using Queue I/O Request system service calls. The system returns the I/O channel number in the FAB$L_STV field. FDL: FILE USER_FILE_OPEN VMS RMS: FAB$L_FOP FAB$V_UFO

9.3 Summary of Record Operation Options

This section briefly describes the options associated with the record retrieval services (Find and Get), the record insertion service (Put), the record modification service (Update), and the record deletion service (Delete).

9.3.1 Record Retrieval Options

The Find and Get services (or the equivalent VAX language statements) can be used to locate and retrieve a record.

The options associated with the Find and Get services are summarized in the following table. These options can be set for *each* Find or Get service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VMS RMS service calls.

Option	Description
Asynchronous record processing	Specifies that record I/O for this record stream is done asynchronously. FDL: CONNECT ASYNCHRONOUS VMS RMS: RAB$L_ROP RAB$V_ASY
Do not lock record	Directs VMS RMS not to lock the record for ensuing operations. FDL: CONNECT NOLOCK VMS RMS: RAB$L_ROP RAB$V_NLK
Key buffer	When locating/reading records randomly, the specified key buffer must contain the desired record's key. FDL: None. VMS RMS: RAB$L_KBF
Key equal or next	When locating or reading records, VMS RMS returns the first record with a key value equal to the specified key. If VMS RMS does not find a record with an equal key value, it returns the record with the next higher key value when you specify ascending sort order. When you specify descending sort order, VMS RMS returns the record with the next lower key value. FDL: CONNECT KEY_GREATER_EQUAL VMS RMS: RAB$L_ROP RAB$V_EQNXT
Next key	When locating or reading records, VMS RMS returns the record with the next higher key value when you specify ascending sort order. When you specify descending sort order, VMS RMS returns the record with the next lower key value. FDL: CONNECT KEY_GREATER_THAN VMS RMS: RAB$L_ROP RAB$V_NXT
Key of reference	For indexed files with multiple keys, the key of reference specifies which key is used for current record stream. FDL: CONNECT KEY_OF_REFERENCE VMS RMS: RAB$B_KRF
Key size	When using a string key to locate or read records, you can specify that all or part of the key be used. FDL: None. VMS RMS: RAB$B_KSZ

Option	Description
Limit key	This option directs VMS RMS, when locating or reading records sequentially, to return an alternate success status if the record key exceeds the specified key. FDL: CONNECT KEY_LIMIT VMS RMS: RAB$L_ROP RAB$V_LIM
Locate mode	Specifies the locate mode, instead of the move mode. Applies to the Get service only. FDL: CONNECT LOCATE_MODE VMS RMS: RAB$L_ROP RAB$V_LOC
Lock nonexistent record	Indicates that VMS RMS is to lock the record position at the location of the following record operation, regardless of whether a record exists at that location. Applies only to relative files. FDL: CONNECT NONEXISTENT_ RECORD VMS RMS: RAB$L_ROP RAB$V_NXR
Lock for read	Locks record for reading and allow other readers (but no writers). FDL: CONNECT LOCK_ON_READ VMS RMS: RAB$L_ROP RAB$V_REA
Lock for write	Locks record for writing and allows other readers (but no writers). FDL: CONNECT LOCK_ON_WRITE VMS RMS: RAB$L_ROP RAB$V_RLK
Manual locking	Allows you to control record locking and unlocking manually. FDL: CONNECT MANUAL_LOCKING VMS RMS: RAB$L_ROP RAB$V_ULK
Read ahead	Improves performance at the expense of additional memory for I/O buffers. For sequential access to sequential files only. FDL: CONNECT READ_AHEAD VMS RMS: RAB$L_ROP RAB$V_RAH
Read regardless	Reads the specified record regardless of whether it is locked by another user. FDL: CONNECT READ_REGARDLESS VMS RMS: RAB$L_ROP RAB$V_RRL
Record access	Specifies the way records are accessed, sequentially, randomly by key (indexed files), by record number (relative files), or randomly by RFA. FDL: None. VMS RMS: RAB$B_RAC values RAB$C_SEQ, RABC_KEY, RABC_RFA

Option	Description
RFA	Specifies the address of the desired record when records are accessed randomly by RFA (RAB$B_RAC contains RAB$C_RFA). This value is also returned by Find and Get services regardless of the type record access used. FDL: None. VMS RMS: RAB$W_RFA
Record header buffer	Contains the symbolic address of the record header buffer that contains the fixed portion of a VFC record. Applies to the Get service only. FDL: None. VMS RMS: RAB$L_RHB
Timeout period	If the wait-if-locked option is specified, this option may be specified to specify a timeout period after which an error is returned. The number of seconds is specified by the CONNECT TIMEOUT_PERIOD or RAB$B_TMO field to eliminate a potential deadlock. FDL: CONNECT TIMEOUT_PERIOD VMS RMS: RAB$L_ROP RAB$V_TMO and RAB$B_TMO
User buffer address	Specifies the address of the user buffer that receives the record. Applies to the Get service only. FDL: None. VMS RMS: RAB$L_UBF
User buffer size	Specifies the maximum length of the user record buffer. Applies to the Get service only. FDL: None. VMS RMS: RAB$L_USZ
Wait if locked	Specifies that if the record is locked, VMS RMS must wait until it is available; also allows use of the wait-timeout-period option. FDL: CONNECT WAIT_FOR_RECORD VMS RMS: RAB$L_ROP RAB$V_WAT

9.3.2 Put Service Options

The Put service (or equivalent VAX language statement) adds a record to the file.

The options associated with the Put service are summarized in the following table. These options can be set for *each* Put service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VMS RMS service calls.

Option	Description
Asynchronous record processing	Specifies that record I/O for this record stream is done asynchronously. FDL: CONNECT ASYNCHRONOUS VMS RMS: RAB$L_ROP RAB$V_ASY
Key buffer	When adding records randomly to a relative file, the specified key buffer must contain the desired record's relative record number. FDL: None. VMS RMS: RAB$L_KBF
Key size	When adding records to a relative file using random record access, this field must specify a value of 4 (the default value provided by VMS RMS). FDL: None. VMS RMS: RAB$B_KSZ
Load buckets	When adding records, the buckets fill to the level specified when the file is created. The default is that buckets fill completely before a bucket split occurs. FDL: CONNECT FILL_BUCKETS VMS RMS: RAB$L_ROP RAB$V_LOA
Read allowed	Allows the locked record being written to be read. FDL: CONNECT LOCK_ON_WRITE VMS RMS: RAB$L_ROP RAB$V_RLK
Record access	Specifies the way records are added, sequentially according to ascending key value or relative record number, randomly by key (indexed files) or by record number (relative files), or randomly by RFA. FDL: None. VMS RMS: RAB$B_RAC values RAB$C_SEQ, RABC_KEY, RABC_RFA
Record header buffer	Contains the symbolic address of the record header buffer that contains the fixed portion of a VFC record. Applies to the Get service only. FDL: None. VMS RMS: RAB$L_RHB
Record buffer address	Specifies the address of the record buffer that contains the record to be written. FDL: None. VMS RMS: RAB$L_RBF
Record buffer size	Specifies the size of the record contained in the record buffer to be written. FDL: None. VMS RMS: RAB$L_RBZ

Option	Description
Timeout period	This option is used with the wait-if-locked option to specify a timeout period after which an error is returned. The number of seconds is specified by the CONNECT TIMEOUT_PERIOD or the RAB$B_TMO field to eliminate a potential deadlock. FDL: CONNECT TIMEOUT_PERIOD VMS RMS: RAB$L_ROP RAB$V_TMO and RAB$B_TMO
Truncate on Put	Specifies that the file is truncated at the record being added. Requires sequential record access and only applies to sequential files. FDL: CONNECT TRUNCATE_ON_PUT VMS RMS: RAB$L_ROP RAB$V_TPT
Update-if	Turns the Put service into an update operation if the record already exists in the file. Care must be taken when using this option with shared files and automatic record locking (see Section 8.1). When using this option with indexed files, note that the file must not allow duplicates for the primary key. This option can only be used when random record access has been specified. FDL: CONNECT UPDATE_IF VMS RMS: RAB$L_ROP RAB$V_UIF
Write-behind	Improves performance at the expense of additional memory for I/O buffers. Requires sequential record access and only applies to sequential files. FDL: CONNECT WRITE_BEHIND VMS RMS: RAB$L_ROP RAB$V_WBH

9.3.3 Record Update Options

The Update service (or equivalent VAX language statement) modifies an existing record in a file. Your program must first locate the appropriate record position and optionally retrieve the record itself by calling the Find or Get service (or equivalent VAX language statement).

The options associated with the Update service are summarized in the following table. These options can be set for *each* Update service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VMS RMS service calls.

Option	Description
Asynchronous record processing	Specifies that record I/O for this record stream is done asynchronously. FDL: CONNECT ASYNCHRONOUS VMS RMS: RAB$L_ROP RAB$V_ASY

Option	Description
Record header buffer	Contains the symbolic address of the record header buffer that contains the fixed portion of a VFC record. Applies to the Get service only. FDL: None. VMS RMS: RAB$L_RHB
Record buffer address	Specifies the address of the record buffer that contains the record to be written. FDL: None. VMS RMS: RAB$L_RBF
Record buffer size	Specifies the size of the records contained in the record buffer to be written. FDL: None. VMS RMS: RAB$W_RSZ

9.3.4 Record Deletion Options

The Delete service (or equivalent VAX language statement) removes a record from the file. You cannot use this service for sequential files; however, a sequential file can be truncated using the Truncate service. Like the Update service, the Delete service must be preceded by a Find or Get service to establish the current record position.

The options associated with the Delete service are summarized in the following table. These options can be set for *each* Delete service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VMS RMS service calls.

Option	Description
Asynchronous record processing	Specifies that record I/O for FDL: CONNECT ASYNCHRONOUS VMS RMS: RAB$L_ROP RAB$V_ASY
Fast delete	Specifies that the record to be deleted is flagged as deleted, but parts of any alternate index key path are not completely erased until a subsequent access using the alternate key occurs. This makes deleting the record occur more quickly, but it requires additional access time for a subsequent Find or Get service. FDL: CONNECT FAST_DELETE VMS RMS: RAB$L_ROP RAB$V_FDL

9.4 Run-Time Example

Example 9–2 shows how to invoke the FDL$PARSE and FDL$RELEASE routines to use the predefined control block values set by an EDIT/FDL editing session.

Example 9–2 Using the FDL$PARSE and FDL$RELEASE Routines

```
;
;  This program calls the FDL utility routines FDL$PARSE and
;  FDL$RELEASE.  First, FDL$PARSE parses the FDL specification
;  PART.FDL.  Then the data file named in PART.FDL is accessed
;  using the primary key.  Last, the control blocks allocated
;  by FDL$PARSE are released by FDL$RELEASE.
;
                .TITLE  FDLEXAM
;
                .PSECT  DATA,WRT,NOEXE

;
MY_FAB:         .LONG   0
MY_RAB:         .LONG   0
FDL_FILE:       .ASCID  /PART.FDL/          ; Declare FDL file
REC_SIZE=80
LF=10
REC_RESULT:     .LONG   REC_SIZE
                .ADDRESS REC_BUFFER
REC_BUFFER:     .BLKB   REC_SIZE
HEADING:        .ASCID /ID   PART   SUPPLIER        COLOR /LF
;
                .PSECT  CODE
;
;  Declare the external routines
;
.EXTRN          FDL$PARSE, -
                FDL$RELEASE
                                            ;
.ENTRY          FDLEXAM,^M<>                ; Set up entry mask
                PUSHAL  MY_RAB              ; Get set up for call with
                PUSHAL  MY_FAB              ; addresses to receive the
                PUSHAL  FDL_FILE            ; FAB and RAB allocated by
                CALLS   #3,G^FDL$PARSE      ; FDL$PARSE
                BLBS    R0,KEY0             ; Branch on success
                BRW     ERROR               ; Signal error
                                            ;
KEY0:           MOVL    MY_FAB,R10          ; Move address of FAB to R10
                MOVL    MY_RAB,R9           ; Move address of RAB to R9
                MOVL    #REC_SIZE,RAB$W_USZ(R9)
                MOVAB   REC_BUFFER,RAB$L_UBF(R9)
                $OPEN   FAB=(R10)           ; Open the file
                BLBC    R0,F_ERROR
                $CONNECT RAB=(R9)           ; Connect to the RAB
                BLBC    R0,R_ERROR
                PUSHAQ  HEADING             ; Display the heading
                CALLS   #1,G^LIB$PUT_OUTPUT
                BLBC    R0,ERROR
                BRB     GET_REC             ; Skip error handling
                                            ;
```

Example 9–2 Cont'd. on next page

Example 9–2 (Cont.) Using the FDL$PARSE and FDL$RELEASE Routines

```
F_ERROR:        BRW     FAB_ERROR
R_ERROR:        BRW     RAB_ERROR
                                            ;
GET_REC:        $GET    RAB=(R9)            ; Get a record
                CMPL    #RMS$_EOF,R0        ; If not end of file,
                BEQLU   CLEAN               ; continue
                BLBC    R0,R_ERROR
                MOVZWL  RAB$W_RSZ(R9),REC_RESULT  ; Move a record into
                PUSHAL  REC_RESULT                ; the buffer
                CALLS   #1,G^LIB$PUT_OUTPUT       ; Display the record
                BLBC    R0,ERROR
                BRB     GET_REC             ; Get another record
                                            ;
CLEAN:          $CLOSE  FAB=(R10)           ; Close the FAB
                BLBC    R0,FAB_ERROR
                PUSHAL  MY_RAB              ; Push RAB address on stack
                PUSHAL  MY_FAB              ; Push FAB address on stack
                CALLS   #2,G^FDL$RELEASE    ; Release the control blocks
                BLBC    R0,ERROR
                BRB     FINI                ; Successful completion
                                            ;
FAB_ERROR:
                PUSHL   FAB$L_STV(R10)      ; Signal file error
                PUSHL   FAB$L_STS(R10)
                BRB     RMS_ERR
                                            ;
ERROR:          PUSHL   R0                  ; Signal error
                CALLS   #1,G^LIB$SIGNAL
                $CLOSE  FAB=(R10)
                BRW     FINI                ; End program
                                            ;
RAB_ERROR:
                PUSHL   RAB$L_STV(R9)       ; Signal record error
                PUSHL   RAB$L_STS(R9)
                                            ;
RMS_ERR:
                CALLS   #2,G^LIB$SIGNAL
                                            ;
FINI:           RET
                .END FDLEXAM
```

10 Maintaining Files

Designing and creating your files and defining their records are only the first steps in the life cycle of your file. You must also consider maintaining the file.

This chapter describes file maintenance with the emphasis on file tuning.

Section 10.1 describes how you can use the Analyze/RMS_File Utility to view the characteristics of a file. Section 10.2 describes how you can create an FDL file from a data file using the Analyze/RMS_File Utility. Section 10.3 explains how to use the Edit/FDL Utility, particularly with Analyze/RMS_File, to optimize and redesign file characteristics. Section 10.4 describes how to make a file contiguous. Section 10.5 explains how to reorganize a file, and Section 10.6 describes how to make archive copies of a file.

10.1 Viewing File Characteristics

The Analyze/RMS_File Utility (ANALYZE/RMS_FILE) allows you to inspect and analyze the internal structure of a VMS RMS file.

ANALYZE/RMS_FILE can check a file's structure for errors and can generate a statistical or summary report. A summary report is identical to a statistical report except that no checking is done. For more information on producing a summary report, see the description of the Analyze/RMS_File Utility in the *VMS Analyze/RMS_File Utility Manual.*

You can also inspect and analyze your file using the Analyze/RMS_File Utility interactively. The analysis can show whether or not the file is properly designed for its application and can point out ways to improve the file design.

In addition, you can use ANALYZE/RMS_FILE to FDL files from data files. You can then use these FDL files with the Create/FDL Utility (CREATE/FDL), the Convert Utility (CONVERT), and the Edit/FDL Utility, (EDIT/FDL). FDL files created with ANALYZE/RMS_FILE contain special analysis sections for each area and key, which are called ANALYSIS_OF_AREA and ANALYSIS_OF_KEY. The Edit/FDL Utility uses these sections in the Optimize script to tune the file's structure.

10.1.1 Performing an Error Check

To check a file's structure for errors, use the following command syntax:

```
ANALYZE/RMS_FILE/CHECK filespec
```

By default with a command of this format, the Check report is displayed on the terminal (SYS$OUTPUT).

If you receive any error messages, the file has been corrupted by a serious error. If you have had a hardware problem such as a power failure or a disk head failure, then the hardware probably caused the corruption. If you have not had any hardware problems, then a software error may have caused the corruption. Note that the /CHECK qualifier does not find all types of file corruption, however.

In either case, you can try using the Convert Utility to fix the problem by using the file specification as both the **input-filespec** and the **output-filespec**. This operation will reorganize the file; if it does not work, use the Backup Utility (BACKUP) to bring in the backup copy of the file. For more information on both CONVERT and BACKUP, see Sections 10.4.2, 10.5, and 10.6.

Note: **If you believe that the software caused the error, submit a Software Performance Report (SPR). Always include the ANALYZE/RMS_FILE check report, a copy of the data file, and a description of what was done with the data file. If possible, also supply a version of the file prior to the corruption and the program or procedure which led to the corruption; being able to reproduce the problem is of tremendous value.**

Example 10–1 is a sample Check report of a file with the file specification DISK$:[HERBER]CUSTDATA.DAT;2.

Example 10–1 Using ANALYZE/RMS_FILE to Create a Check Report

```
Check RMS File Integrity                     14-JUN-1985 21:51:47.38   Page 1
DISK$:[HERBER]CUSTDATA.DAT;2

FILE HEADER

        File Spec: DISK$:[HERBER]CUSTDATA.DAT;2
        File ID: (10044,39,1)
        Owner UIC: [011,310]
        Protection:  System: RWED, Owner: RWED, Group: RWE, World: RWE
        Creation Date:    9-JUN-1985 22:30:24.78
        Revision Date:    9-JUN-1985 22:30:30.86, Number: 4
        Expiration Date: none specified
        Backup Date:     none posted
        Contiguity Options:  none
        Performance Options: none
        Reliability Options: none
        Journaling Enabled: none

RMS FILE ATTRIBUTES

        File Organization: indexed
        Record Format: variable
        Record Attributes:  carriage-return
        Maximum Record Size: 80
        Blocks Allocated: 30, Default Extend Size: 2
        Bucket Size: 1
        Global Buffer Count: 0

FIXED PROLOG

        Number of Areas: 8, VBN of First Descriptor: 3
        Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')
```

Example 10–1 Cont'd. on next page

Example 10–1 (Cont.) Using ANALYZE/RMS_FILE to Create a Check Report

```
        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 1, Blocks: 9, Used: 4, Next: 5
        Default Extend Quantity: 2
        Total Allocation: 9

AREA DESCRIPTOR #1 (VBN 3, offset %X'0040')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 10, Blocks: 3, Used: 1, Next: 11
        Default Extend Quantity: 1
        Total Allocation: 3
AREA DESCRIPTOR #2 (VBN 3, offset %X'0080')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 13, Blocks: 3, Used: 1, Next: 14
        Default Extend Quantity: 1
        Total Allocation: 3

AREA DESCRIPTOR #3 (VBN 3, offset %X'00C0')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 16, Blocks: 3, Used: 1, Next: 17
        Default Extend Quantity: 1
        Total Allocation: 3

AREA DESCRIPTOR #4 (VBN 3, offset %X'0100')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 19, Blocks: 3, Used: 1, Next: 20
        Default Extend Quantity: 1
        Total Allocation: 3

AREA DESCRIPTOR #5 (VBN 3, offset %X'0140')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 22, Blocks: 3, Used: 1, Next: 23
        Default Extend Quantity: 1
        Total Allocation: 3

AREA DESCRIPTOR #6 (VBN 3, offset %X'0180')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 25, Blocks: 3, Used: 1, Next: 26
        Default Extend Quantity: 1
        Total Allocation: 3

AREA DESCRIPTOR #7 (VBN 3, offset %X'01C0')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 28, Blocks: 3, Used: 1, Next: 29
        Default Extend Quantity: 1
        Total Allocation: 3
```

Example 10–1 Cont'd. on next page

Example 10–1 (Cont.) Using ANALYZE/RMS_FILE to Create a Check Report

```
KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

        Next Key Descriptor VBN: 2, Offset: %X'0000'
        Index Area: 1, Level 1 Index Area: 1, Data Area: 0
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 10
        Key Flags:
                (0)  KEY$V_DUPKEYS    0
                (3)  KEY$V_IDX_COMPR  0
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  0
                (7)  KEY$V_REC_COMPR  1

        Key Segments: 1
        Key Size: 4
        Minimum Record Size: 4
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      0
        Segment Sizes:          4
        Data Type: string
        Name: "PART_NUM"
        First Data Bucket VBN: 4

KEY DESCRIPTOR #1 (VBN 2, offset %X'0000')

        Next Key Descriptor VBN: 2, Offset: %X'0066'
        Index Area: 3, Level 1 Index Area: 3, Data Area: 2
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 16
        Key Flags:
                (0)  KEY$V_DUPKEYS    1
                (1)  KEY$V_CHGKEYS    0
                (2)  KEY$V_NULKEYS    0
                (3)  KEY$V_IDX_COMPR  0
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  0
        Key Segments: 1
        Key Size: 5
        Minimum Record Size: 9
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      4
        Segment Sizes:          5
        Data Type: string
        Name: "PART_NAME"
        First Data Bucket VBN: 13
```

Example 10–1 Cont'd. on next page

Example 10–1 (Cont.) Using ANALYZE/RMS_FILE to Create a Check Report

```
KEY DESCRIPTOR #2 (VBN 2, offset %X'0066')

        Next Key Descriptor VBN: 2, Offset: %X'00CC'
        Index Area: 5, Level 1 Index Area: 5, Data Area: 4
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 22
        Key Flags:
                (0)  KEY$V_DUPKEYS    1
                (1)  KEY$V_CHGKEYS    0
                (2)  KEY$V_NULKEYS    0
                (3)  KEY$V_IDX_COMPR  1
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  1
        Key Segments: 1
        Key Size: 10
        Minimum Record Size: 19
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      9
        Segment Sizes:         10
        Data Type: string
        Name: "SUPPLIER_NAME"
        First Data Bucket VBN: 19

KEY DESCRIPTOR #3 (VBN 2, offset %X'00CC')

        Index Area: 7, Level 1 Index Area: 7, Data Area: 6
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 28
        Key Flags:
                (0)  KEY$V_DUPKEYS    1
                (1)  KEY$V_CHGKEYS    0
                (2)  KEY$V_NULKEYS    0
                (3)  KEY$V_IDX_COMPR  1
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  1
        Key Segments: 1
        Key Size: 10
        Minimum Record Size: 29
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:     19
        Segment Sizes:         10
        Data Type: string
        Name: "COLOR"
        First Data Bucket VBN: 25

The analysis uncovered NO errors.

ANALYZE/RMS_FILE/OUTPUT=CUSTDATA.ANL CUSTDATA.DAT
```

To place the Check report in a file, use a command of the form

ANALYZE/RMS_FILE/CHECK/OUTPUT=output-filespec input-filespec

The Check report will be placed in the file you named with the **output-filespec** parameter. This file will receive the file type ANL by default. For example, the following command will perform an error check on PRLG2.IDX and place the Check report in the file ERROR.ANL:

```
$ ANALYZE/RMS_FILE/CHECK/OUTPUT=ERROR PRLG2.IDX
```

10.1.2 Generating a Statistics Report

For indexed files, the Statistics report consists of the Check report plus additional information about the areas and keys in the file. (A Statistics report on a sequential or relative file is thus the same as a Check report.)

To generate a Statistics report with ANALYZE/RMS_FILE, enter a DCL command of the form

ANALYZE/RMS_FILE/STATISTICS filespec

Example 10–2 is an example of a Statistics report.

Example 10–2 Using ANALYZE/RMS_FILE to Create a Statistics Report

```
RMS File Statistics                          18-APR-1985 11:22:27.14   Page 1
DISK$:[TEST.PROGRAM]INDEX.DAT;1

FILE HEADER

        File Spec: DISK$:[TEST.PROGRAM]INDEX.DAT;1
        File ID: (15960,8,0)
        Owner UIC: [011,310]
        Protection:  System: RWED, Owner: RWED, Group: RWED, World: RWE
        Creation Date:   19-APR-1985 22:15:55.70
        Revision Date:   19-APR-1985 22:16:01.74, Number: 4
        Expiration Date: none specified
        Backup Date:     18-APR-1985 00:57:54.24
        Contiguity Options:  contiguous-best-try
        Performance Options: none
        Reliability Options: none
        Journaling Enabled:  none
RMS FILE ATTRIBUTES

        File Organization: indexed
        Record Format: variable
        Record Attributes:   carriage-return
        Maximum Record Size: 80
        Blocks Allocated: 30, Default Extend Size: 2
        Bucket Size: 1
        Global Buffer Count: 0

FIXED PROLOG

        Number of Areas: 8, VBN of First Descriptor: 3
        Prolog Version: 3
AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 1, Blocks: 9, Used: 4, Next: 5
        Default Extend Quantity: 2
        Total Allocation: 9
STATISTICS FOR AREA #0

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #1 (VBN 3, offset %X'0040')
```

Example 10–2 Cont'd. on next page

Example 10–2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

```
        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 10, Blocks: 3, Used: 1, Next: 11
        Default Extend Quantity: 1
        Total Allocation: 3

STATISTICS FOR AREA #1

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #2 (VBN 3, offset %X'0080')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 13, Blocks: 3, Used: 1, Next: 14
        Default Extend Quantity: 1
        Total Allocation: 3

STATISTICS FOR AREA #2

        Count of Reclaimed Blocks:              0
AREA DESCRIPTOR #3 (VBN 3, offset %X'00C0')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 16, Blocks: 3, Used: 1, Next: 17
        Default Extend Quantity: 1
        Total Allocation: 3

STATISTICS FOR AREA #3

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #4 (VBN 3, offset %X'0100')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 19, Blocks: 3, Used: 1, Next: 20
        Default Extend Quantity: 1
        Total Allocation: 3

STATISTICS FOR AREA #4

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #5 (VBN 3, offset %X'0140')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 22, Blocks: 3, Used: 1, Next: 23
        Default Extend Quantity: 1
        Total Allocation: 3
```

Example 10–2 Cont'd. on next page

Example 10–2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

```
STATISTICS FOR AREA #5

        Count of Reclaimed Blocks:             0

AREA DESCRIPTOR #6 (VBN 3, offset %X'0180')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 25, Blocks: 3, Used: 1, Next: 26
        Default Extend Quantity: 1
        Total Allocation: 3

STATISTICS FOR AREA #6

        Count of Reclaimed Blocks:             0

AREA DESCRIPTOR #7 (VBN 3, offset %X'01C0')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 28, Blocks: 3, Used: 1, Next: 29
        Default Extend Quantity: 1
        Total Allocation: 3
STATISTICS FOR AREA #7

        Count of Reclaimed Blocks:             0

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

        Next Key Descriptor VBN: 2, Offset: %X'0000'
        Index Area: 1, Level 1 Index Area: 1, Data Area: 0
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 10
        Key Flags:
                (0)  KEY$V_DUPKEYS    0
                (3)  KEY$V_IDX_COMPR  0
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  0
                (7)  KEY$V_REC_COMPR  1
        Key Segments: 1
        Key Size: 4
        Minimum Record Size: 4
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      0
        Segment Sizes:          4
        Data Type: string
        Name: "ID_NUM"
        First Data Bucket VBN: 4

STATISTICS FOR KEY #0

        Number of Index Levels:                1
        Count of Level 1 Records:              1
        Mean Length of Index Entry:            6
        Count of Index Blocks:                 1
        Mean Index Bucket Fill:                4%
        Mean Index Entry Compression:          0%
```

Example 10–2 Cont'd. on next page

Example 10–2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

```
        Count of Data Records:                 10
        Mean Length of Data Record:            33
        Count of Data Blocks:                   1
        Mean Data Bucket Fill:                 90%
        Mean Data Key Compression:              0%
        Mean Data Record Compression:          -2%

        Overall Space Efficiency:               2%
KEY DESCRIPTOR #1 (VBN 2, offset %X'0000')

        Next Key Descriptor VBN: 2, Offset: %X'0066'
        Index Area: 3, Level 1 Index Area: 3, Data Area: 2
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 16
        Key Flags:
                (0)  KEY$V_DUPKEYS    1
                (1)  KEY$V_CHGKEYS    0
                (2)  KEY$V_NULKEYS    0
                (3)  KEY$V_IDX_COMPR  0
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  0
        Key Segments: 1
        Key Size: 5
        Minimum Record Size: 9
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:       4
        Segment Sizes:           5
        Data Type: string
        Name: "ID_NAME"
        First Data Bucket VBN: 13

STATISTICS FOR KEY #1

        Number of Index Levels:                 1
        Count of Level 1 Records:               1
        Mean Length of Index Entry:             7
        Count of Index Blocks:                  1
        Mean Index Bucket Fill:                 4%
        Mean Index Entry Compression:           0%

        Count of Data Records:                  6
        Mean Duplicates per Data Record:        0
        Mean Length of Data Record:            19
        Count of Data Blocks:                   1
        Mean Data Bucket Fill:                 24%
        Mean Data Key Compression:              0%
```

Example 10–2 Cont'd. on next page

Example 10–2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

```
KEY DESCRIPTOR #2 (VBN 2, offset %X'0066')

        Next Key Descriptor VBN: 2, Offset: %X'00CC'
        Index Area: 5, Level 1 Index Area: 5, Data Area: 4
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 22
        Key Flags:
                (0)  KEY$V_DUPKEYS    1
                (1)  KEY$V_CHGKEYS    0
                (2)  KEY$V_NULKEYS    0
                (3)  KEY$V_IDX_COMPR  1
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  1
        Key Segments: 1
        Key Size: 10
        Minimum Record Size: 19
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      9
        Segment Sizes:         10
        Data Type: string
        Name: "ADDRESS"
        First Data Bucket VBN: 19

STATISTICS FOR KEY #2

        Number of Index Levels:              1
        Count of Level 1 Records:            1
        Mean Length of Index Entry:         12
        Count of Index Blocks:               1
        Mean Index Bucket Fill:              4%
        Mean Index Entry Compression:       58%

        Count of Data Records:               7
        Mean Duplicates per Data Record:     0
        Mean Length of Data Record:         20
        Count of Data Blocks:                1
        Mean Data Bucket Fill:              30%
        Mean Data Key Compression:          21%
```

Example 10–2 Cont'd. on next page

Example 10–2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

```
KEY DESCRIPTOR #3 (VBN 2, offset %X'00CC')

        Index Area: 7, Level 1 Index Area: 7, Data Area: 6
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 28
        Key Flags:
                (0)  KEY$V_DUPKEYS    1
                (1)  KEY$V_CHGKEYS    0
                (2)  KEY$V_NULKEYS    0
                (3)  KEY$V_IDX_COMPR  1
                (4)  KEY$V_INITIDX    0
                (6)  KEY$V_KEY_COMPR  1
        Key Segments: 1
        Key Size: 10
        Minimum Record Size: 29
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      19
        Segment Sizes:          10
        Data Type: string
        Name: "CHARGES"
        First Data Bucket VBN: 25

STATISTICS FOR KEY #3

        Number of Index Levels:                 1
        Count of Level 1 Records:               1
        Mean Length of Index Entry:            12
        Count of Index Blocks:                  1
        Mean Index Bucket Fill:                 4%
        Mean Index Entry Compression:          58%

        Count of Data Records:                  5
        Mean Duplicates per Data Record:        1
        Mean Length of Data Record:            23
        Count of Data Blocks:                   1
        Mean Data Bucket Fill:                 25%
        Mean Data Key Compression:             34%


The analysis uncovered NO errors.

ANALYZE/RMS_FILE/OUTPUT=INDEX/STATISTICS INDEX.DAT
```

10.1.3 Using Interactive Mode

The /INTERACTIVE qualifier begins an interactive session in which you can examine the structure of a VMS RMS file.

ANALYZE/RMS_FILE imposes a hierarchical tree structure on the internal VMS RMS file structure. Each data structure in the file is a node, with a branch for each pointer in the data structure. The file header is always the root node. Each of the three file organizations (sequential, relative, and indexed) has its own tree structure.

To examine a file, you enter commands that move the current position to particular structures within the tree. The utility displays the current structure on the screen.

Table 10–1 summarizes the ANALYZE/RMS_FILE commands.

Table 10–1 ANALYZE/RMS_FILE Command Summary

Command	Function
AGAIN	Displays the current structure again.
DOWN [branch]	Moves the structure pointer down to the next level. If the current node has more than one branch, the branch keyword must be specified. If a branch keyword is required but not specified, the utility will display a list of possibilities to prompt you. You can also display the list by specifying "DOWN ?."
DUMP n	Displays a hexadecimal dump of the specified block.
EXIT	Ends the interactive session.
FIRST	Moves the structure pointer to the first structure on the current level. The structure is displayed. For example, if you are examining data buckets and want to examine the first bucket, this command will put you there and display the first bucket's header.
HELP [keyword ...]	Displays help messages about the interactive commands.
NEXT	Moves the structure pointer to the next structure on the current level. The structure is displayed. Pressing the RETURN key is equivalent to a NEXT command.
REST	Moves the structure pointer along the rest of the structures on the current level, and each is displayed in turn.
TOP	Moves the structure pointer up to the file header. The file header is displayed.
UP	Moves the structure pointer up to the next level. The structure at that level is displayed.

10.1.4 Examining a Sequential File

Figure 10–1 shows the tree structure of a sequential file.

The FILE HEADER structure is always the first structure displayed. From the FILE HEADER structure, the DOWN command moves the current position to the FILE ATTRIBUTES structure. The DOWN command from the FILE ATTRIBUTES structure moves the current position to the first record in the file. From the first record, the REST command will move the current position through the records in the file, displaying each one in turn. A series of NEXT commands will also accomplish this same operation.

Figure 10–1 Tree Structure for Sequential Files

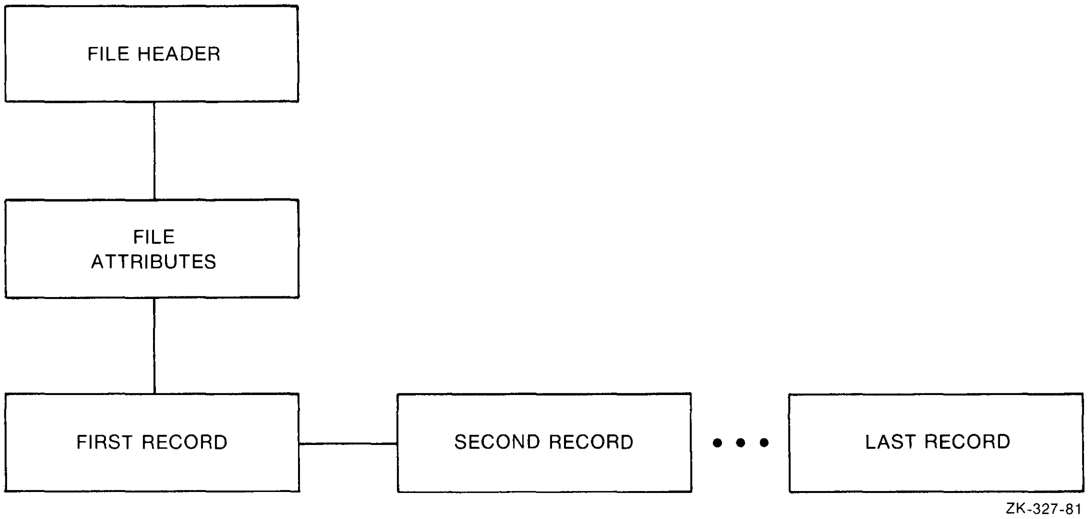
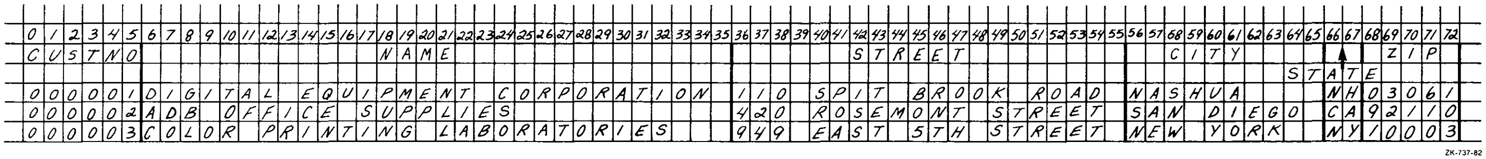


Figure 10–2 shows the layout and contents of the records in a sequential file SEQ.DAT. Example 10–3 is an interactive examination of SEQ.DAT, showing the contents of three records in the file.

Figure 10–2 Record Layout and Content for SEQ.DAT



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
C	U	S	T	N	O													N	A	M	E																					S	T	R	E	E	T											C	I	T	Y					↑			Z	I	P	
																																																																S	T	A	T	E				
0	0	0	0	0	1	D	I	G	I	T	A	L		E	Q	U	I	P	M	E	N	T		C	O	R	P	O	R	A	T	I	O	N		1	1	0		S	P	I	T		B	R	O	O	K		R	O	A	D		N	A	S	H	U	A					N	H	0	3	0	6	1
0	0	0	0	0	2	A	D	B		O	F	F	I	C	E		S	U	P	P	L	I	E	S												4	2	0		R	O	S	E	M	O	N	T		S	T	R	E	E	T		S	A	N		D	I	E	G	O		C	A	9	2	1	1	0
0	0	0	0	0	3	C	O	L	O	R		P	R	I	N	T	I	N	G		L	A	B	O	R	A	T	O	R	I	E	S				9	4	9		E	A	S	T		5	T	H		S	T	R	E	E	T		N	E	W		Y	O	R	K			N	Y	1	0	0	0	3

ZK-737-82

Example 10–3 Examining a Sequential File

```
$ ANALYZE/RMS_FILE/INTERACTIVE SEQ.DAT

FILE HEADER
       File Spec: DISK$DELPHIWORK:[RMS32]SEQ.DAT;3
       File ID: (1170,2,2)
       Owner UIC: [730,465]
       Protection:  System: RWED, Owner: RWED, Group: RWED, World:
       Creation Date:    7-MAY-1985 16:51:30.92
       Revision Date:    8-MAY-1985 14:02:17.15, Number: 3
       Expiration Date: none specified
       Backup Date:     none posted
       Contiguity Options:  none
       Performance Options: none
       Reliability Options: none

ANALYZE> DOWN
RMS FILE ATTRIBUTES
       File Organization: sequential
       Record Format: variable
       Record Attributes:   carriage-return
       Maximum Record Size: 0
       Longest Record: 73
       Blocks Allocated: 3, Default Extend Size: 0
       End-of-File VBN: 1, Offset: %X'00E4'

ANALYZE> DOWN
       DATA BYTES (VBN 1, offset %X'0000'):
                       7  6  5  4  3  2  1  0          01234567
                     ------------------------          --------
                      31 30 30 30 30 30 00 49|  0000  |I.000001|
                      20 4C 41 54 49 47 49 44|  0008  |DIGITAL |
                      4E 45 4D 50 49 55 51 45|  0010  |EQUIPMEN|
                      52 4F 50 52 4F 43 20 54|  0018  |T CORPOR|
                      31 31 20 4E 4F 49 54 41|  0020  |ATION 11|
                      42 20 54 49 50 53 20 30|  0028  |0 SPIT B|
                      41 4F 52 20 4B 4F 4F 52|  0030  |ROOK ROA|
                      41 55 48 53 41 4E 20 44|  0038  |D NASHUA|
                      33 30 48 4E 20 20 20 20|  0040  |    NH03|
                                  00 31 36 30|  0048  |061.    |
ANALYZE> NEXT
       DATA BYTES (VBN 1, offset %X'004C'):
                       7  6  5  4  3  2  1  0          01234567
                     ------------------------          --------
                      32 30 30 30 30 30 00 49|  0000  |I.000002|
                      49 46 46 4F 20 42 44 41|  0008  |ADB OFFI|
                      4C 50 50 55 53 20 45 43|  0010  |CE SUPPL|
                      20 20 20 20 20 53 45 49|  0018  |IES     |
                      32 34 20 20 20 20 20 20|  0020  |      42|
                      4F 4D 45 53 4F 52 20 30|  0028  |0 ROSEMO|
                      45 52 54 53 20 54 4E 55|  0030  |UNT STRE|
                      49 44 20 4E 41 53 54 45|  0038  |ETSAN DI|
                      32 39 41 43 20 4F 47 45|  0040  |EGO CA92|
                                  00 30 31 31|  0048  |110.    |
```

Example 10–3 Cont'd. on next page

Example 10–3 (Cont.) Examining a Sequential File

```
ANALYZE> NEXT
      DATA BYTES (VBN 1, offset %X'0098'):
                            7  6  5  4  3  2  1  0         01234567
                           ------------------------       --------
                           33 30 30 30 30 30 00 49|  0000  |I.000003|
                           52 50 20 52 4F 4C 4F 43|  0008  |COLOR PR|
                           4C 20 47 4E 49 54 4E 49|  0010  |INTING L|
                           52 4F 54 41 52 4F 42 41|  0018  |ABORATOR|
                           34 39 20 20 20 53 45 49|  0020  |IES   94|
                           35 20 54 53 41 45 20 39|  0028  |9 EAST 5|
                           45 45 52 54 53 20 48 54|  0030  |TH STREE|
                           4F 59 20 57 45 4E 20 54|  0038  |T NEW YO|
                           30 31 59 4E 20 20 4B 52|  0040  |RK  NY10|
                                       00 33 30 30|  0048  |003.    |
ANALYZE> EXIT
```

10.1.5 Examining a Relative File

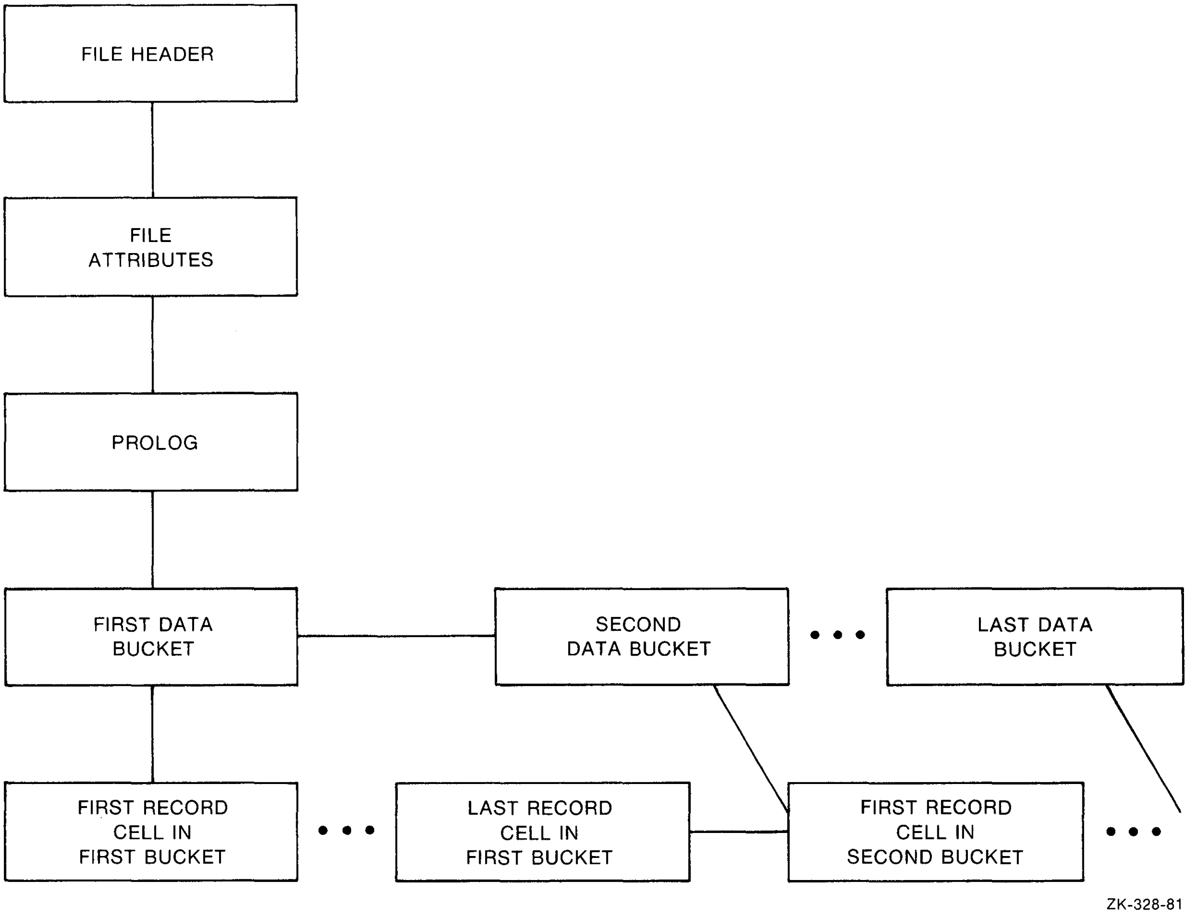
Figure 10–3 shows the tree structure of relative files.

The tree structure of relative files also begins with the FILE HEADER and FILE ATTRIBUTES structures. From the FILE ATTRIBUTES structure, the next structure down is the PROLOG. The first structure down from the PROLOG is the FIRST DATA BUCKET. The data bucket structures can be examined with the REST command or one at a time with the NEXT command. The only information at the data bucket level is the number of the data bucket's virtual block.

The next structure down is the FIRST RECORD CELL IN FIRST BUCKET. You can examine the records in each cell by specifying either the REST command or a series of NEXT commands.

Example 10–4 shows an interactive examination of a relative file.

Figure 10–3 Tree Structure of Relative Files



Example 10–4 Examining a Relative File

```
FILE HEADER
       File Spec: DISK$NEWWORK:[RMS32]REL.DAT;1
       File ID: (9573,7,2)
       Owner UIC: [181,065]
       Protection:  System: RWED, Owner: RWED, Group: RE, World:
       Creation Date:   22-MAY-1982 10:42:04.95
       Revision Date:   22-MAY-1982 10:42:05.81, Number: 1
       Expiration Date: none specified
       Backup Date:     none posted
       Contiguity Options:  contiguous-best-try
       Performance Options: none
       Reliability Options: none

ANALYZE> DOWN
RMS FILE ATTRIBUTES
       File Organization: relative
       Record Format: variable
       Record Attributes:   carriage-return
       Maximum Record Size: 75
       Blocks Allocated: 9, Default Extend Size: 0
       Bucket Size: 3
       Global Buffer Count: 0

ANALYZE> DOWN
FIXED PROLOG
       Prolog Flags:
              (0)  PLG$V_NOEXTEND    0
       First Data Bucket VBN: 2
       Maximum Record Number: 2147483647
       End-of-File VBN: 10
       Prolog Version: 1

ANALYZE> DOWN
DATA BUCKET (VBN 2)
ANALYZE> DOWN
       RECORD CELL (VBN 2, offset %X'0000'):
              Cell Control Flags:
                     (2)  DLC$V_DELETED     0
                     (3)  DCL$V_REC         1
              Record Bytes:
                       7  6  5  4  3  2  1  0           01234567
                     ------------------------          --------
                      31 30 30 30 30 30 00 49|  0000  |I.000001|
                      20 4C 41 54 49 47 49 44|  0008  |DIGITAL |
                      4E 45 4D 50 49 55 51 45|  0010  |EQUIPMEN|
                      52 4F 50 52 4F 43 20 54|  0018  |T CORPOR|
                      31 31 20 4E 4F 49 54 41|  0020  |ATION 11|
                      42 20 54 49 50 53 20 30|  0028  |0 SPIT B|
                      41 4F 52 20 4B 4F 4F 52|  0030  |ROOK ROA|
                      41 55 48 53 41 4E 20 44|  0038  |D NASHUA|
                      33 30 48 4E 20 20 20 20|  0040  |    NH03|
                                     31 36 30|  0048  |061     |
```

If you use the REST command at the CELL AND RECORD level, the utility will display all the cells and records in the file, not just the cells and records in the current bucket.

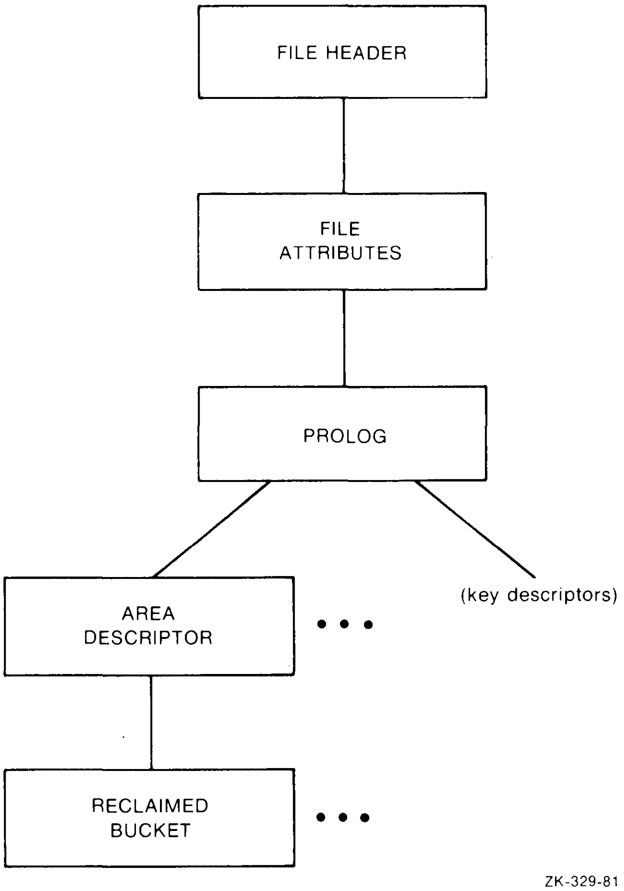
10.1.6 Examining an Indexed File

The structure of an indexed file also begins with the FILE HEADER, FILE ATTRIBUTES, and PROLOG structures. From the PROLOG structure, the file structure branches to the area descriptors and the key descriptors. To branch to the area descriptor path, specify the command DOWN AREA. To branch to the key descriptor path, specify DOWN KEY.

The area descriptor path contains structures that show information about the various areas in the file. The key descriptor path contains the primary key structures (and data records) and any secondary key structures.

Figure 10–4 shows the structure following the area descriptor path.

Figure 10–4 An Area Descriptor Path



Example 10–5 is an example of an examination of an area descriptor path from the PROLOG level.

Example 10–5 Examining an Area Descriptor Path

```
ANALYZE> DOWN AREA
AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')
       Bucket Size: 1
       Alignment: AREA$C_NONE
       Alignment Flags:
              (0)  AREA$V_HARD      0
              (1)  AREA$V_ONC       0
              (5)  AREA$V_CBT       0
              (7)  AREA$V_CTG       0
       Current Extent Start: 1, Blocks: 9, Used: 7, Next: 8
       Default Extend Quantity: 0
```

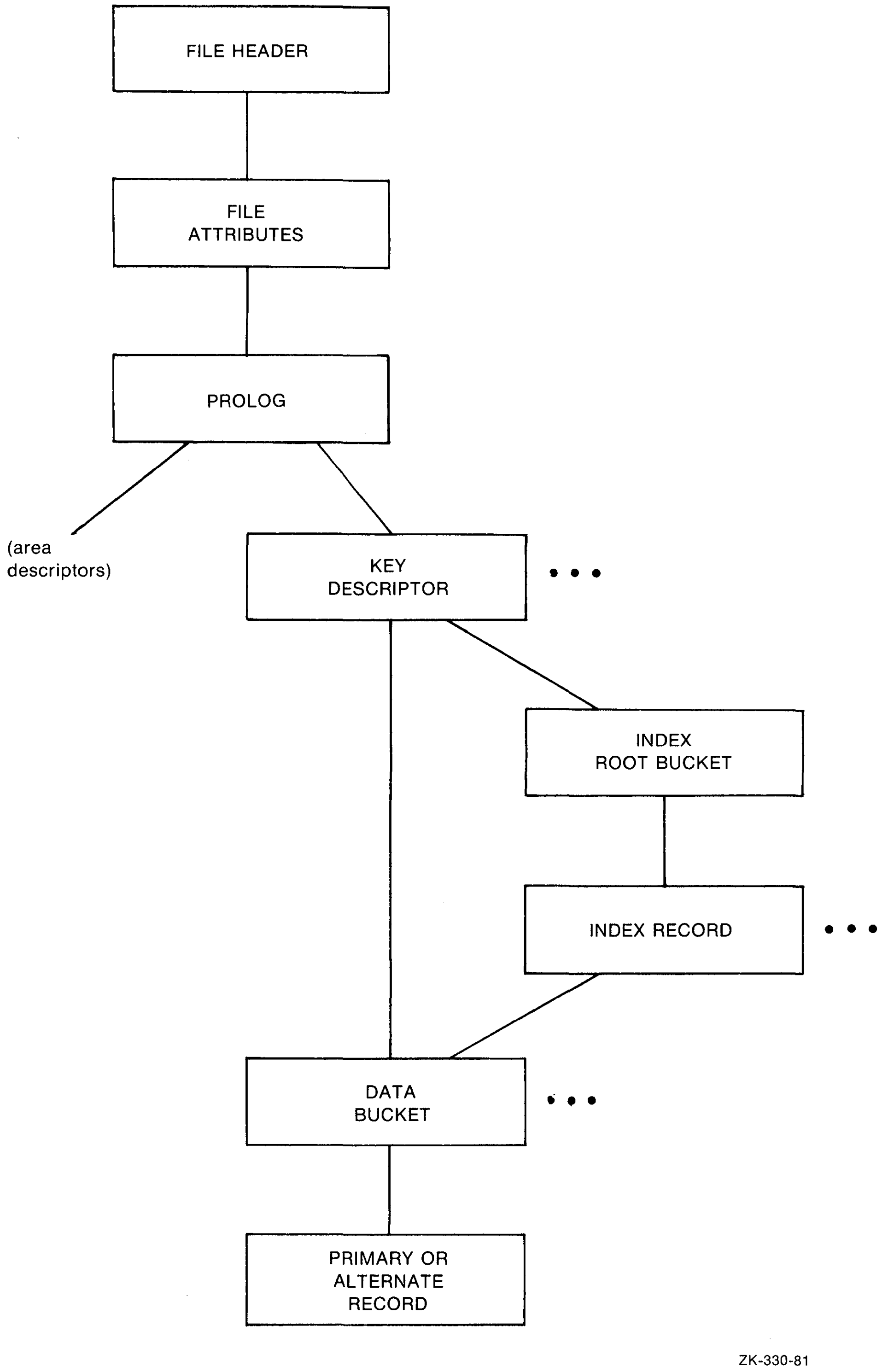
Figure 10–5 shows the structure following the key descriptor path.

As shown in Figure 10–5, you can branch directly to the DATA BUCKET, or you can branch to the INDEX ROOT BUCKET to begin examination of the index structure, eventually reaching the DATA BUCKET structure. Depending on whether you are examining the primary index structure or one of the alternate index structures, there is a difference in the contents of the record structure.

The PRIMARY RECORD structure contains the actual data records; the ALTERNATE RECORD structures contain secondary index data records (SIDRs).

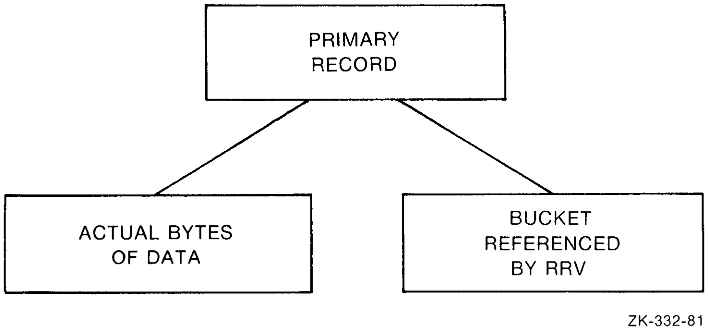
Figure 10–6 displays the structure of the primary records.

Figure 10–5 A Key Descriptor Path



As shown in Figure 10–6, the branch from the primary record structure allows you to either examine the actual bytes of data within the record or to follow the RRV.

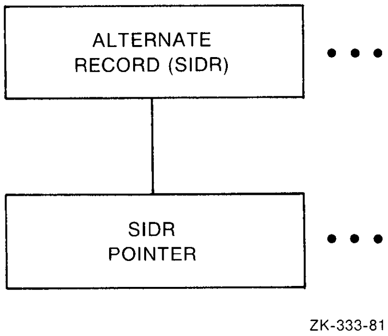
Figure 10–6 Structure of Primary Records



Example 10–6 shows an examination of a primary record.

Figure 10–7 displays the structure of the alternate records.

Figure 10–7 Structure of Alternate Records



Example 10–6 Examining a Primary Record

```
       PRIMARY DATA RECORD (VBN 4, offset %X'000E')
              Record Control Flags:
                     (2)  IRC$V_DELETED    0
                     (3)  IRC$V_RRV        0
                     (4)  IRC$V_NOPTRSZ    0
              Record ID: 1
              RRV ID: 1, 4-Byte Bucket Pointer: 4
              Key:
                      7  6  5  4  3  2  1  0        01234567
                     ------------------------       --------
                            31 30 30 30 30 30|  0000    |000001  |

ANALYZE> DOWN BYTES
        7  6  5  4  3  2  1  0        01234567
       ------------------------       --------
        31 30 30 30 30 30 00 49|  0000   |I.000001|
        20 4C 41 54 49 47 49 44|  0008   |DIGITAL |
        4E 45 4D 50 49 55 51 45|  0010   |EQUIPMEN|
        52 4F 50 52 4F 43 20 54|  0018   |T CORPOR|
        31 31 20 4E 4F 49 54 41|  0020   |ATION 11|
        42 20 54 49 50 53 20 30|  0028   |0 SPIT B|
        41 4F 52 20 4B 4F 4F 52|  0030   |ROOK ROA|
        41 55 48 53 41 4E 20 44|  0038   |D NASHUA|
        33 30 48 4E 20 20 20 20|  0040   |    NH03|
                       31 36 30|  0048   |061     |

ANALYZE> UP
       PRIMARY DATA RECORD (VBN 4, offset %X'000E')
              Record Control Flags:
                     (2)  IRC$V_DELETED    0
                     (3)  IRC$V_RRV        0
                     (4)  IRC$V_NOPTRSZ    0
              Record ID: 1
              RRV ID: 1, 4-Byte Bucket Pointer: 4
              Key:
                      7  6  5  4  3  2  1  0        01234567
                     ------------------------       --------
                            31 30 30 30 30 30|  0000    |000001  |

ANALYZE> DOWN RRV
BUCKET HEADER (VBN 4)
       Check Character: %X'00'
       Area Number: 0
       VBN Sample: 4
       Free Space Offset: %X'0104'
       Free Record ID Range: 4 - 255
       Next Bucket VBN: 4
       Level: 0
       Bucket Header Flags:
              (0)  BKT$V_LASTBKT    1
              (1)  BKT$V_ROOTBKT    0
```

Example 10–7 shows an examination of an alternate record.

Example 10–7 Examining an Alternate Record

```
ANALYZE> DOWN
SIDR RECORD (VBN 6, offset %X'000E')
       Control Flags:
              (4)  IRC$V_NOPTRSZ    0
       Record ID: 1
       Key:
                7  6  5  4  3  2  1  0          01234567
              ------------------------          --------
                        31 36 30 33 30|  0000  |03061    |

ANALYZE> DOWN
              sidr pointer control flags:
                     (2)  IRC$V_DELETED    0
                     (5)  IRC$V_KEYDELETE  0
              sidr pointer record id: 1, 4-byte record VBN: 4
```

10.2 Generating an FDL File from a Data File

You can use the Analyze/RMS_File Utility to create an FDL file generally called an *analysis file*. FDL files created by ANALYZE/RMS_FILE contain statistics about each area and key in the primary sections named ANALYSIS_OF_AREA and ANALYSIS_OF_KEY.

These analysis sections are then used by the Edit/FDL Utility in its Optimize script. You can compare the statistics in these sections with your assumptions about the file's use; you may find some places in the file's structure where additional tuning will be possible.

To generate an FDL file from a data file, use the following command syntax:

ANALYZE/RMS_FILE/FDL filespec

With a command of this type, the FDL file obtains its file name from the input file specification; to assign a different file name, use the /OUTPUT qualifier. For example, the following command would generate an FDL file named INDEXDEF.FDL from the data file CUSTFILE.DAT:

```
$ ANALYZE/RMS_FILE/FDL/OUTPUT=INDEXDEF CUSTFILE.DAT
```

Example 10–8 illustrates an FDL file showing the KEY and ANALYSIS_OF_KEY sections for an indexed file with two keys.

Example 10–8 KEY and ANALYSIS_OF_KEY Sections in an FDL File

```
IDENT        2-JUN-1985 16:15:35       VMS ANALYZE/RMS_FILE Utility

SYSTEM
       SOURCE                  VMS
FILE
       ALLOCATION              9
       BEST_TRY_CONTIGUOUS     no
       BUCKET_SIZE             1
       CONTIGUOUS              no
       EXTENSION               0
       GLOBAL_BUFFER_COUNT     0
       NAME                    DISK$USERWORK:[WORK.RMS32]CUSTDATA.DAT;4
       ORGANIZATION            indexed
       OWNER                   [520,50]
       PROTECTION              (system:RWED, owner:RWED, group:RWED, world:)
       READ_CHECK              no
       WRITE_CHECK             no

RECORD
       BLOCK_SPAN              yes
       CARRIAGE_CONTROL        carriage_return
       FORMAT                  variable
       SIZE                    0

AREA 0
       ALLOCATION              9
       BEST_TRY_CONTIGUOUS     no
       BUCKET_SIZE             1
       CONTIGUOUS              no
       EXTENSION               0

KEY 0
       CHANGES                 no
       DATA_AREA               0
       DATA_FILL               100
       DUPLICATES              no
       INDEX_AREA              0
       INDEX_FILL              100
       LEVEL1_INDEX_AREA       0
       NULL_KEY                no
       PROLOG                  1
       SEG0_LENGTH             6
       SEG0_POSITION           0
       TYPE                    string
```

Example 10–8 Cont'd. on next page

Example 10–8 (Cont.) KEY and ANALYSIS_OF_KEY Sections in an FDL File

```
KEY 1
        CHANGES                 no
        DATA_AREA               0
        DATA_FILL               100
        DUPLICATES              yes
        INDEX_AREA              0
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       0
        NULL_KEY                no
        SEG0_LENGTH             5
        SEG0_POSITION           68
        TYPE                    string

ANALYSIS_OF_AREA 0
        RECLAIMED_SPACE         0

ANALYSIS_OF_KEY 0
        DATA_FILL               50
        DATA_RECORD_COUNT       3
        DATA_SPACE_OCCUPIED     1
        DEPTH                   1
        INDEX_FILL              4
        INDEX_SPACE_OCCUPIED    1
        MEAN_DATA_LENGTH        73
        MEAN_INDEX_LENGTH       9

ANALYSIS_OF_KEY 1
        DATA_FILL               14
        DATA_RECORD_COUNT       3
        DATA_SPACE_OCCUPIED     1
        DEPTH                   1
        DUPLICATES_PER_SIDR     1
        INDEX_FILL              4
        INDEX_SPACE_OCCUPIED    1
        MEAN_DATA_LENGTH        19
        MEAN_INDEX_LENGTH       8
```

10.3 Optimizing and Redesigning File Characteristics

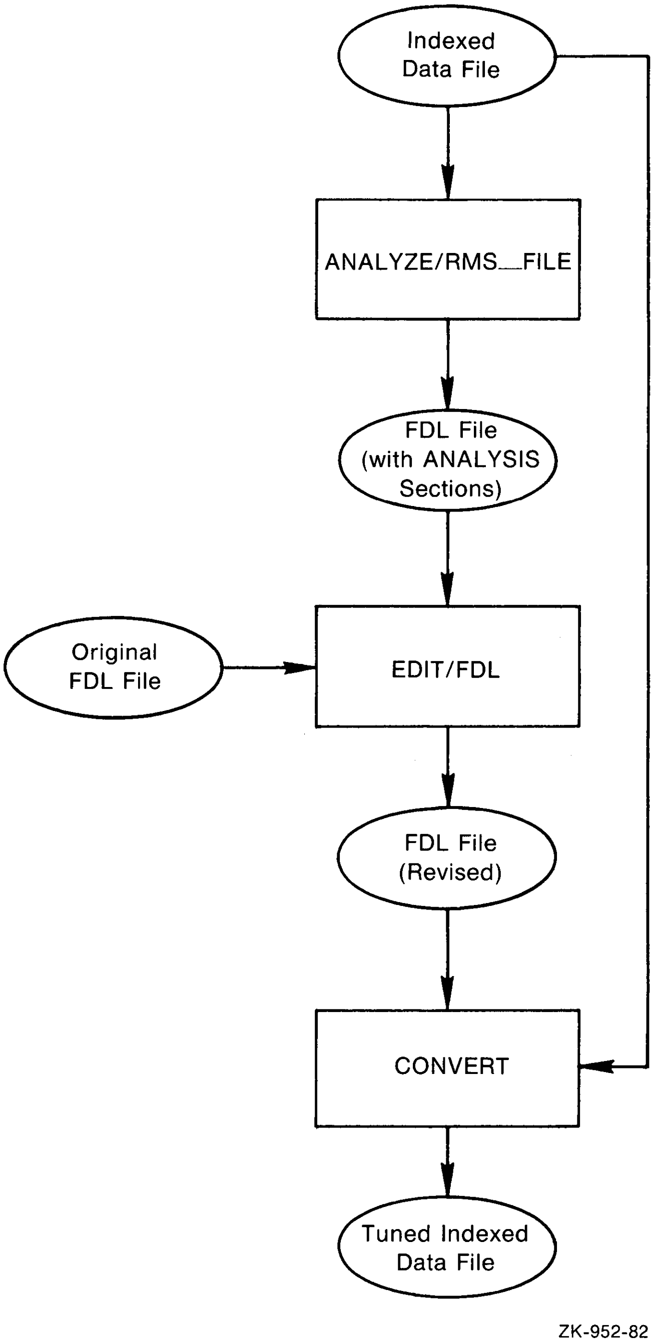
To maintain your files properly, you must occasionally tune them. Tuning involves adjusting and readjusting the characteristics of the file, generally to make the file run faster or more efficiently, and then reorganizing the file to reflect those changes.

There are two ways to tune files. You can redesign your FDL file to change file characteristics or parameters. You can change these characteristics either interactively with EDIT/FDL (the preferred method) or by using a text editor. With the redesigned FDL file, then, you can create a new data file.

You can also optimize your data file by using ANALYZE/RMS_FILE with the /FDL qualifier. This method, rather than actually redesigning your FDL file, produces an FDL file containing certain statistics about the file's use that you can then use to tune your existing data file.

Figure 10–8 shows how to use the VMS RMS utilities to perform the tuning cycle.

Figure 10–8 The VMS RMS Tuning Cycle



Section 10.3.1 describes how to redesign an FDL file, and Section 10.3.2 explains how to optimize the run-time performance of a data file.

10.3.1 Redesigning an FDL File

There are many ways to redesign an FDL file. If you want to make small changes, you can use the ADD, DELETE, and MODIFY commands at the main menu (main editor) level.

Command	Function
ADD	Allows you to add one or more new lines to the FDL file. When you give the ADD command at the main menu level, EDIT/FDL prompts you with a menu displaying all legal primary attributes; your FDL file does not necessarily have to contain all these attributes. You can add a new primary attribute to your file, or you can add a new secondary attribute to an existing primary attribute. When you type in a primary attribute, EDIT/FDL displays all the legal secondary attributes for that primary attribute with their possible values. You can then select the secondary attribute that you want to add to your FDL file and supply the appropriate value for the secondary attribute.
DELETE	Allows you to delete one or more lines from the FDL file. When you give the DELETE command at the main menu level, EDIT/FDL prompts you with a menu displaying the current primary attributes of your FDL file. When you select the primary attribute that has the secondary attribute you want to remove from your current FDL definition, EDIT/FDL displays all the existing secondary attributes of your FDL file with their current values. When you select the secondary attribute, EDIT/FDL removes it from the FDL definition. Also, when you delete the last secondary attribute of a particular primary attribute, EDIT/FDL also removes the primary attribute from the current definition.
MODIFY	Allows you to change an existing line in the FDL definition. When you issue the MODIFY command at the main menu level, EDIT/FDL prompts you with a menu displaying the current primary attributes of your FDL file. When you type in a primary attribute, EDIT/FDL displays all the existing secondary attributes for that primary attribute with their current values. You can then select the secondary attribute of which you want to change the value and the supply the appropriate value for the secondary attribute.

However, if you want to make substantial changes to an FDL file, you should invoke the Touch-up script. Because sequential and relative files are simple in design, the Touch-up script works only with FDL files that describe indexed files. If you want to redesign sequential and relative files, you can use the command listed above (ADD, DELETE, or MODIFY), or you can go through the design phase again, using the scripts for those organizations.

To completely redesign an existing FDL file that describes an indexed sequential file, use the following command syntax:

```
EDIT/FDL/SCRIPT=TOUCHUP fdl-filespec
```

10.3.2 Optimizing a Data File

You can optimize the run-time performance of an existing data file either interactively or noninteractively. In either case, however, the first step is to create an FDL file from the data file by using the Analyze/RMS_File Utility. This analysis produces an FDL file that includes the ANALYSIS_OF_AREA and ANALYSIS_OF_KEY sections.

If you then want to optimize your data file interactively by going through the Optimize script, use the following command syntax:

```
EDIT/FDL/ANALYZE=fdl-filespec/SCRIPT=OPTIMIZE original-fdl-filespec
```

The **fdl-filespec** parameter represents the file specification of the FDL file created with ANALYZE/RMS_FILE. The **original-fdl-filespec** parameter represents an optimized version of the original FDL file.

The rest of the script is similar to an Indexed, Relative, or Sequential script.

If you want to optimize an existing FDL file but do not wish to go through an interactive session, use the following command syntax:

```
EDIT/FDL/ANALYZE=fdl-filespec/NOINTERACTIVE original-fdl-filespec
```

The final stage of optimization is to use the Convert Utility with the old data file and the new FDL file to create a new, optimal data file using the following syntax:

```
CONVERT/FDL=new-fdl-file old-file new-file
```

10.4 Making a File Contiguous

If your file has been used for some time or if it is extremely volatile, the numerous deletions and insertions of records may have caused the optimal design of the file to deteriorate. For example, numerous extensions will degrade performance by causing window-turn operations. In indexed files, deletions can cause empty but unusable buckets to accumulate.

If additions or insertions to a file cause too many extensions, the file's performance will also deteriorate. To improve performance, you could increase the file's window size, but this uses an expensive system resource and at some point may itself hurt performance. A better method is to make the file contiguous again.

This section presents techniques for cleaning up your files. These techniques include using the Copy Utility, the Convert Utility, and the Convert/Reclaim Utility.

10.4.1 Using the Copy Utility

You can use the COPY command with the /CONTIGUOUS qualifier to copy the file, creating a new contiguous version. The /CONTIGUOUS qualifier can be used only on an output file.

To use the COPY command with the /CONTIGUOUS qualifier, use the following command syntax:

```
COPY input-filespec output-filespec/CONTIGUOUS
```

If you do not want to rename the file, use the same name for **input-filespec** and **output-filespec**.

By default, if the input file is contiguous, COPY likewise tries to create a contiguous output file. By using the /CONTIGUOUS qualifier, you ensure that the output file is copied to consecutive physical disk blocks.

The /CONTIGUOUS qualifier can only be used when you copy disk files; it does not apply to tape files. For more information, see the COPY command in the *VMS DCL Dictionary*.

10.4.2 Using the Convert Utility

The Convert Utility can also make a file contiguous if contiguity is an original attribute of the file.

To use the Convert Utility to make a file contiguous, use the following command syntax:

```
CONVERT input-filespec output-filespec
```

If you do not want to rename the file, use the same name for **input-filespec** and **output-filespec**.

10.4.3 Reclaiming Buckets in Prolog 3 Files

If you delete a number of records from a Prolog 3 indexed file, it is possible that you deleted all of the data entries in a particular bucket. VMS RMS generally cannot use such empty buckets to write new records.

With Prolog 3 indexed files, you can reclaim such buckets by using the Convert/Reclaim Utility. This utility allows you to reclaim the buckets without incurring the overhead of reorganizing the file with CONVERT.

As the data buckets are reclaimed, the pointers to them in the index buckets are deleted. If as a result any of the index buckets become empty, they too are reclaimed.

Note that RFA access is retained after bucket reclamation. The only effect that CONVERT/RECLAIM has on a Prolog 3 indexed file is that empty buckets are reclaimed.

To use CONVERT/RECLAIM, use the following command syntax, in which **filespec** specifies a Prolog 3 indexed file:

```
CONVERT/RECLAIM filespec
```

Please note that the file cannot be open for shared access at the time that you give the CONVERT/RECLAIM command.

10.5 Reorganizing a File

Using the Convert Utility is the easiest way to reorganize a file. In addition, CONVERT cleans up split buckets in indexed files. Also, because the file is completely reorganized, buckets in which all the records were deleted will disappear. (Note that this is not the same as bucket reclamation. With CONVERT, the file becomes a new file and records receive new RFAs.)

To use the Convert Utility to reorganize a file, use the following command syntax:

```
CONVERT input-filespec output-filespec
```

If you do not want to rename the file, use the same name for **input-filespec** and **output-filespec**.

10.6 Making Archive Copies

Another part of maintaining files is making sure that the you protect the data in them. You should keep duplicates of your files in another place in case something happens to the originals. In other words, you need to *back up* your files. Then, if something does happen to your original data, you can restore the duplicate files.

The Backup Utility (BACKUP) allows you to create backup copies of files and directories and to restore them, as well. These backup copies are called *save sets,* and they can reside on either disk or magnetic tape. Save sets are also written in BACKUP format; only BACKUP can interpret the data.

Unlike the DCL command COPY, which makes new copies of file (updating the revision dates and assigning protection from the defaults that apply), BACKUP makes copies that are identical in all respects to the originals, including dates and protection.

To use the Backup Utility to create a save set of your file, use the following command syntax:

```
BACKUP input-filespec output-filespec[/SAVE_SET]
```

You have to use the /SAVE_SET qualifier only if the output file will be backed up to disk. You can omit the qualifier for magnetic tape.

For more information about BACKUP, see the description of the Backup Utility in the *VMS Backup Utility Manual.*

A EDIT/FDL Optimization Algorithms

This appendix lists the algorithms used by the Edit/FDL Utility to determine the optimum values for file attributes.

A.1 Allocation

For sequential files with block spanning, EDIT/FDL allocates enough blocks to hold the specified number of records of mean size. If you do not allow block spanning, EDIT/FDL factors in the potential wasted space at the end of each block.

For relative files, EDIT/FDL calculates the total number of buckets in the file and then allocates enough blocks to hold the required number of buckets and associated overhead. EDIT/FDL calculates the total number of buckets by dividing the total number of records in the file by the bucket record capacity. The overhead consists of the prolog which is equal to one cluster.

For indexed files, EDIT/FDL calculates the depth to determine the actual bucket size and number of buckets at each level of the index. It then allocates enough blocks to hold the required number of buckets. Areas for the data level (Level 0) have separate allocations from the areas for the index levels of each key.

In all cases, allocations are rounded up to a multiple of bucket size.

A.2 Extension Size

For sequential files, EDIT/FDL sets the extension size to one-tenth of the allocation size and truncates any fraction. For relative files and indexed files, EDIT/FDL extends the file by 25 percent rounded up to the next multiple of the bucket size.

A.3 Bucket Size

Because most records that EDIT/FDL accesses are close to each other, it makes the buckets large enough to hold 16 records or the total record capacity of the file, whichever is smaller. The maximum bucket size is 63 blocks.

For indexed files, EDIT/FDL permits you to decide the bucket size for any particular index. The data and index levels get the same bucket size but you can use the MODIFY command to change these values.

EDIT/FDL calculates the default bucket size by first finding the most common index depth produced by the various bucket sizes. If you specify smaller buffers rather than fewer levels, EDIT/FDL establishes the default bucket size as the smallest size needed to produce the most common depth. On Surface_Plot graphs, these values are shown on the leftmost edge of each bucket size.

Note: **If you specify a separate bucket size for the Level 1 index, it should match the bucket size assigned to the rest of the index.**

The bucket size is always a multiple of disk cluster size. The ANALYZE/RMS_FILE primary attribute ANALYSIS_OF_KEY now has a new secondary attribute called LEVEL1_RECORD_COUNT that represents the index level immediately above the data. It makes the tuning algorithm more accurate when duplicate key values are specified.

A.4 Global Buffers

The global buffer count is the number of I/O buffers that two or more processes can access. This algorithm tries to cache or "map" the whole Key 0 index (at least up to a point) into memory for quicker and more efficient access.

A.5 Index Depth

The indexed design routines simulate the loading of data buckets with records based on your data regarding key sizes, key positions, record sizes (mean and maximum), compression values, load method, and fill factors.

When EDIT/FDL finds the number of required data buckets, it can determine the actual number of index records in the next level up (each of which points to a data bucket). The process is repeated until all the required index records for a level can fit in one bucket, the root bucket. When a file exceeds 32 levels, EDIT/FDL issues an error message.

With a line_plot, the design calculations are performed up to 63 times—once for each legal bucket size. With a surface_plot, each line of the plot is equivalent to a line_plot with a different value for the variable on the Y-axis.

GLOSSARY

accessor: A process that accesses a file or a record stream that accesses a record.

alternate key: An optional key within the data records in an indexed file; used by VMS RMS to build an alternate index. See also *key (indexed files)* and *primary key*.

area: Areas are VMS RMS-maintained regions of an indexed file. They allow you to specify placement or specific bucket sizes, or both, for particular portions of a file. An area consists of any number of buckets, and there may be from 1 to 255 areas in a file.

asynchronous record operation: An operation in which your program may possibly regain control before the completion of a record retrieval or storage request. Completion ASTs and the Wait service are the mechanisms provided by VMS RMS for programs to synchronize with asynchronous record operations. See also *synchronous record operation*.

bits per inch: The recording density of a magnetic tape. Indicates how many characters can fit on one inch of the recording surface. See also *density*.

block: The smallest number of consecutive bytes that VMS RMS transfers during read and write operations. A block is 512 8-bit bytes on a Files–11 On-Disk Structure disk; on magnetic tape, a block may be anywhere from 8 to 8192 bytes.

block I/O: The set of VMS RMS procedures that allow you direct access to the blocks of a file regardless of file organization.

block spanning: In a sequential file, the option for records to cross block boundaries.

bootstrap block: A block in the index file of a system disk. Can contain a program that loads the operating system into memory.

bpi: See also *bits per inch*.

bucket: A storage structure, consisting of 1 to 32 blocks, used for building and processing relative and indexed files. A bucket contains one or more records or record cells. Buckets are the units of contiguous transfer between VMS RMS buffers and the disk.

bucket split: The result of inserting records into a full bucket. To minimize bucket splits, VMS RMS attempts to keep half of the records in the original bucket and transfer the remaining records to a newly created bucket.

buffer: A memory area used to temporarily store data. Buffers are generally categorized as being either user buffers or I/O buffers.

cluster: The basic unit of space allocation on a Files–11 On-Disk Structure volume. Consists of one or more contiguous blocks, with the number being specified when the volume is initialized.

contiguous area: A group of physically adjacent blocks.

count field: A field prefixed to a variable-length record that specifies the number bytes in the record.

cylinder: The tracks at the same radius on all recording surfaces of a disk.

density: The number of bits per inch (bpi) of magnetic tape. Typical values are 800 bpi and 1600 bpi. See also *bits per inch.*

directory: A file used to locate files on a volume. A directory file contains a list of files and their unique internal identifications.

directory tree: The subdirectories created beneath a directory and the subdirectories within the subdirectories (and so forth).

extent: One or more adjacent clusters allocated to a file or to a portion of a file.

FDL: See *File Definition Language*

file: An organized collection of related items (records) maintained in an accessible storage area, such as disk or tape.

File Definition Language: A special-purpose language used to write file creation and run-time specifications for data files. These specifications are written in text files called FDL files; they are then used by the VMS RMS utilities and library routines to create the actual data files.

file header: A block in the index file describing a file on a Files–11 On-Disk Structure disk, including the location of the file's extents. There is at least one file header for every file on the disk.

file organization: The physical arrangement of data in the file. You select the specific organization from those offered by VMS RMS, based on your individual needs for efficient data storage and retrieval. See also *indexed file organization, relative file organization,* and *sequential file organization.*

Files–11 On-Disk Structure: The standard physical disk structure used by VMS RMS.

fixed-length control field: A fixed-size area, prefixed to a VFC record, containing additional information that can be processed separately and that may have no direct relationship to the other contents of the record. For example, the fixed-length control field might contain line sequence numbers for use in editing operations.

fixed-length record format: Property of a file in which all records are the same length. This format provides simplicity in determining the exact location of a record in the file and eliminates the need to prefix a record size field to each record.

global buffer: A buffer that many processes share.

home block: A block in the index file, normally next to the bootstrap block, that identifies the volume as a Files–11 On-Disk Structure volume and provides specific information about the volume, such as volume label and protection.

index: The structure that allows retrieval of records in an indexed file by key value. See also *key (indexed files).*

index file: A file on each Files–11 On-Disk Structure volume that provides the means for identification and initial access to the volume. Contains the access information for all files (including itself) on the volume: bootstrap block, home block, file headers.

indexed file organization: A file organization that allows random retrieval of records by key value and sequential retrieval of records in sorted order by key value. See also *key (indexed files).*

interrecord gap (IRG): An interval of blank space between data records on the recording surface of a magnetic tape. The IRG enables the tape unit to decelerate, stop if necessary, and accelerate between record operations.

I/O buffer: A buffer used for performing input/output operations.

IRG: See *interrecord gap.*

key (indexed file): A character string, a packed decimal number, a 2- or 4-byte unsigned binary number, or a 2- or 4-byte signed integer within each data record in an indexed file. You define the length and location within the records; VMS RMS uses the key to build an index. See also *primary key, alternate key,* and *random access by key value.*

key (relative file): The relative record number of each data record cell in a data file; VMS RMS uses the relative record numbers to identify and access data records in a relative file in random access mode. See also *relative record number.*

local buffer: A buffer that is dedicated to one process.

locate mode: Technique used for a record input operation in which the data records are not copied from the VMS RMS I/O buffer, but a pointer is returned to the record in the VMS RMS I/O buffer. See also *move mode.*

move mode: Technique used for a record transfer in which the data records are copied between the I/O buffer and your program buffer for calculations or operations on the record. See also *locate mode.*

multiblock: An I/O unit that includes up to 127 blocks. Use is restricted to sequential files.

multiple-extent file: A disk file having two or more extents.

native mode: The processor's primary execution mode in which the programmed instructions are interpreted as byte-aligned, variable-length instructions that operate on the following data types: byte, word, longword, and quadword integers; floating and double floating character strings; packed decimals; and variable-length bit fields. The other instruction execution mode is compatibility mode.

primary key: The mandatory key within the data records of an indexed file; used by VMS RMS to determine the placement of records within the file and to build the primary index. See also *key (indexed files)* and *alternate key.*

random access by key (indexed file): Retrieval of a data record in an indexed file by either a primary or alternate key within the data record. See also *key (indexed files).*

random access by key (relative file): Retrieval of a data record in a relative file by the relative record number of the record. See also *key (relative files).*

random access by record file address (RFA): Retrieval of a record by the record's unique address, which VMS RMS returns to you. This record access mode is the only means of randomly accessing a sequential file containing variable-length records.

random access by relative record number: Retrieval of a record by its relative record number. For relative files and sequential files (on disk devices) that contain fixed-length records, random access by relative record number is synonymous with random access by key. See also *random access by key (relative files only)* and *relative record number*.

read-ahead processing: A software option used for sequentially accessing sequential files using two buffers. One buffer holds records to be read from the disk. The other buffer awaits I/O completion.

record: A set of related data that your program treats as a unit.

record access mode: The manner in which VMS RMS retrieves or stores records in a file. Available record access modes are determined by the file organization and specified by your program.

record access mode switching: Term applied to the switching from one type of record access mode to another while processing a file.

record blocking: The technique of grouping multiple records into a single block. On magnetic tape, an IRG is placed after the block rather than after each record. This technique reduces the number of I/O transfers required to read or write the data, and, in addition (for magnetic tape), it increases the amount of usable storage area. Record blocking also applies to disk files.

record cell: A fixed-length area in a relative file that can contain a record. Fixed-length record cells permit VMS RMS to directly calculate the record's actual position in the file.

record file address (RFA): The unique address VMS RMS returns to your program whenever it accesses a record. Using the RFA, your program can access disk records randomly regardless of file organization. The RFA is valid only for the life of the file, and when an indexed file is reorganized, each record's RFA will typically change.

record format: The way a record physically appears on the recording surface of the storage medium. The record format defines the method for determining record length.

record length: The size of a record in bytes.

record locking: A facility that prevents access to a record by more than one record stream or process until the initiating record stream or process releases the record.

Record Management Services: See VMS RMS (Record Management Services)

record stream: The access environment for reading, writing, deleting and updating records.

relative file organization: The arrangement of records in a file in which each record occupies a cell of equal length within a bucket. Each cell is assigned a successive number, called a relative record number, which represents the cell's position relative to the beginning of the file.

relative record number: An identification number used to specify the position of a record cell relative to the beginning of the file; used as the key during random access by key mode to relative files.

reorganization: A record-by-record copy of an indexed file to another indexed file with the same key attributes as the input file.

RFA: See *record file address.*

RMS: See *VMS RMS*

RMS–11: A set of routines that are linked with compatibilty mode and PDP–11 programs and provide similar features for VMS RMS. The file organizations and record formats used by RMS–11 are very similar to those of VMS RMS; one exception is that RMS–11 does not support Prolog 3 indexed files, which are supported by VMS RMS.

root bucket: The primary routing bucket for an index; geometrically, the top of the index tree. When a key search begins, VMS RMS goes first to the index root bucket to determine which bucket, at the next lower level, is the next link in the bucket chain.

sequential file organization: The arrangement of records in a file in one-after-the-other fashion. Records appear in the order in which they were written.

sequential record access mode: Record storage or retrieval that starts at a designated point in the file and continues in one-after-the-other fashion through the file. That is, records are accessed in the order in which they physically appear in the file.

shared access: A file management technique that allows more than one user to simultaneously access a file or a group of files.

stream: An access window to a file associated with a record access control block (RAB) supporting record operation requests.

stream record format: Property of a file specifying that the data in the file is interpreted as a continuous sequence of bytes, without control information, except for terminators that are recognized as record separators. Stream record format applies to sequential files only.

synchronous record operation: An operation in which your program does not regain control until after the completion of a record retrieval or storage request. See also *asynchronous record operation.*

terminator: Special characters or character sequences used to delimit the records in files using the stream record format.

track: A collection of blocks at a single radius on one recording surface of a disk.

tuning: The process of designing your files to achieve better processing performance.

user buffer: A buffer within an application program.

variable-length record format: Property of a file in which record length may vary.

variable-length with fixed-length control field (VFC) record format: Property of a file in which records of variable-length contain an additional fixed-length control field capable of storing data that may have no connection with the other contents of the record. VFC record format is not applicable to indexed files.

VFC record format: *See variable-length with fixed-length control field (VFC) record format*

VMS RMS (VMS Record Management Services): The file and record access subsystem of the VMS operating system. VMS RMS helps your application program process records within files, thereby allowing interaction between your application program and the data.

volume (disk): An ordered set of 512-byte blocks. The medium that carries Files–11 On-Disk Structure files.

volume (magnetic tape): A reel of magnetic tape, which may contain a part of a file, a complete file, or more than one file.

volume set: A collection of related volumes.

write-behind processing: A software option used for sequentially accessing sequential files using two buffers. One buffer holds records to be written to the disk. The other buffer awaits I/O completion.

Index

Reader's Comments

Guide to VMS File Applications
AA–LA78A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	□	□	□	□
Completeness (enough information)	□	□	□	□
Clarity (easy to understand)	□	□	□	□
Organization (structure of subject matter)	□	□	□	□
Figures (useful)	□	□	□	□
Examples (useful)	□	□	□	□
Index (ability to find topic)	□	□	□	□
Page layout (easy to find information)	□	□	□	□

I would like to see more/less ______________________________

What I like best about this manual is ______________________________

What I like least about this manual is ______________________________

I found the following errors in this manual:

Page ______ Description ______________________________

Additional comments or suggestions to improve this manual:

I am using **Version** ______ of the software this manual describes.

Name/Title ______________________________ Dept. ______________

Company ______________________________ Date ______________

Mailing Address ______________________________

______________________________ Phone ______________

-- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here

Cut Along Dotted Line