RSX–11M/M–PLUS
RMS–11 User's Guide

Order No. AA-L669A-TC

digital software

RSX-11M/M-PLUS
RMS-11 User's Guide

Order No. AA-L669A-TC

April 1983

This manual provides information on file and task design using RMS-11. The information includes design considerations for writing application programs in both MACRO-11 and high-level languages.

SUPERSESSION/UPDATE INFORMATION:	This revised document supersedes the *RMS-11 User's Guide* (Order No. AA-D538A-TC).
OPERATING SYSTEM AND VERSION:	RSX-11M Version 4.1, RSX-11M-PLUS Version 2.1
SOFTWARE VERSION:	RMS-11 Version 2.0

digital equipment corporation · maynard, massachusetts

First Printing, April 1983

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.

Copyright © 1983 by Digital Equipment Corporation
All Rights Reserved.

Printed in U.S.A.

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

ZK2168

HOW TO ORDER ADDITIONAL DOCUMENTATION

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

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

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

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

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

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

CONTENTS

Page

PREFACE

MANUAL OBJECTIVES ix
INTENDED AUDIENCE ix
STRUCTURE OF THIS DOCUMENT ix
ASSOCIATED DOCUMENTS x
CONVENTIONS USED IN THIS DOCUMENT x

SUMMARY OF TECHNICAL CHANGES xiii

CHAPTER 1 RMS-11 CONCEPTS AND PROCESSING ENVIRONMENT

1.1 CONCEPTS OF DATA ORGANIZATION AND ACCESS 1-1
1.1.1 Records 1-1
1.1.2 Files 1-2
1.1.3 Access 1-7
1.1.4 Processing 1-10
1.1.5 File Maintenance 1-10
1.2 RMS-11 IMPLEMENTATION OF DATA ORGANIZATION AND ACCESS 1-11
1.2.1 RMS-11 Record Formats 1-11
1.2.2 RMS-11 File Organizations 1-11
1.2.3 RMS-11 Record Access Modes 1-12
1.2.4 RMS-11 Utilities 1-12
1.3 RMS-11 PROCESSING ENVIRONMENT 1-12
1.3.1 RMS-11 Task Structure 1-13
1.3.2 RMS-11 Record Processing 1-15
1.3.3 RMS-11 File Processing 1-16
1.4 FILE ATTRIBUTES 1-18
1.5 PROCESSING BY BLOCK ACCESS 1-20

CHAPTER 2 APPLICATION DESIGN

2.1 WHEN TO DESIGN 2-2
2.2 DESIGN CONSIDERATIONS 2-3
2.2.1 Speed 2-3
2.2.2 Space 2-4
2.2.2.1 Data Storage 2-5
2.2.2.2 Task Size 2-5
2.2.2.3 Buffer Sizes 2-5
2.2.3 Shared Access 2-5
2.2.3.1 Bucket Locking 2-8
2.2.3.2 Sharing among Access Streams 2-9
2.2.3.3 Programming Considerations 2-10
2.2.4 Ease of Design 2-10
2.3 DESIGN PROCESS 2-11
2.4 SELECTING A FILE ORGANIZATION 2-11
2.4.1 Record Formats 2-15
2.4.1.1 Fixed-Length Format 2-15
2.4.1.2 Variable-Length Format 2-15

2.4.1.3 Variable-with-Fixed-Control Format 2-16
2.4.1.4 Stream Format 2-16
2.4.1.5 Undefined Format 2-17
2.4.2 I/O Techniques 2-17

CHAPTER 3 SEQUENTIAL FILE APPLICATIONS

3.1 FILE STRUCTURE 3-1
3.2 RECORD SIZE 3-2
3.3 FILE DESIGN 3-3
3.3.1 Data Storage Medium 3-3
3.3.2 File Allocation 3-4
3.3.2.1 Initial Allocation 3-4
3.3.2.2 Default Extension Quantity 3-4
3.3.3 Contiguity 3-5
3.4 ACCESS SHARING 3-5
3.4.1 Record Access to Sequential Files 3-6
3.4.2 Block Access to Sequential Files 3-6
3.5 RECORD AND FILE PROCESSING OF SEQUENTIAL FILES . . 3-6
3.5.1 Record and Stream Operations 3-7
3.5.1.1 CONNECT 3-7
3.5.1.2 DISCONNECT 3-7
3.5.1.3 FIND 3-7
3.5.1.4 FLUSH 3-9
3.5.1.5 GET 3-9
3.5.1.6 PUT 3-10
3.5.1.7 REWIND 3-11
3.5.1.8 TRUNCATE 3-12
3.5.1.9 UPDATE 3-12
3.5.2 Record Transfer Modes 3-13
3.5.2.1 Move Mode 3-13
3.5.2.2 Locate Mode 3-14
3.5.3 I/O Techniques 3-14
3.5.3.1 Asynchronous Record Operations 3-14
3.5.3.2 Deferred Write 3-14
3.5.3.3 Multiple Buffers 3-14
3.5.3.4 Multiple Access Streams 3-15
3.5.3.5 Multiblock Count 3-15
3.5.4 File and Directory Operations 3-15

CHAPTER 4 RELATIVE FILE APPLICATIONS

4.1 FILE STRUCTURE 4-1
4.2 RECORD SIZE 4-2
4.3 FILE DESIGN 4-2
4.3.1 Bucket Size 4-2
4.3.2 File Allocation 4-3
4.3.2.1 Initial Allocation 4-3
4.3.2.2 Default Extension Quantity 4-4
4.3.3 Contiguity 4-4
4.3.4 Maximum Record Number 4-5
4.4 ACCESS SHARING 4-6
4.4.1 Record Access to Relative Files 4-6
4.4.2 Block Access to Relative Files 4-6
4.5 RECORD AND FILE PROCESSING OF RELATIVE FILES . . . 4-6
4.5.1 Record and Stream Operations 4-7
4.5.1.1 CONNECT 4-7
4.5.1.2 DELETE 4-7
4.5.1.3 DISCONNECT 4-8
4.5.1.4 FIND 4-8
4.5.1.5 FLUSH 4-10
4.5.1.6 GET 4-10
4.5.1.7 PUT 4-11
4.5.1.8 REWIND 4-12

4.5.1.9 UPDATE 4-12
4.5.2 Record Transfer Modes 4-12
4.5.2.1 Move Mode 4-12
4.5.2.2 Locate Mode 4-13
4.5.3 I/O Techniques 4-14
4.5.3.1 Asynchronous Record Operations 4-14
4.5.3.2 Deferred Write 4-14
4.5.3.3 Multiple Buffers 4-15
4.5.3.4 Multiple Access Streams 4-15
4.5.4 File and Directory Operations 4-15

CHAPTER 5 INDEXED FILE STRUCTURE AND ACCESS

5.1 PHYSICAL FILE STRUCTURE 5-2
5.2 CONCEPTUAL FILE STRUCTURE 5-4
5.2.1 Data 5-5
5.2.1.1 Level 0 of the Primary Index 5-5
5.2.1.2 Level 0 of an Alternate Index 5-5
5.2.2 Indexes 5-6
5.2.3 Random Access Using the RMS-11 Indexed File Structure 5-7
5.2.4 Why this Structure? 5-8
5.3 PROCEDURES FOR PERFORMING RANDOM RECORD OPERATIONS 5-9
5.3.1 Writing a Record 5-10
5.3.1.1 Simplest Case 5-10
5.3.1.2 Bucket Splitting 5-11
5.3.1.3 Incremental Reorganization 5-12
5.3.2 Getting and/or Finding a Record 5-13
5.3.3 Updating a Record 5-14
5.3.4 Deleting a Record 5-15
5.4 PROCEDURES FOR PERFORMING SEQUENTIAL RECORD OPERATIONS 5-16
5.5 I/O COST OF PERFORMING RECORD OPERATIONS 5-17

CHAPTER 6 INDEXED FILE DESIGN

6.1 RECORD SIZE 6-1
6.2 KEY SELECTION 6-2
6.2.1 Number of Keys 6-2
6.2.2 Key Data Types 6-3
6.2.2.1 String Type 6-3
6.2.2.2 Two-Byte Signed Integer Type 6-4
6.2.2.3 Four-Byte Signed Integer Type 6-4
6.2.2.4 Two-Byte Unsigned Binary Type 6-5
6.2.2.5 Four-Byte Unsigned Binary Type 6-5
6.2.2.6 Packed Decimal Type 6-6
6.2.3 Key Size 6-6
6.2.4 Position of Key in Record 6-7
6.2.5 Key Characteristics 6-8
6.2.5.1 Duplicates 6-8
6.2.5.2 Changes 6-9
6.2.5.3 Null Key 6-10
6.3 AREAS 6-10
6.4 PLACEMENT CONTROL 6-13
6.5 BUCKET SIZE 6-15
6.5.1 Bucket Size for Primary Index 6-16
6.5.2 Bucket Sizes for Alternate Indexes 6-19
6.5.3 Program Syntax 6-21
6.6 FILE ALLOCATION 6-22
6.6.1 Initial Allocation 6-22
6.6.2 Default Extension Quantity 6-26
6.7 POPULATION TECHNIQUES 6-26
6.7.1 Ascending Order by Primary Key 6-27

6.7.2 Random Insertions after File Population . . . 6-28
6.7.2.1 Bucket Fill Size 6-28
6.7.2.2 Mass Insertion 6-29

CHAPTER 7 RECORD AND FILE PROCESSING OF INDEXED FILES

7.1 ACCESS SHARING 7-1
7.1.1 Record Access to Indexed Files 7-1
7.1.2 Block Access to Indexed Files 7-1
7.2 RECORD AND STREAM OPERATIONS 7-2
7.2.1 CONNECT 7-2
7.2.2 DELETE 7-3
7.2.3 DISCONNECT 7-3
7.2.4 FIND 7-3
7.2.5 FLUSH 7-5
7.2.6 GET 7-5
7.2.7 PUT 7-5
7.2.8 REWIND 7-6
7.2.9 UPDATE 7-6
7.3 RECORD TRANSFER MODES 7-6
7.3.1 Move Mode 7-7
7.3.2 Locate Mode 7-7
7.4 I/O TECHNIQUES 7-8
7.4.1 Asynchronous Record Operations 7-8
7.4.2 Deferred Write 7-8
7.4.3 Multiple Buffers 7-9
7.4.4 Multiple Access Streams 7-10
7.4.5 Sequentially Reading Write-Shared Files . . . 7-10
7.5 FILE AND DIRECTORY OPERATIONS 7-10

CHAPTER 8 TASK BUILDING AND COMMON OPTIMIZATION TECHNIQUES

8.1 TASK BUILDING WITH RMS-11 ROUTINES 8-1
8.1.1 Disk-Resident Overlays 8-3
8.1.1.1 ODL Files 8-5
8.1.2 Memory-Resident Overlays 8-6
8.1.2.1 Task Building against the RMS-11 Resident Library 8-6
8.1.2.2 Using RMS-11 Operations from within Your Own Resident Library 8-8
8.1.2.3 Deciding Between Types of Overlays 8-9
8.2 PROGRAM DEVELOPMENT 8-9
8.2.1 Flow of Operations Should Reflect RMS-11 Code Structure 8-10
8.2.2 Task Builder Considerations 8-10
8.3 VIRTUAL-TO-LOGICAL-BLOCK MAPPING 8-11
8.3.1 Retrieval Pointers on Disk 8-11
8.3.2 Retrieval Pointers in Memory 8-11
8.3.3 Optimizing Window Turning 8-12
8.4 OTHER OPTIMIZATIONS 8-13
8.4.1 Allocating More Resources to the Task 8-13
8.4.2 Disk Usage 8-13

APPENDIX A FILE SPECIFICATION PARSING

A.1 STANDARD FILE SPECIFICATION SYNTAX A-1
A.1.1 Device A-1
A.1.2 Directory A-1
A.1.3 Name A-2
A.1.4 Type A-3
A.1.5 Version A-3
A.2 ANSI MAGNETIC TAPE FILE SPECIFICATION SYNTAX . . . A-4
A.2.1 Device A-4

A.2.2 Directory A-4
A.2.3 Quoted String A-5
A.2.4 Version A-5
A.3 GENERATION OF A FULL FILE SPECIFICATION A-5

APPENDIX B REMOTE FILE AND RECORD ACCESS VIA DECNET

B.1 REMOTE NODE SPECIFICATION B-2
B.2 REMOTE ACCESS ENVIRONMENTS B-3
B.3 REMOTE ACCESS POOL CONSIDERATIONS B-3

INDEX

FIGURES

FIGURE 1-1 Record Formats 1-2
1-2 Files 1-3
1-3 Sequential File Organization 1-4
1-4 Relative File Organization 1-5
1-5 Indexed File Organization 1-6
1-6 Indexed File Example 1-6
1-7 Record Access Modes 1-8
1-8 RMS-11 Task Structure 1-14
1-9 Records Spanning Blocks 1-17
2-1 Time Factors in an I/O Operation 2-4
2-2 System Protection Concepts 2-6
2-3 Bucket Locking Example 2-9
2-4 Count Field on Disk and Tape 2-15
3-1 RMS-11 Task Structure 3-13
4-1 RMS-11 Task Structure 4-13
5-1 Indexed File with and without Areas 5-3
5-2 Formatted Bucket 5-4
5-3 Index as a Pyramid 5-4
5-4 Format for Secondary Index Data Record 5-6
5-5 Example of a Primary Index 5-7
5-6 Search Time Curves 5-9
6-1 Single-Area Indexed File 6-11
6-2 Example of Single-Area Indexed File 6-11
6-3 Two-Area Indexed File 6-12
6-4 Example of Multi-Area Indexed File 6-13
7-1 RMS-11 Task Structure 7-7
8-1 Source-to-Task Sequence 8-2
8-2 RMS-11 Tasks 8-4

TABLES

TABLE 1-1 Record Formats and File Organizations 1-20
2-1 File Organization Characteristics and Capabilities 2-12
2-2 File Organization Advantages and Disadvantages . 2-13
3-1 End-of-Block Indicators 3-2
3-2 Sequential File Data Sizes (in bytes) 3-3
4-1 Relative File Data Sizes (in bytes) 4-2
5-1 I/O Cost of Performing Record Operations 5-18
6-1 Key Data Types 6-6

PREFACE

MANUAL OBJECTIVES

This document is a guide to using RMS-11 capabilities and operations in file and task design for application programs written in either MACRO-11 or high-level languages.

INTENDED AUDIENCE

This document is intended for application programmers who want to achieve optimal performance with new applications they are writing or with existing applications.

NOTE

Only MACRO-11 programmers can use the full set of RMS-11 capabilities. Subsets of these capabilities are available to high-level language programmers. See your high-level language documentation to determine:

- What RMS-11 facilities you can use in your high-level language
- The syntax for using these facilities

STRUCTURE OF THIS DOCUMENT

This manual contains eight chapters and two appendixes:

- Chapter 1, RMS-11 Concepts and Processing Environment, introduces the concepts of data organization and access and the RMS-11 implementation of these concepts.
- Chapter 2, Application Design, presents general considerations that apply to application design and information that will help the application designer select a file organization.
- Chapter 3, Sequential File Applications, discusses sequential file structure, design, and processing.
- Chapter 4, Relative File Applications, discusses relative file structure, design, and processing.

- Chapters 5, Indexed File Structure and Access, 6, Indexed File Design, and 7, Record and File Processing of Indexed Files, discuss indexed file structure, design, and processing.

- Chapter 8, Task Building and Common Optimization Techniques, describes techniques that can be used to optimize application programs that use RMS-11, regardless of the file organization selected.

- Appendix A, File Specification Parsing, documents RMS-11's handling of file specifications.

- Appendix B, Remote File and Record Access via DECnet, briefly describes the remote access environment and remote file specification syntax.

ASSOCIATED DOCUMENTS

In addition to this user's guide, the RMS-11 documentation set contains the following manuals.

RSX-11M/M-PLUS RMS-11: An Introduction presents the major concepts of RMS-11, introduces the RMS-11 operations, and defines key terms required for understanding RMS-11 capabilities and functions. You should read the introduction before proceeding to other manuals in the RMS-11 documentation set.

The RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide is a reference document for MACRO-11 programmers that describes the macros and symbols that make up the interface between a MACRO-11 program or subprogram and the RMS-11 operation routines.

The RSX-11M/M-PLUS RMS-11 Utilities manual is both a user and a reference document for all users, both programmers and nonprogrammers. It describes the RMS-11 utilities that are available for creating and maintaining RMS-11 files.

In addition, the Mini-Reference Insert includes an easy-reference guide for users who are familiar with RMS-11 and its documentation. It summarizes the RMS-11 utilities and error codes.

CONVENTIONS USED IN THIS DOCUMENT

Convention	Meaning
UPPERCASE	Uppercase words and letters, used in format examples, indicate that you should type the word or letter exactly as shown.
lowercase	Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice.
quotation marks	The term "quotation marks" refers to double quotation marks (").
apostrophes	The term "apostrophe" refers to a single quotation mark (').

[ ]	Square brackets indicate that the enclosed item is optional.
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated one or more times. For example: file-spec[,file-spec...]
: :	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.
TKB>//	In examples of commands you enter and system responses, all output lines and prompting characters that the system prints or displays are shown in black letters. All the lines you type are shown in red letters.

Unless otherwise noted, all numeric values are represented in decimal notation.

Unless otherwise specified, you terminate commands by pressing the RETURN key.

SUMMARY OF TECHNICAL CHANGES

RMS-11 Version 2.0 supports random access to fixed-format disk sequential files and sequential block access to disk files of any format and organization.

The RMS-11 Version 2.0 resident libraries are task independent. This means that once a program is linked with this library, the library can be rebuilt or replaced without requiring that the task linked to it be rebuilt.

RMS-11 Version 2.0 contains no library equivalent to the RMSSEQ memory-resident library included with RMS-11 Version 1.8. The RMSRES resident library or the disk-resident ODL files can be used to obtain equivalent functionality and performance.

New versions of the RMS-11 Version 1.8 ODL files are provided. These ODL files are: RMS11S.ODL, RMS11X.ODL, RMS12X.ODL, and RMS11.ODL. The Version 1.8 ODL files will still work with Version 2.0, but the new versions will be more efficient. RMS-11 V1.8 ODL structures other than RMS11S.ODL, RMS11X.ODL, and RMS12X.ODL may not work correctly with the RMS-11 V2.0 code; when in doubt, verify them by comparison with the V2.0 RMS11.ODL file. In addition, two new ODL files are provided with Version 2.0: RMS12S.ODL and DAP11X.ODL.

Files with stream and VFC records can now be created on unit-record devices to avoid the need for special-case code in copy-type operations.

- For VFC files, the record header is thrown away on output unless the file is a "print format" file.
- For stream files, if none of the 3 carriage control bits is set (print file format, carriage control, or FORTRAN carriage control), and if the last character is not a linefeed, formfeed, or vertical tab, the carriage-return/linefeed (CR/LF) is appended at the end of the record.
- For stream files, if either the carriage control or FORTRAN carriage control attribute is set, and if the last 2 characters of the record are CR/LF, the trailing CR/LF is stripped off and then definition of the carriage control attribute (CR or FTN) is applied.

For similar ease-of-copying reasons, RMS-11 now allows creation of relative and indexed files for output to nondisk devices (for magtape, however, the record format must be variable length or fixed length).

The RMS-11 File Design Utility (RMSDES) is a new utility that allows you to design and create files interactively. It is fully documented in the RSX-11M/M-PLUS RMS-11 Utilities manual.

RMS-11 Version 2.0 supports five new directory operations: $ENTER, $PARSE, $REMOVE, $RENAME, and $SEARCH. These operations are fully documented in the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide.

RMS-11 Version 2.0 supports a new wildcard file specification facility and a new print-record output handling format. These are also fully documented in the macro programmer's guide.

User-provided interlocks allow a special, limited form of sequential file sharing among a group of accessors that includes at most one read/write accessor and any number of read-only accessors.

If suitable DECnet facilities exist on your system and on the target system, RMS-11 Version 2.0 will allow file and record access to files on remote network nodes, if those nodes include an RMS-11-based file access listener (FAL).

For magtape, RMS-11 now allows fixed-format records to be less than 18 bytes.

Files with stream or VFC records can now be created on unit-record devices. In addition, RMS-11 now allows the creation of relative and indexed files for output to nondisk devices, although they will be treated as sequential files.

<CTRL/Z> and <ESC> are no longer recognized as record terminators for stream files, and <CTRL/Z> is no longer recognized as a file terminator for stream files.

RMS-11 Version 2.0 pads stream files with null characters, to the high block of the file (not just to the end of the current block).

The memory-resident library RMSRES can be clustered with any other resident library that supports clustering.

On RSX-11M-PLUS systems that include hardware support for supervisor mode, RMSRES can also be used in supervisor mode.

On RSX-11M systems, an optional subset library, which contains support for sequential and relative files only, is available.

NOTE

All new RMS-11 features are fully accessible only to MACRO-11 programmers. See your high-level language documentation for supported features.

CHAPTER 1

RMS-11 CONCEPTS AND PROCESSING ENVIRONMENT

Your business, whether commercial, scientific, governmental, or educational, relies on data. That data indicates the current state of your business and helps you control the future of the business. Therefore, you want fast, efficient access to the right data when you need it.

You are familiar with dealing with data on paper and know that records of transactions and reports on your business's activities can occupy a very large number of file folders. You also know that finding exactly the data you need can be a time-consuming process.

Computer hardware, however, with its speed and mass data storage capabilities, provides the means for fast, efficient access to data. Computer software provides the means for translating the data from the format you use to a format the computer system can handle -- and back again.

RMS-11 is such a translater between you and your system. This chapter introduces RMS-11 in terms of general concepts of data organization and access, which apply regardless of whether data is stored on paper or within a computer's memory. It then discusses the RMS-11 implementation of data organization and access, and the RMS-11 data processing environment.

1.1 CONCEPTS OF DATA ORGANIZATION AND ACCESS

This section examines the general concepts of data organization, using images from the noncomputer environment you may be most familiar with.

1.1.1 Records

When data is stored on paper, it is recorded in groups of items whose form is repeated throughout the data. Each group of items is called a record. Within each record are the specific items of data you are concerned with. For example, all the information on an employee constitutes a personnel record; all the information on a stock item constitutes an inventory record.

On paper, a record can be a form; different types of records require different forms. Some forms are always the same length; their information does not expand with time or use. For example, a product information form does not vary in size. If the facts about a product change, you fill out a new form. If a new product is added, you also fill out a new form.

Other forms vary in length with time and use, continuing on to new pages as they grow. For example, an employee with the company for 10 years has more data in his or her personnel record than a new employee.

Other forms might use a combination of these two formats. For example, a record of service on a piece of equipment might begin with control information describing the specific piece of equipment (name, model number, date of installation, and so on) and continue on to new pages documenting the service performed on it.

Figure 1-1 illustrates various record formats.

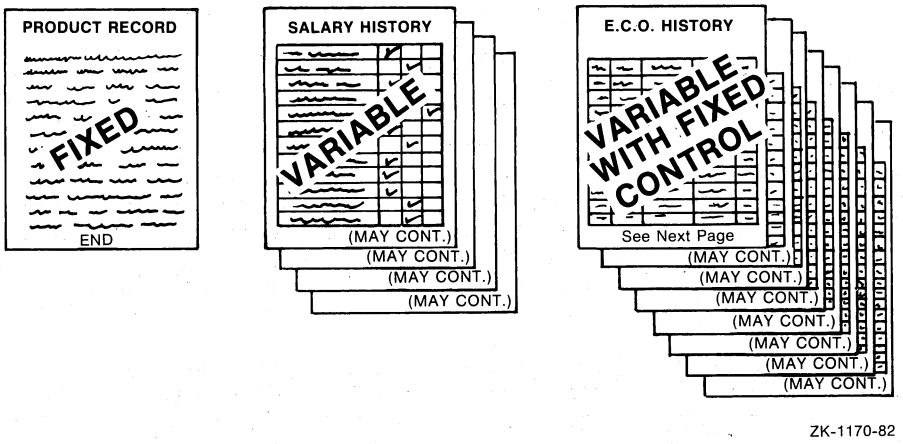


Figure 1-1: Record Formats

1.1.2 Files

When data is stored on paper records, it is usually gathered into files and stored physically in filing cabinets, organized by related records. For example, all employee records might be stored in one file and placed in one drawer of the filing cabinet.

A file not only keeps related data in one place, it also segregates that data from other, unrelated data.

As data grows, the file and storage requirements become more complicated, and the number of filing cabinets multiplies. Then, the files acquire names or numbers, the drawers acquire signs indicating the contents of the drawers and who may use them, and cross-referencing systems are introduced to help locate data. These identifying characteristics and restrictions upon who may read or alter specific files can be called attributes.

Figure 1-2 illustrates data storage using filing cabinets.

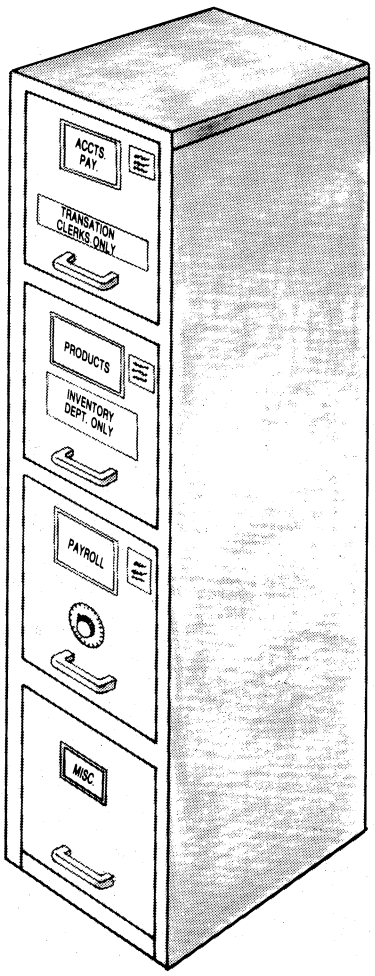


Figure 1-2: Files

In general, the person who uses a file establishes a method of organizing the records within it. This method reflects the file's use and dictates what information is needed and how much time is required to locate a record within the file.

There are several typical methods for organizing records in a file, depending on how the records are used. If you generally use all the records in a file whenever you open it (that is, you have little or no need to locate individual records in the file) and the order of the records is not important, then you can organize the records **sequentially:**

- The records assume the physical sequence in which they are inserted into the file (that is, records are **appended** to the file).
- No empty spaces are left in the sequence of records, where records could be inserted later. Each record, except the first, has a record before it; each record, except the last, has a record following it.

Employee payroll records, for example, might be kept in a sequential file. Because all the records must be accessed every time the payroll is done, sequential file organization would allow easy access to the records.

The overhead and maintenance for sequential files is minimal. To insert a record into the file, you simply put it after the last record already there. Figure 1-3 illustrates sequential file organization.

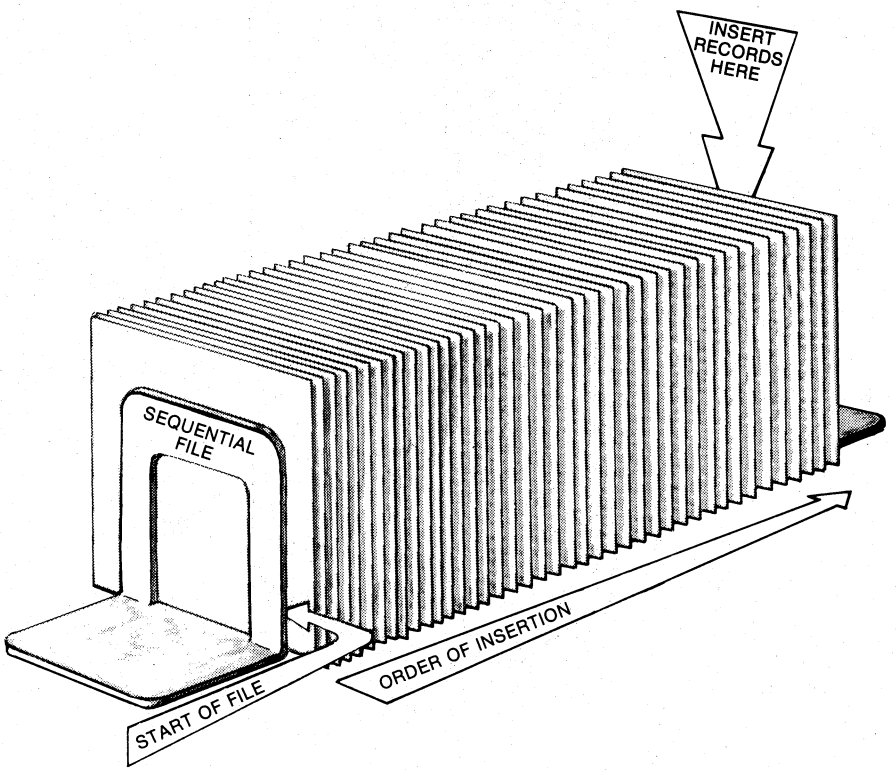


Figure 1-3: Sequential File Organization

For more access flexibility than sequential files, if you want to be able to locate individual records easily, you can set up a series of file folders and number them in sequence from first folder to last. Each folder is the same size; it holds only one record, but it can be empty. Thus, you do not have to look sequentially through the records to locate the one you want (although you can if you want to access all the records). You use the numbers on the folders to locate or insert records; each record will be numbered **relative** to the beginning of the file. The numbers can relate to some numbering system meaningful to your business: for example, order numbers or part numbers.

Figure 1-4 illustrates relative file organization.

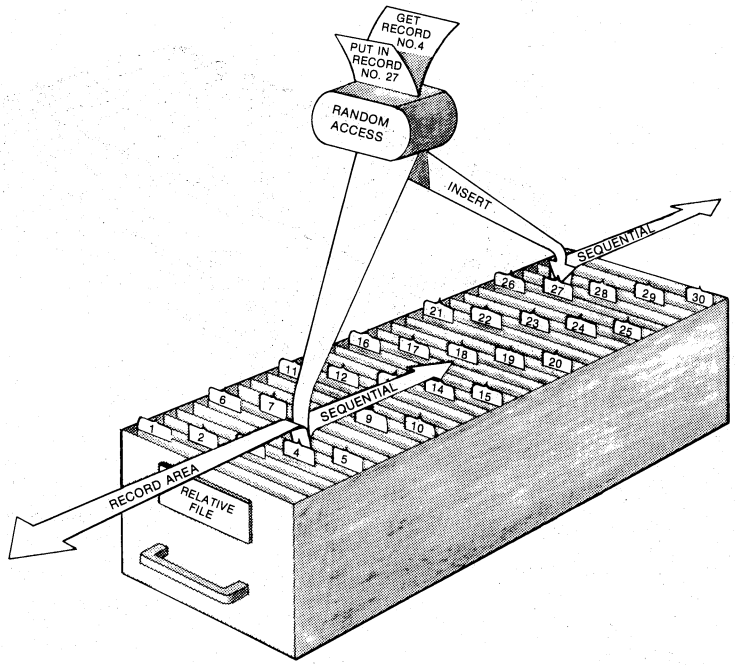
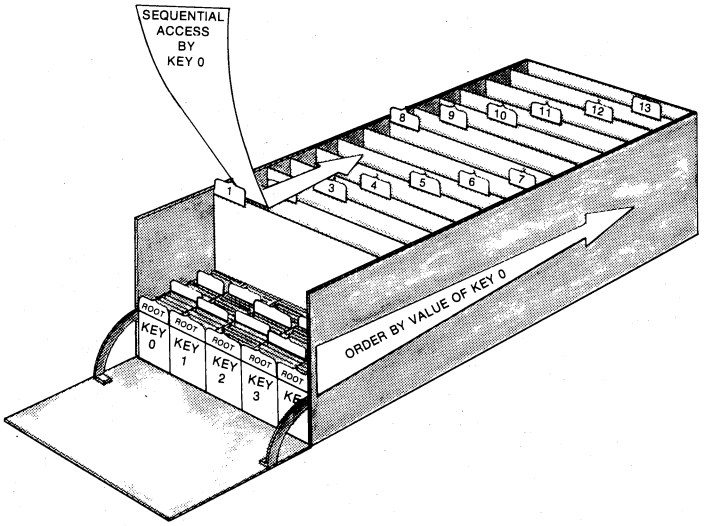


Figure 1-4: Relative File Organization

If you have a large file and most of the time you want to be able to locate individual records, you may want to **index** your files. Indexing is useful when you want to be able to use several kinds of information to locate records. For example, in an employee file, you may want to use last-name information to obtain a report on all employees, and job-designation information to obtain a report on all clerical employees.

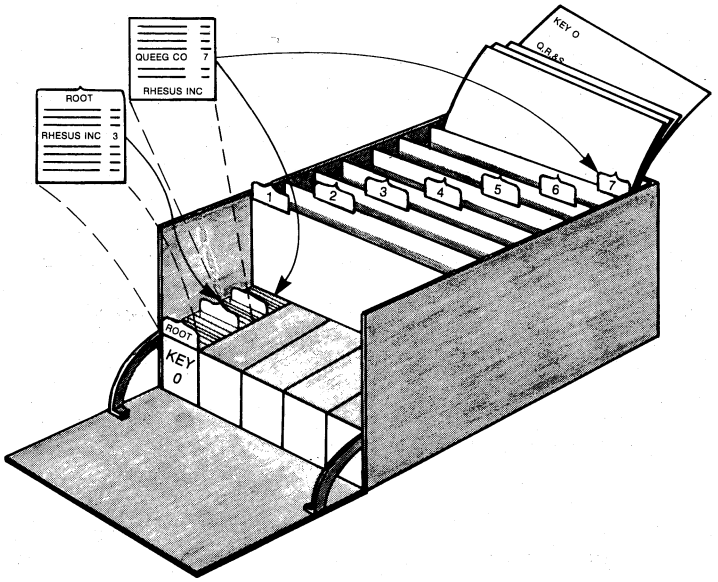
When you open an indexed file drawer, you find records filed with numbered tabs separating them. At the front of the drawer is a set of small card files, containing groups of cards separated by dividers. The cards in each of these small card files are an index to the records at the back of the file. To insert a record in the file, you find the data item marked "key" on the record, and using that information, consult the appropriate index to determine where the record should be inserted. Figure 1-5 illustrates indexed file organization.

To find a record in an indexed file, you look for the specific key information in the appropriate key file and use that information to locate the record. For example, if you want the record of a transaction with the Q,R,&S Company, you open the indexed file drawer for transactions, which contains data records filed at the back and indexes at the front. Figure 1-6 illustrates this example.



ZK-1169-82

Figure 1-5: Indexed File Organization



ZK-1175-82

Figure 1-6: Indexed File Example

You know that company name is the primary key for records in the file and that index 0 indexes the primary keys. The first record in index 0 is the root, which lists selected primary key values, that is, the company names, in alphabetical order. Not all company names appear here: instead, a small subset of names, distributed fairly evenly across the full set of names, is used as the highest level of indexing. By selecting one name, you establish the region of the file (range of names) that interests you.

You look down the list until you find a name that either matches Q,R,&S or occurs after this name in the alphabet. You find Rhesus, Inc with the number 3 alongside it.

You put the root record back in the file and go to the first divider and the third index record behind it. Again, the set of names here is incomplete: only a small set of names distributed fairly evenly across the range covered by the highest level index entry exists. This provides an intermediate level of indexing, and further limits the range of names in which you are interested.

Rhesus, Inc is the last entry on this card, but you scan the list and find the name Queeg Co, which is the first entry at or after Q,R,&S in alphabetical sequence. The entry for Queeg Co has the number 7 alongside it.

So you reach into the data records at the back of the drawer to tab number 7. You search sequentially through the records behind this tab until you find the record of the Q,R,&S transaction.

For another example, using the same transaction file, suppose you want to find a record but all you know is its transaction number. Fortunately, the second alternate key for the file is transaction number. Index 2 indexes the second alternate keys (recall from the previous example, that the indexes are numbered starting with primary index 0). You look at the root record in index 2 and move through the index as you did in the previous example until you find a card listing the transaction number you are looking for. Next to the number is the code 7/5.

So you reach into the data records at the back of the drawer to tab number 7 and count back to the fifth record behind the tab. You find that the transaction you are looking for was made with the Q,R,&S Company.

Here, only one level of indexing -- the root record -- was used. If many records exist in the file, another intermediate level would also be used, as it was in index 0. Use of intermediate index levels allows the number of entries you must scan in each level to be small, regardless of the total number of records in the file.

1.1.3 Access

Once you have records organized in a file, you can get, or access, them in two ways:

- You can search all the records one after the other. This is called sequential access.
- You can use an identifier to locate an individual record. This is called random access.

Note that access means not only retrieving a record from a file but putting a record into the file as well.

Figure 1-7 illustrates the random and sequential access modes.

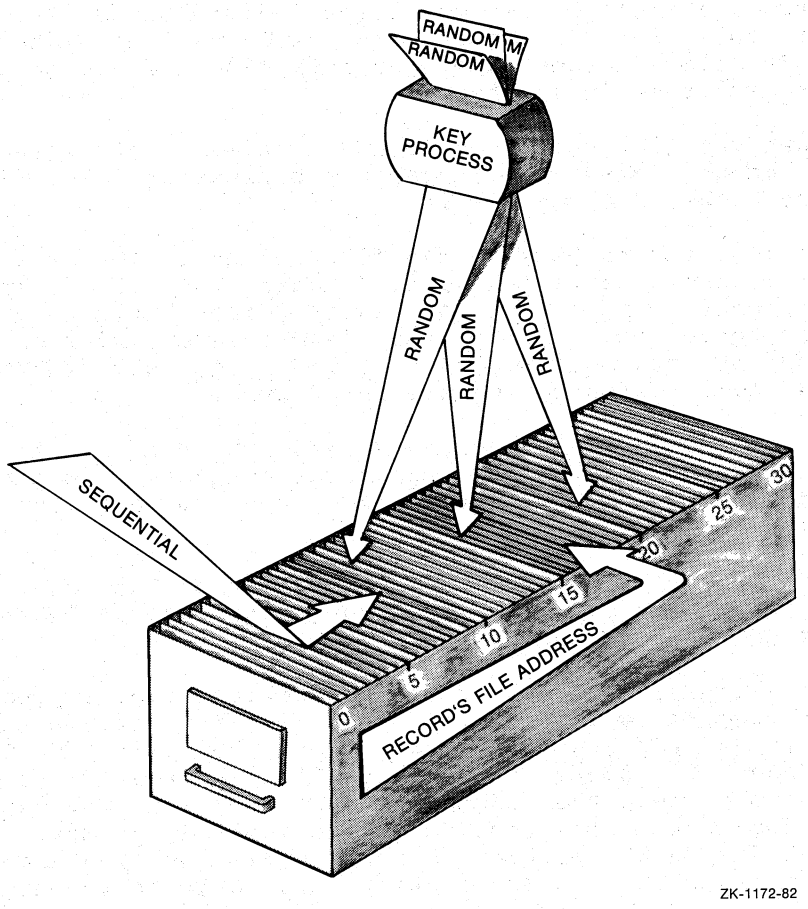


Figure 1-7: Record Access Modes

Sequential Access

For sequential access, you pick a point in the file and access the records beyond that point one at a time. At times, the starting point is the beginning of the file because you want to look at, or access, each record in the file. Other times, you may begin midway through the file.

To read each record, you take it out of the file, marking the position of the record you just removed with a card or some other marker so that you know:

- Where to put the record back into the file
- Where the next record is

To insert records sequentially, you reach into the drawer to the place where you want the records to go and mark the position of that place. Often, the point at which you will insert the new records will be the end of the file. At other times, it may be midway through the file.

You insert the records by taking the first record from the stack of new records and slipping it into position in the file. You then mark the position after the record you just inserted and add the next record in that position. You continue in this manner until all the new records are inserted.

Note that in both retrieving and inserting records you move through the records consecutively. Each record is retrieved or inserted with respect to the record accessed right before it.

Random Access

For random access, you determine the location of the record you want on the basis of some identifier, rather than on the basis of the record's position within the file. If, for example, you have a list of locations of records in the file, you can reach into the file to a record's exact location. Each record selection is independent of the previously accessed record and of the next record to be accessed.

The record identifier can be a number, as for relative files, or it can be a key, as for indexed files. Or, the identifier can be a physical location within the file drawer; for example, you could place each record in a numbered slot within the file drawer and use the slot number to access the records in the file. The slot number would be the address of the record. This type of random access could be used with any type of file organization.

Often, you will want to switch the mode of access you use. You may want to use random access to find the first record in a series and then use sequential access to retrieve all the records in that series. For example, if your employee records are grouped by department codes within the file, you can use a specific department code as the identifier to randomly access the first record with that department code and then switch to sequential access to consecutively read all the records with that code.

Context

In either type of access, sequential or random, the marking of position in the file is important. This is called context: the position of the record you are accessing is the current record, and the position of the record that follows it is the next record.

Access Control

One advantage of the segregation of data provided by files is controlled access. Some files, such as budget or payroll, should be available to only a small group of authorized people. Other files, such as inventory or transaction files, may be used by larger groups of people. And some files, such as the telephone directory, must be accessible to everyone.

Files allow you to control who can use what data. You can lock the filing cabinet that contains the payroll data and give keys to yourself and the payroll manager only. And you can distribute telephone directories to every employee.

In addition, within a file, you can further control how the data can be used within the group of authorized users. Some users may be allowed to write new data in the file or to modify existing data, while others may be allowed only to read the data.

1.1.4 Processing

Once you locate, either sequentially or randomly, a record's position within a file, you will probably want to do something with the record that belongs there. Record operations fall generally into the following categories:

- Verify that the record exists in the right location
- Read the record; that is, examine its data contents
- Insert a record in the position that you have located
- Revise the contents of the record; that is, modify some of its data contents
- Remove the record from the file

1.1.5 File Maintenance

Once you establish files and their records and begin using them regularly, you will want to be able to maintain them to ensure both the protection of the data within them and their continued usability.

Typically, maintenance might include the following activities.

- The data in a file is valuable or you would not keep it. You should have duplicates of your records in some other place in case something happens to the originals. Therefore, you need the ability to back up files.
- If something does happen to your original data, you must be able to obtain, or restore, the duplicate records.
- You need the ability to list, or display, your files, with their names and other attributes.
- Files often grow very large and their usage can change over time. Therefore, you may want to change a file's organization from sequential to indexed; or you may want to reload a file that has grown very large to use space more efficiently. Conversely, usage and file size might decrease and you may want to make a file simpler. It is also possible that the information in one file is suitable for another application. In all these cases, you would want to be able to convert a file into a new one, perhaps changing some attributes (including organization) to make it more usable.
- You want to be able to design and create files that you require.
- Creating an indexed file and putting records into it can be complicated and time-consuming. You would want a procedure -- indexed file loading -- that would produce an optimal indexed file quickly and efficiently.

1.2 RMS-11 IMPLEMENTATION OF DATA ORGANIZATION AND ACCESS

RMS-11 provides file structure capabilities that allow you to organize your data within a computer's memory using the same concepts that were described in Section 1.1 for paper records in filing cabinets.

The following sections briefly present the RMS-11 file structure capabilities. For more details, see RSX-11M/M-PLUS RMS-11: An Introduction.

1.2.1 RMS-11 Record Formats

RMS-11 supports the following record formats that allow you to define the size of your data records:

- Fixed length -- Every record in the file is the same size.
- Variable length -- Records in the file are of different lengths, up to a maximum size that you can optionally specify.
- Variable with fixed control -- Records in the file are of different lengths, up to a maximum size that you can optionally specify, and in addition, a fixed-length control area precedes the data.
- Stream -- Records consist of a continuous stream of ASCII characters delimited by a special terminator character or sequence of characters.
- Undefined -- Records in a file may have no record format or may be in a format different from the four standard RMS-11 formats.

RMS-11's support of stream and undefined record formats provides limited support for non-RMS-11 files.

1.2.2 RMS-11 File Organizations

RMS-11 supports three file organizations:

- Sequential -- Records are arranged within the file in the order in which they were written into the file.
- Relative -- Records are stored in the file in cells, or fixed-length units of storage, one record per cell. The cells are numbered sequentially. These numbers, called relative record numbers, are identifiers for the records.
- Indexed -- Records are arranged in the file in ascending order by key. A key is a data field within the record that RMS-11 uses as an identifier to access the record. An indexed file must have one primary key and may optionally have other alternate keys.

1.2.3 RMS-11 Record Access Modes

RMS-11 provides three record access modes for storing and retrieving records in files:

- Sequential -- RMS-11 stores and retrieves records sequentially, one after another.
- Random by key -- RMS-11 uses either a key (for an indexed file) or a relative record number (for a relative file or for a disk sequential file with fixed-length format records) as an identifier to gain direct access to an individual record in the file.
- Random by record file address (RFA) -- RMS-11 uses the RFA as an identifier to gain direct access to an individual record in the file. The RFA is a unique identifier that RMS-11 establishes for every record that it writes into a disk file.

1.2.4 RMS-11 Utilities

RMS-11 provides utility programs that can help you perform file and record maintenance:

- RMSBCK -- The RMS-11 File Back-Up Utility transfers the contents of an RMS-11 file to another file, which may be on another device, to maintain the file should the original file be lost or damaged.
- RMSRST -- The RMS-11 File Restoration Utility transfers files that were backed up using RMSBCK back to you so your programs can access them.
- RMSDSP -- The RMS-11 File Display Utility produces a concise description of any RMS-11 file, including back-up files.
- RMSCNV -- The RMS-11 File Conversion Utility reads records from an RMS-11 file of any organization and loads them into another RMS-11 file of any organization.
- RMSDES -- The RMS-11 File Design Utility allows you to design and create sequential, relative, and indexed files.
- RMSIFL -- The RMS-11 Indexed File Load Utility reads records from an RMS-11 file of any organization and loads them into an indexed file.

1.3 RMS-11 PROCESSING ENVIRONMENT

The RMS-11 software routines organize data on your computer, implementing the concepts discussed in the previous sections, and provide the interface between your application programs and the computer system.

Your computer system consists of layers of hardware and software:

- The hardware devices -- magnetic tapes and disks -- to store the data.

- The operating system software -- file control processor, device drivers -- controls the hardware to maintain files.
- RMS-11 software controls the internal structure of files (as described in Section 1.2).
- Your application program makes use of these hardware and software facilities to process data records and files.

1.3.1 RMS-11 Task Structure

You use the RMS-11 software routines by combining them with a program you have written in a language that implements RMS-11.

NOTE

Only MACRO-11 programmers can use the full set of RMS-11 capabilities. Subsets of these capabilities are available to high-level language programmers. See your high-level language documentation to determine:

- Which RMS-11 facilities you can use in your high-level language
- The syntax for using these facilities

Once you write your program, you convert it to **object code**, using either a **compiler** or an **assembler**.

To combine your object code with the RMS-11 routines, you use the **task builder**, which converts object code (modules) to an executable form called a **task**. In the process, the task builder not only combines different object modules, but may also arrange the task so that some executable modules **overlay** each other when the task is run.

You can combine RMS-11 routines with your object code in either of the following ways:

- In the task itself, with **nonoverlaid** routines or a **disk-resident overlay structure**
- In **memory-resident overlays**, a form apart from your task

The primary difference between these techniques is that memory-resident overlays can be shared among programs. Nonoverlaid and disk-resident overlaid routines cannot be shared; each accessing program must have its own copy of such routines. In addition, memory-resident overlays eliminate the I/O operations needed to bring disk-resident overlays from disk, thereby making your tasks run significantly faster.

In either case, your task takes a logical form in which program code exists in one part of the task and the RMS-11 routines run in another part. When your program performs an RMS-11 operation, it sets up the necessary parameters and data and calls the appropriate RMS-11 routine. Control jumps to that part of the task, the routine runs to completion, and control returns to your program. Figure 1-8 illustrates this logical structure.

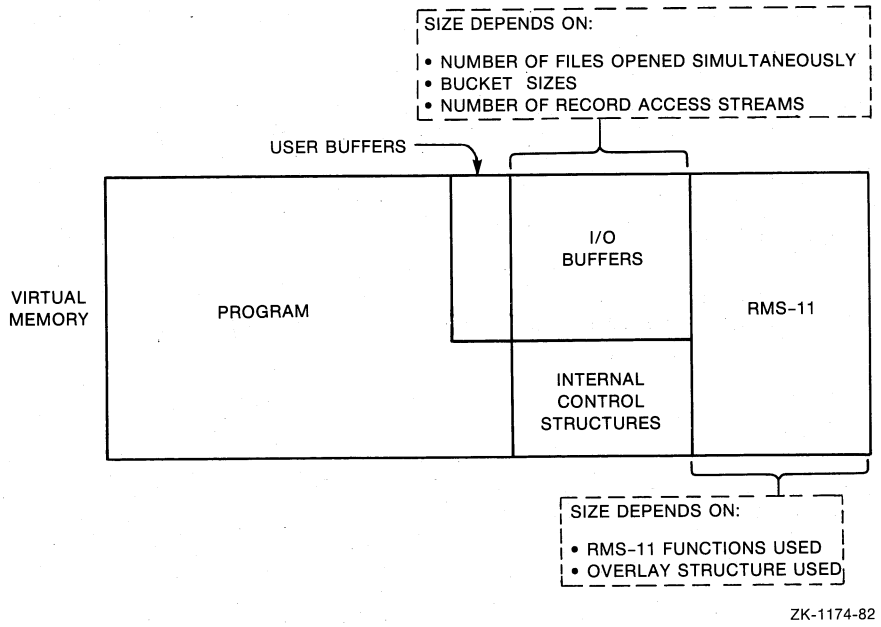


Figure 1-8: RMS-11 Task Structure

Also part of the task are storage structures, which generally take three forms:

- User buffers -- These buffers are used to pass data records between your program and RMS-11. They are available to your program and the data in them can be manipulated, read, changed, used for calculations, and so on.

- I/O buffers -- For each file your program has open, RMS-11 normally requires at least one internal I/O buffer. All data going to or coming from disk is stored in an I/O buffer as follows:

  - RMS-11 requests the file control processor to move block(s) from a disk file into this buffer to satisfy your program's requirements. Each request normally specifies the same number of blocks, called an I/O unit. The size of the I/O unit depends on the file organization, file design, and settings at access time (such as multiblock count).

  - RMS-11 moves records between the I/O buffer and the user buffer. Your program can also directly access a record within the I/O buffer in certain restricted circumstances.

- Control structures -- RMS-11 control structures, called control blocks, are used to communicate between your program and the RMS-11 routines and with each other. Some are accessible to your program; others are for RMS-11 internal use only.

1.3.2 RMS-11 Record Processing

The RMS-11 stream and record operations are the interface between your program and the data records your program requires.

Before your program can access records in a file, the file must be open and an access stream must be initiated.

NOTE

Most high-level languages do not support access streams at the user level. They use the RMS-11 access stream facilities to implement their own file access techniques.

An access stream is a path to the file's data records; record operations are performed via that stream, one operation at a time. RMS-11 keeps track of the stream's position, or context, in a file, in terms of current record and next record. The stream's position changes at the completion of an operation. Chapters 3, 4, and 7 discuss context for record operations with the different file organizations.

The stream operations control the stream associated with a file. They are:

- CONNECT -- initiates an access stream.
- DISCONNECT -- terminates a stream.
- FLUSH -- writes the currrent contents of I/O buffers to the file.
- FREE -- releases control of the record or block most recently accessed (and locked) by the stream.
- REWIND -- resets the stream context to the first record in the file.
- WAIT -- suspends processing until an outstanding asynchronous operation is completed.

The record operations process records within a file. They are:

- FIND -- reads a record from a file to an I/O buffer and sets the current-record context to that record.
- GET -- reads a record from a file to an I/O buffer and then to a user buffer, and sets the current-record context to that record.
- PUT -- writes a new record from a user buffer to an I/O buffer and then to a file.
- UPDATE -- transfers a modified record from a user buffer to an I/O buffer and then to a file, overwriting the previous copy of the record in the file.
- DELETE -- removes an existing record from a relative or indexed file.
- TRUNCATE -- effectively deletes all records in a sequential file from the current record to the logical end-of-file.

For the FIND, GET, and PUT operations, your program specifies the record access mode -- sequential, random by RFA (FIND and GET only), or random by key -- which determines which record is the target of the operation.

See RSX-11M/M-PLUS RMS-11: An Introduction, Chapter 4, for a more detailed introduction to record processing. Chapters 3, 4, and 7 of this user's guide describe specifically how the record operations work depending on the file organization selected and (for FIND, GET, and PUT) the access mode specified.

1.3.3 RMS-11 File Processing

RMS-11 must manipulate the contents of files so that it can process records. However, RMS-11 does not directly perform the actual file manipulation, and the flow of data, control, and overlay segments that the file manipulation entails. RMS-11 issues requests to the file control processor to perform the actual I/O and other operations on the files. Thus, the file control processor's internal operation, while invisible to RMS-11, can affect your program's performance.

The file control processor is not concerned with the data records in a file. It knows only virtual and logical block numbers, directories and other information, and the disk drivers involved. Therefore, RMS-11 can direct file manipulation as long as it makes the proper requests to the file control processor. To do so, RMS-11 maintains the following structures, or I/O units:

- Blocks -- The I/O unit for sequential files is the block. You can adjust the block count for each record access stream so that more than one block can be moved during each I/O operation.

  In addition, you must decide whether records can cross block boundaries. When records can cross block boundaries, RMS-11 can pack them with optimal density in the file because a record can be stored in one or more blocks. This is called block spanning. Figure 1-9 illustrates block spanning.

  When records are restricted by block boundaries, each record must be no more than 512 bytes (one block) long, and unused bytes may be left at the end of each block.

- Buckets -- The I/O unit for relative and indexed files is the bucket. A bucket consists of one or more blocks that RMS-11 treats as a unit. Records can cross block boundaries but they cannot cross bucket boundaries. Bucket size is a file attribute that you specify when you create the file.

  Buckets are an RMS-11 concept, so when RMS-11 initiates an operation for a relative or indexed file, it requests the file control processor to move a bucket by specifying the virtual block number for the first block in the bucket and the size of the bucket in bytes. Note that buckets are fixed within the file; once created, buckets contain the same virtual blocks at all times.

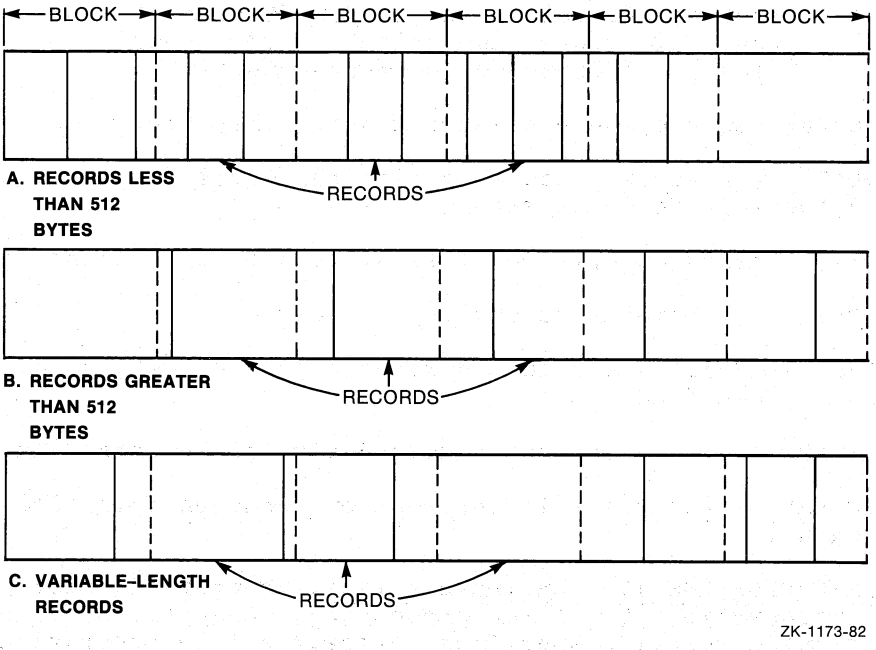


Figure 1-9: Records Spanning Blocks

You can also direct RMS-11 to request the file control processor to place a file on a disk at a specific location. This is called placement control and can improve performance by taking advantage of, for example, tracks and cylinders.

RMS-11 provides access sharing; that is, your program can control who can gain concurrent access to the data in a file and what type of operations they can perform on the data. See Section 2.2.3 for more information on access sharing.

The RMS-11 directory and file operations perform the file-level functions. The directory operations affect file specification entries in directories (not the contents of the files). They are:

- ENTER -- places a disk file specification in a directory.
- REMOVE -- deletes a disk file specification from a directory.
- RENAME -- replaces an existing disk file specification with a new one.
- PARSE -- returns file specification information to your program.
- SEARCH -- examines one or more directories for a specified file and returns the file specification and location.

NOTE

Most high-level languages do not support the directory operations. See your high-level language documentation.

The file operations provide access to files. They are:

- CREATE -- creates a new file with the attributes you specify and opens it for processing.
- OPEN -- makes an existing file available for processing.
- CLOSE -- terminates access to a file.
- ERASE -- deletes a file and removes its directory entry, if one is specified.
- EXTEND -- increases the allocated size of an open file.
- DISPLAY -- returns file information about an open file to your program.

See RSX-11M/M-PLUS RMS-11: An Introduction, Chapter 4, for a more detailed introduction to file processing. Chapters 3, 4, and 7 of this user's guide describe specifically how the file operations work depending on the file organization selected.

1.4 FILE ATTRIBUTES

When you create an RMS-11 file, either through a program (using the CREATE file operation routine) or by using the RMSDES utility, you must specify the following information:

- Medium -- Disk or magnetic tape. You can also create files on unit-record devices, such as line printers and terminals. Note that relative and indexed files are restricted to disk devices.
- File specification -- The name you assign to a file enables RMS-11 to find the file later. Use the file specification conventions specific to your operating system.
- Protection -- RMS-11 allows you to assign a protection code to a file when you create it. Use the protection codes specific to your operating system.
- File organization -- Sequential, relative, or indexed.
- Record format -- Fixed length, variable length, VFC, stream, or undefined.
- Record size -- For fixed-length records, the size is the same for every record in the file. For variable-length records, the size is the maximum length any record can be.

  For VFC records, there are two size specifications: (1) the fixed length of the control area, and (2) the maximum length of the variable data area.

  RMS-11 also keeps the length of the longest record actually stored in a sequential file for variable-length and VFC records.

- Block spanning (sequential files) -- Whether records can cross block boundaries.

- Bucket size (relative and indexed files) -- The number of blocks in each bucket.

- Maximum record number (relative files) -- The maximum number of records that the file will contain.

- Keys (indexed files) -- The number of keys; the position and size of each key; the data type for each key; and other key characteristics.

- Record-output handling -- You can specify three (mutually exclusive) types of handling for records being written directly to a unit-record device, although you need not specify any:

  - Carriage control -- The device driver inserts a linefeed character as a prefix to each record and a carriage-return character as a suffix to each record before passing it to the device.

  - FORTRAN -- The device interprets the first byte of each record as a FORTRAN forms control character.

  - Print file format (VFC records with a fixed header size of 0 or 2 bytes) -- RMS-11 interprets the first byte of the header as a prefix for the record and the second byte as a suffix for the record.

- File allocation -- You must specify two quantities:

  - Initial allocation -- the size of the file in blocks when it is created.

  - Default extension quantity -- the number of blocks to be added to a file when RMS-11 automatically extends it.

- Contiguity -- Whether the disk space initially allocated to the file is to be allocated in continuous, adjacent logical blocks.

- Placement control -- Where the file is to be physically located on the disk.

During the file creation process, RMS-11 stores this information, called the file attributes, in the file directory and, for relative and indexed files, in the first blocks, or prologue, of the file as well.

After creation, for the life of the file, RMS-11 gets information about a file from the file itself. This offers several advantages:

- Most file attributes do not change.

- You can design your RMS-11 files offline. No program accessing the files need specify attributes (except those that may be required by high-level languages), because RMS-11 requires only a file specification from a program to open a file.

- You can open an RMS-11 file with its file specification only. After that, RMS-11 enables you to read the file attributes. You can write a program or use the RMSDSP utility to display those attributes.

Note that some of the attributes are interdependent; that is, the selection of one attribute directly affects, or restricts, other attribute options. File organization, record format, and medium are all interdependent. For example, if you select magnetic tape medium, you must use sequential file organization. And if you select VFC records, you cannot use indexed file organization and you must use a disk device.

Table 1-1 lists the record format and file organization interdependencies.

Table 1-1: Record Formats and File Organizations

File Organization	Record Format: Fixed	Variable	VFC	Stream	Undefined
Sequential:					
Magtape	Yes	Yes	No	No	No
Disk	Yes	Yes	Yes	Yes	Yes
Relative	Yes	Yes	Yes	No	No
Indexed	Yes	Yes	No	No	No

Chapters 3 through 7 discuss your file design options in detail, depending on your selection of file organization. Chapter 2 provides information to help you make that selection.

1.5 PROCESSING BY BLOCK ACCESS

Your program can bypass RMS-11 record processing and process any RMS-11 file in a mode called block access.

Your program can read or write blocks in a file either sequentially or (on disk only) randomly by virtual block number (VBN). But your program must be able to interpret the contents of those blocks.

See RSX-11M/M-PLUS RMS-11: An Introduction for an introduction to block access and processing. See the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide for detailed information on block access and processing.

CHAPTER 2

APPLICATION DESIGN

When you write an application program, you want that program to input data, process it, store it, update it if necessary, and at intervals output it in the proper formats.

You want all this to happen simply, quickly, and accurately. You must therefore take the time to design your application by carefully considering RMS-11 file structure and file and record processing capabilities. Important RMS-11 considerations are data storage medium, record format, file organization, access mode, allocation, overlays, and so on.

If you do not consider RMS-11 capabilities when you design your application, you may not get the best peformance possible from your application because of the defaults that will be applied automatically to your files (see Section 2.1).

Example: The first time one user created a file, she used a high-level language program and took all the defaults. Then she loaded records into the file; the process was quite lengthy.

However, when she re-examined the file and re-created it applying some RMS-11 design considerations, the record insertion process went 10 times as fast.

Example: Some users, accustomed to programming with BASIC-PLUS record I/O, learned that RMS-11 uses 15 bytes of control data in each bucket and 7 bytes of control data for each fixed-length record in an indexed file (see Chapter 6). Then, because they were accustomed to working with whole blocks, they set up single-block buckets (512 bytes) and subtracted RMS-11 overhead (22 bytes) to come up with a record size of 490 bytes.

But when they used those files, the users were alarmed to see them grow at high rates. They had not read that RMS-11 preserves its fast sequential and alternate key access during random insertions by moving records and leaving behind 7-byte pointers (see Chapter 5). Therefore, when one of those 490-byte records was moved, it left behind 7 bytes, which meant that no other record fit into that bucket. Soon the file was filled with practically empty buckets that could not be used because the designers did not allow for the full implications of RMS-11 structure.

If you develop an application with a high-level language, you probably will not worry about RMS-11. You will accept the language's concept of design, if any. It is possible, however, that the defaults the language uses in its interface with RMS-11 are not well suited for your application.

This chapter presents general design considerations that apply to all application designs and information that will help you make the first important design decision: selection of a file organization.

2.1 WHEN TO DESIGN

There are two times to design an application:

1. Before you write the application, especially if you have:
   - Large file(s)
   - Many users simultaneously accessing the file(s)
   - A high level of activity (many records read, written, updated, or deleted in a given time period)
2. After you write the application, if you are not happy with its performance.

Often, poor performance results from default values that are inappropriate for your application. You can frequently find improvements by studying the nature and source of the defaults and how they affect the structure of your application and your file.

Basically, defaults have three sources:

1. Source language compilers

   In many instances, source language compilers such as COBOL-81 or BASIC-PLUS-2 supply default values for RMS-11 file attributes and/or facilities.

   Example: RMS-11 does not calculate an optimal bucket size for indexed files. Rather, the program creating the file must specify a bucket size. When that program is the product of a compiler, the bucket size can be explicitly specified in the source code or it can be implicitly set by the compiler, using a default value.

2. RMS-11

   The interface between the RMS-11 routines and your program has the same structure in all tasks, regardless of their source (PDP-11 COBOL-81, RPG, MACRO-11, and so on). This interface consists of control blocks (see the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide for details). The information provided by your program in these blocks effectively controls RMS-11, causing it to create, open, access, and close files. However, when explicit information is not provided, RMS-11 uses its default values.

3. Operating system

   RMS-11 acts as an intermediary between your task and the operating system. As such, RMS-11 can supply control information for system functions such as protection codes. However, if RMS-11 supplies no control data, the system uses its defaults.

2.2 DESIGN CONSIDERATIONS

When you design your application, you are concerned primarily with four design considerations:

1. Speed -- You want to maximize the speed with which the programs process data.

2. Space -- You want to minimize the room for the data and the task on disk and the memory the task takes to run.

3. Shared access -- You want your data to be exactly as accessible to the people using the computer system as necessary.

4. Ease of design -- You do not want to spend more time than necessary writing the application.

Remember, the importance of design is proportional to the complexity of the file organization. That is, design is least important for applications using sequential files and most important for applications using indexed files.

2.2.1 Speed

You can make many performance (speed) decisions before you have to consider anything else. Therefore, the first criterion to apply throughout the design process is minimize I/O time.

The mechanics of the mass storage devices on your system consume most of the time for any RMS-11 operation. The memory-resident routines that prepare the data for I/O or process it afterwards are very much faster (one to three orders of magnitude).

An application's entire environment affects I/O time:

- File structure -- A variety of file attributes affect I/O time, including:

  bucket size (for a relative or indexed file)
  number of keys (for an indexed file)
  number of duplicate key values (for an indexed file)
  initial file allocation
  default extension quantity

- File size -- The number of records in the file affects the I/O operations required to scan a file sequentially or follow an index.

- Program -- Your program affects I/O time by requiring I/O operations for file operations (OPEN, CLOSE, and so on), record operations (GET, PUT, and so on), and overlays.

- RMS-11 -- The RMS-11 routines can be structured as disk-resident overlays or as memory-resident overlays.

- File control processor -- Besides requiring overlay segments from disk, the file control processor can also request I/O operations required to map virtual blocks of the file to logical blocks on the storage device.

- Device hardware -- The storage device that contains the task and data files is the primary contributor to the length of an I/O operation. The type of device chosen (moving-head, fixed-head, and so on) and the demands on it (amount of I/O activity for that device within the system) are crucial to I/O performance.

Figure 2-1 illustrates this environment.

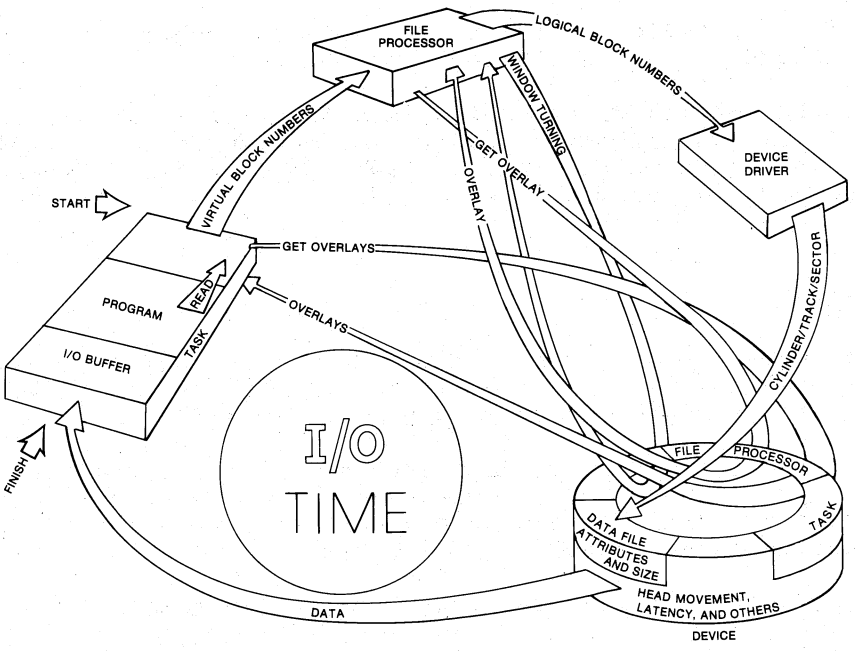


Figure 2-1: Time Factors in an I/O Operation

2.2.2 Space

RMS-11 requires space for three reasons:

1. To store data in a file
2. To store the RMS-11 routines either (a) on disk when they are not in use, or (b) in memory when they are being executed
3. To buffer data in memory while the task runs

2.2.2.1 Data Storage - The space RMS-11 requires to store data is proportional to the organization of the file, and to the processing capabilities of that organization:

- Sequential file organization -- RMS-11 adds to the size of your data an empty byte, if necessary, to align each fixed-length, variable-length, or VFC record on an even-numbered byte boundary. When the file contains variable-length records, RMS-11 also prefixes a count field to each record.

- Relative file organization -- RMS-11 constructs a series of record storage cells based on the length of the records. The cells are 1 byte longer than the fixed size of fixed-length records or 3 bytes longer than the maximum size specified for variable-length records.

- Indexed file organization -- RMS-11 adds to your data:

  - An index for each defined key.

  - 15 bytes of formatting information for each bucket.

  - A 7-byte header for each record.

  - A count field for each variable-length record.

  - Other overhead of varying lengths for records RMS-11 moves during file activity and for deleted records.

You should keep the size of records to the minimum required for your application.

2.2.2.2 Task Size - The space RMS-11 routines occupy in a task depends on the method you use to link the routines with your program. See Section 8.1 for more details.

2.2.2.3 Buffer Sizes - You can vary the size of the I/O buffers RMS-11 uses to store data in memory. Generally, the larger the buffers, the faster the task processes data. See Section 3.5.3, Section 4.5.3, or Section 7.4 for the file organization(s) you are interested in.

2.2.3 Shared Access

Shared access revolves around the question: Who is allowed to read from or write to a file? The answer involves your operating system's protection codes, your access declaration, and your sharing declaration.

System Protection Codes: Before you can access an RMS-11 file, you must log into your computer system using an account number that will allow you the kinds of access you need when your access request is validated against the file's protection codes.

Operating systems allow you to assign a protection code to each file when it is created. This code describes concentric circles of users who are allowed different levels of access to that file. See your operating system documentation for specific protection conventions.

Figure 2-2 illustrates the system protection concepts.

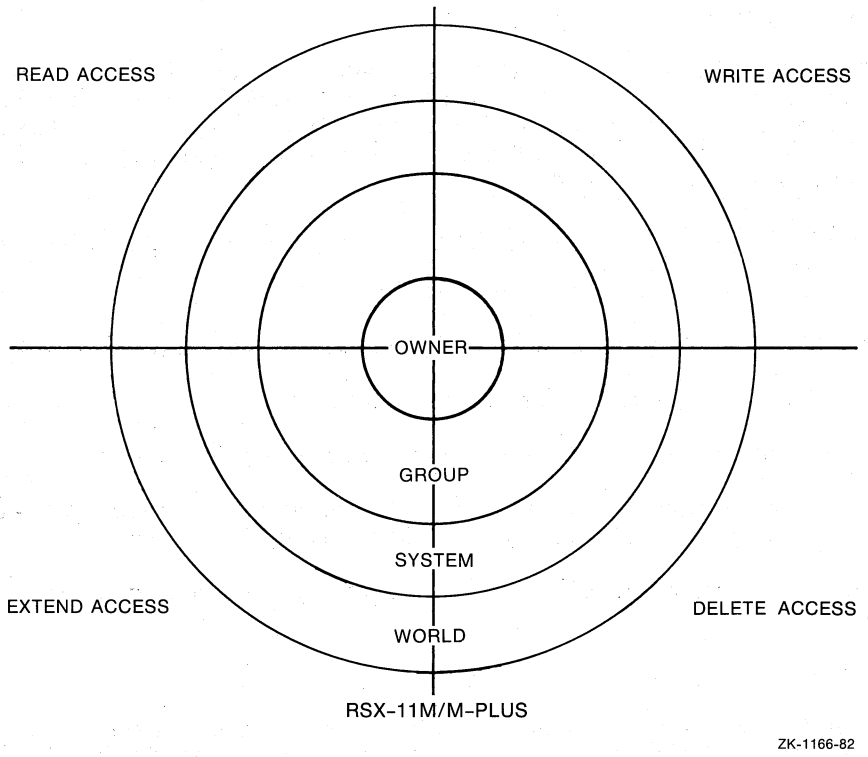


Figure 2-2: System Protection Concepts

Access Declarations: Your program must declare the types of access you need by specifying the record or block operations it intends to perform on the file, as follows:

- **Read-only access** is granted if your program specifies that only FIND/GET or READ operations can be performed.

  No PUT, UPDATE, DELETE, TRUNCATE, or WRITE operations will be allowed, nor will any other operation which would modify the file (an EXTEND operation, for example, will not be allowed for read-only access).

- **Read/write access** is granted if your program specifies that PUT, UPDATE, DELETE, TRUNCATE, or WRITE operations can be performed. FIND/GET and READ operations will also be allowed, as will EXTEND operations.

  Note that, in addition to any access declaration, a CREATE operation always forces read/write access so that the newly created file can be populated (using PUT operations for record access or WRITE operations for block access).

Sharing Declarations: Your sharing declaration specifies the types of access to the file that your program is willing to allow to other programs that request access to the file at the same time that your program is accessing it. These declarations can be:

- **No sharing** -- You do not want any other program to access the file.

  A no-sharing specification in your sharing declaration overrides any other sharing specification you may also have included, and guarantees that no other program will have concurrent read/write access to the file. That is, no other program will be able to modify the file.

  Note, however, that it is not possible to guarantee that concurrent read-only accessors will be denied access.

- **Read-only sharing** -- You are willing to allow other programs read-only access to the file.

- **Read/write sharing** -- You are willing to allow other programs read/write, as well as read-only, access to the file.

- **Sharing with user-provided interlocks** (sequential files only) -- This specifies a special form of sharing among a group of programs that includes any number of read-only accessors and at most one read/write accessor.

  User-provided interlocks offer a limited form of access sharing of sequential files. If the file organization is sequential, this specification in your sharing declaration overrides any other sharing specification (except no sharing). For any other file organization, this specification is ignored.

NOTE

High-level languages may use slightly different terms to designate the access and sharing declarations, and may not provide equivalents for all the sharing options. See your high-level language documentation.

Once the operating system's protection checks are passed, RMS-11 and the operating system cooperate to determine whether the type of access you request (your access declaration) and the type of sharing you permit (your sharing declaration) are consistent with any other current accessors of the file.

If no other tasks have accessed the file at the time that your program requests access, your access request must only pass the system protection checks to be granted. However, if one or more programs already have access to the file, RMS-11 and the operating system will use the access and sharing declarations of those programs along with those of your program to determine whether your program will be allowed concurrent access.

No-sharing and read-only declarations are processed as described above for files of all organizations and access method (block or record). In other cases, however, RMS-11 and the operating system interpret the access and sharing declarations in the manner best suited to the file's organization and the access method, as described in Section 3.4 for sequential files, Section 4.4 for relative files, and Section 7.1 for indexed files.

NOTE

As noted, file sharing is a cooperative effort between RMS-11 and the operating system. The RMS-11 processing algorithms depend upon the detailed nature of this cooperation. If you access a file concurrently with multiple programs, some of which use RMS-11 and some of which do not, the results may be unpredictable.

2.2.3.1 Bucket Locking - Any time a record is updated, accessing programs must be assured that the data written to the file is current until the record is re-accessed and the record updated again.

If no control is placed on access, two or more programs could access the same record, one after the other, and update it, one after the other. Only the last update would remain in the file. Access sharing could thus impair data integrity.

To ensure data integrity, RMS-11 uses bucket locking for a relative or indexed file when the file is open for write-shared access. From that point, RMS-11 requests the operating system to lock each bucket read from disk until RMS-11 explicitly releases the bucket. After a GET, FIND, or mass-insert PUT operation, only the bucket containing the data record remains locked. (See Chapter 7 for information on mass insertion.) While that bucket is locked, no other program can access it.

RMS-11 requests the operating system to unlock such a bucket when one of the following occurs:

- The GET, FIND, or PUT operation fails.
- The GET or FIND operation succeeds -- if the program has declared read-only access to the file.
- The program initiates another record operation that accesses a different bucket.

After the bucket is unlocked, other programs can access it.

Example: Programs A and B are write-sharing a file named RMSREL.DAT. Both try to update relative record number 12. However, program B initiates the prerequisite GET operation first, locking the bucket containing the record. The operating system keeps program A from accessing that bucket while program B uses it. After program B updates record 12, RMS-11 unlocks the bucket and the operating system allows program A to get record 12 (including program B's updated data). Figure 2-3 illustrates this example.

Bucket locking incurs costs: The operating system administers bucket locking. It establishes, for each file, a list of virtual blocks that are locked. The system must scan this list every time RMS-11 performs an I/O operation and then either permit the operation or return an error. In addition to this lock-list overhead, extra instructions are executed to lock and unlock the buckets.

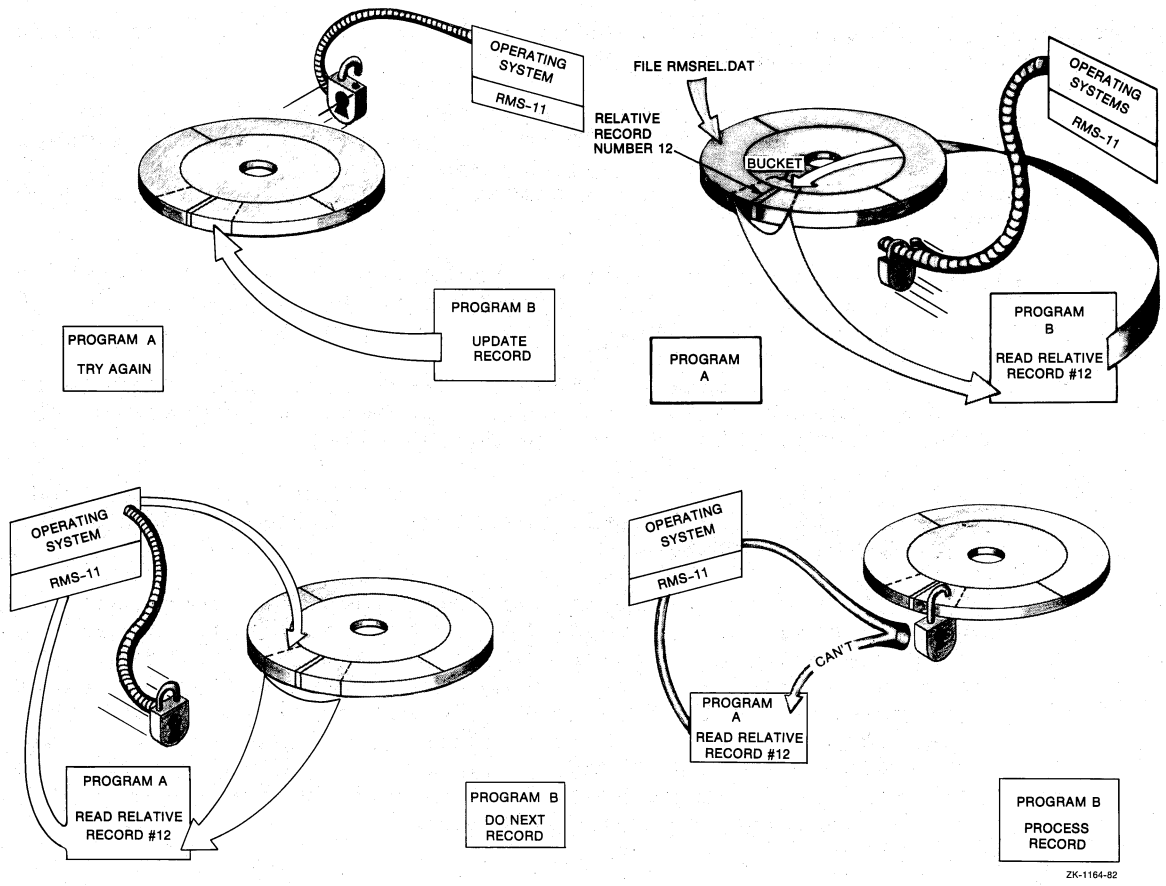


Figure 2-3: Bucket Locking Example

2.2.3.2 Sharing among Access Streams - In addition to the bucket locking used when programs allow sharing, RMS-11 provides its own version of bucket locking when a program accesses a file for write-type operations. This locking allows multiple streams to share the file. RMS-11 bucket locking works the same way as the locking provided by the operating system, except that the locks can be encountered only by different access streams within the same program.

The overhead for RMS-11 bucket locking is small.

2.2.3.3 Programming Considerations - For the greatest flexibility at run time, you should assume that access to any record by your program can be denied because the bucket containing the record is locked. RMS-11 returns the error code ER$RLK when the bucket is locked by another access stream in the same or in another program.

Therefore, you should use the following techniques when you write RMS-11 programs that involve shared access:

- Never keep a bucket locked longer than necessary. You should follow any successful GET or FIND operation with another record operation of any type as soon as possible. The second operation unlocks the bucket locked by the read-type operation.

  Alternatively, you can release the bucket explicitly with a FREE operation. A FREE operation releases only the bucket locked by the access stream associated with the operation.

- If your program detects an ER$RLK error (or its high-level language equivalent), its error processing depends on the number of access streams active on the file:

  - Single stream -- Set up a loop that stalls, then re-initiates the record operation until RMS-11 indicates a successful completion.

  - Multiple streams -- Do not set up a loop that continuously re-initiates the record operation. You should either (a) continue processing on the other streams, attempting the record operation on the locked-out stream periodically, or (b) release the buckets locked by all other streams, then re-initiate the record operation that failed. Any GET-UPDATE or FIND-UPDATE sequences interrupted on the other streams must be restarted, because the release of a bucket destroys the record context.

2.2.4 Ease of Design

When you design and write your application, you should consider yourself and the person who will maintain the application. Keep the following design guidelines in mind:

- Keep things simple. You can apply this criterion to the whole development process, from program flowcharts to the record layouts to file organization and design.

  Example: From sequential through indexed, the RMS-11 file organizations offer increasing capabilities, but they are also increasingly complex. Choose the organization that supplies enough capabilities, but no more. For instance, if you want to randomly access a file by a single key only, you might use a relative file and a hashing algorithm instead of an indexed file.

- Apply optimizations one by one until you reach a satisfactory level of performance. Generally, further improvements are not necessary.

Example: The optimization of performance of applications using indexed files can be involved, but you do not have to use every technique discussed in this manual. You should only satisfy current performance requirements. For instance, when an application program needed optimization, the indexed file being read was made contiguous (see Chapter 6) and the RMS-11 overlay structure was changed (see Chapter 8). Execution time dropped from 16 minutes to 8.5. Since this performance was adequate, no further optimizations were considered.

Some optimizations apply to one type of record operation, but not to others. Determine whether an optimization will benefit your processing before you implement it.

2.3 DESIGN PROCESS

The first step in the design process is the selection of the file organization. Section 2.4 presents information to help you make this selection.

Once you have selected a file organization, go to the appropriate chapter(s):

Sequential	Chapter 3
Relative	Chapter 4
Indexed	Chapters 5, 6, 7

Each chapter discusses file structure (physical and conceptual) as well as design considerations. Indexed files are the most complex to design because of their power and flexibility.

After you read the file organization chapter(s), go to Chapter 8, Task Building and Common Optimization Techniques.

Finally, apply the design considerations described in these chapters. Write your application; create and populate the files, using the RMS-11 utilities when they are useful; use the programs and files in a simulated environment while you evaluate performance. You may have to return to this manual, changing your design and/or combining attributes and RMS-11 facilities in different ways, until the application runs to your satisfaction.

Good design is important to the success of your RMS-11 application.

2.4 SELECTING A FILE ORGANIZATION

Table 2-1 lists important features of each file organization -- sequential, relative, and indexed -- to help you decide which one(s) you need. Table 2-2 points out advantages and disadvantages of each organization.

The sections that follow the tables provide information about two of the features of file organization -- record format and I/O techniques -- to help you select a file organization.

Table 2-1: File Organization Characteristics and Capabilities

Characteristics and Capabilities	Sequential	Relative	Indexed
Medium			
Disk	Yes	Yes	Yes
Magnetic Tape	Yes	No	No
Unit Record	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	None	1 byte	7 bytes
Access Modes			
Sequential	Yes	Yes	Yes
Random	Yes[1]	Yes	Yes
RFA access (disk only)	Yes	Yes	Yes
Record Operations			
CONNECT	Yes	Yes	Yes
DELETE	No	Yes	Yes
DISCONNECT	Yes	Yes	Yes
FIND	Yes	Yes	Yes
FLUSH	Yes	Yes	Yes
FREE	No	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 of operation	Selectable	Selectable
Multiblock count	Yes	Bucket size	Bucket size
Multiple access streams	No	Yes	Yes
Multiple buffers	No	Yes	Yes
Mass insertion	No	No	Yes
Access Sharing[2]	Read-only	Read/write	Read/write
Other Features	Block-spanning records	Maximum record number	Areas

1. For fixed-format disk sequential files only.

2. See exceptions in Section 2.2.3, and in Sections 3.4, 4.4, and 7.1.

Table 2-2: File Organization Advantages and Disadvantages

Organization	Advantages	Disadvantages
Sequential	Simplest organization. Optimal use of disk and memory: • minimum overhead on disk • block spanning Optimal if application accesses all records on each run, except if file must be write-shared. Most versatile in record formats: • exchange data with non RMS-11 systems • compatible with RSX-11M/M-PLUS FCS files[1] • compatible with ANSI magnetic tape format • compatible with RSTS/E stream files[1] Most versatile in storage media; file is portable. Random by key (RRN) record access available on fixed-format disk sequential files.	To get a record, most high-level languages must access all records before it (no access by RFA or by key).[2] You can add records only at end of file.[2] Interactive process is awkward: operator must wait as a program searches for a record.[2] Certain compiled programs cannot access a record already passed without closing and re-opening file (REWIND is not available). You can delete records only at end of file; use TRUNCATE record operation. Sharing normally restricted to multiple readers.
Relative	Random access in all languages. Allows deletions. Allows random GET and PUT operations.	Restricted to disk. File contains a cell for each cell number between 1 and last record in file; data may not be stored densely.

1. RMS-11 can read these file structures and return a record to your program. However, differences in data storage techniques among programming languages can keep the program from properly interpreting the contents of that record.

2. These restrictions do not exist for disk sequential files with fixed-length record format; records in such files can be stored and retrieved using random by key access, depending on your high-level language capabilities.

(Continued on next page)

Table 2-2 (Cont.): File Organization Advantages and Disadvantages

Organization	Advantages	Disadvantages
Relative (Cont.)	Optimal if application accesses all records on each run and file must be write-shared.	Program must know relative record number or RFA of record 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 number.
	Can be write-shared.	You can insert records only into unused record cells, but you can update existing records.
		RMS-11 does not allow duplicate relative record numbers.
Indexed	Most flexible random access:	Highest overhead on disk and in memory.
	• by any one of multiple keys or RFA • key access by generic or approximate value • you access records by record contents	Restricted to disk.
		Least simple programming.
	Duplicate key values possible.	
	Automatic sort of records by primary and alternate keys; available during sequential access.	
	Record location is transparent to user.	
	Can be write-shared.	
	Potential range of key values not physically present as in relative file organization.	
	Variety of data formats for keys.	

2.4.1 Record Formats

RMS-11 supports all of the record formats described in the following sections for sequential files, but restricts relative and indexed file organizations (see Table 2-1).

2.4.1.1 Fixed-Length Format - Records in the file are the same size, which is a file attribute. The fixed-length record format requires no RMS-11 overhead.

RMS-11 limits fixed block-spanning records to 32,765 bytes, while the minimum valid record is 1 byte of data.

2.4.1.2 Variable-Length Format - Records in the file can be any length, up to a maximum of 32,763 bytes for block-spanning records. This file attribute is user-settable and optional. For each record, RMS-11 maintains a count field specifying the number of data bytes in the record. The size of this field depends on the storage medium for the file.

- On disk, the count field is a 2-byte binary count that does not include the 2 bytes for the field.
- On ANSI magnetic tape, the count field is a 4-character decimal count that does include the 4 characters for the field.

Figure 2-4 illustrates the count field for each medium.

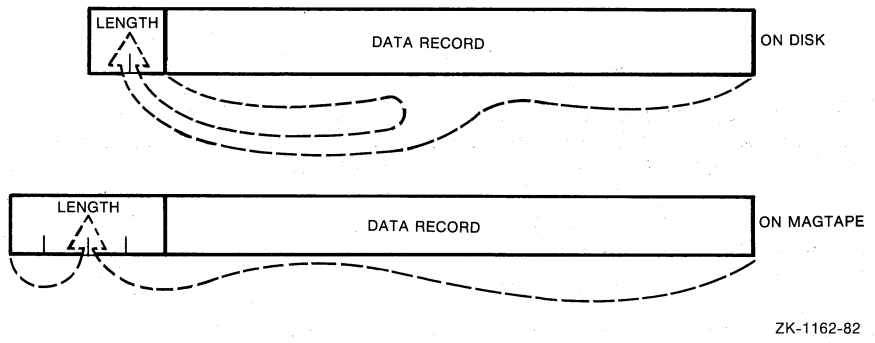


Figure 2-4: Count Field on Disk and Tape

Choose the variable-length record format if:

- The data truly varies in length, because the format adds the length field to each record's size.
- You are designing a new application where future uses may require records to change length.

NOTE

Changing a record's size during an UPDATE operation is restricted by file organization. See Sections 3.5.1, 4.5.1, and 7.2 for more information on using the UPDATE operation with the specific file organizations.

RMS-11 limits variable-length block-spanning records on disk to 32,763 bytes because of the count field. RMS-11 allows records to reach this maximum only in sequential files; other file organizations place further restrictions on record size. The minimum valid record is 2 bytes of zeroes, representing a null record.

2.4.1.3 Variable-with-Fixed-Control Format - A VFC record consists of two areas:

- A fixed-length control area from 1 to 255 bytes long; the length is maintained as a file attribute.
- A variable-length area that can vary in length from zero bytes to the maximum record size stored as a file attribute.

For each record, RMS-11 maintains a count field specifying the number of data bytes in the record including fixed and variable areas. The size of this field is a 2-byte binary count that does not include the 2 bytes for the field.

RMS-11 limits VFC block-spanning records to 32,763 bytes because of the count field. The minimum valid record is 3 bytes: the length field plus the minimum fixed area of 1 byte. The maximum variable area is the difference between 32,763 and the length of the fixed area.

2.4.1.4 Stream Format - A stream record consists of a series of contiguous bytes. RMS-11 detects the end of a stream record only by the presence of one of the following terminators:

Form feed (014 octal)
Line feed (012 octal)
Vertical tab (013 octal)

RMS-11 limits stream format to disk sequential files. In addition, the format causes the most CPU overhead because RMS-11 must examine each record character by character for the terminator.

During record operations, RMS-11 processes stream records as follows:

- For FIND and GET operations, RMS-11 scans the stream of bytes, removing leading NULL (000) characters and searching for the first occurrence of one of the terminators. If it finds a form feed, vertical tab, or line feed, RMS-11 includes the terminator character with the record and considers the record complete.

If it finds a carriage return, RMS-11 checks the character following the carriage return. If the next character is a line feed, RMS-11 discards both characters (carriage return and line feed) and considers the record complete. Otherwise, RMS-11 includes the carriage-return character in the record and resumes its search for a terminator.

During a GET operation, RMS-11 moves each character included in the record into the user buffer as it scans the stream of bytes. RMS-11 does not move any data into the user buffer during a FIND operation.

- For PUT and UPDATE operations, RMS-11 checks the last character of the record in the user buffer. If it finds a line feed, vertical tab, or form feed, RMS-11 moves the record as it is to the I/O buffer. If it does not find one of these terminators, RMS-11 moves the record to the I/O buffer and adds a carriage-return/line-feed pair to the end of the record.

2.4.1.5 Undefined Format - The undefined format means that RMS-11 reads only blocks, not records. Your program must interpret the contents of each block.

2.4.2 I/O Techniques

RMS-11 supports the following I/O techniques so you can adjust the performance of record operations:

- Asynchronous record operations -- When operating asynchronously, your program may regain control before the operation is completed; that is, the program will continue processing while the operation is being performed. This may improve processing time.

- Multiple access streams -- A stream can handle only one record at a time, but you can connect more than one access stream to a relative or indexed file if you want to:

  - Process more than one record in a file at a time with asynchronous record operations.

  - Maintain more than one context during the processing of a file.

  Each stream represents an independent, concurrently active sequence of record operations.

- Deferred write -- Normally, every write-type record operation to a relative or indexed file results in a physical I/O operation. However, you can sometimes have RMS-11 defer this write function until the I/O buffer is full or must be used for another bucket. Deferred write is the normal mode of I/O for sequential files.

- Multiblock count -- You can open a disk sequential file so that RMS-11 reads or writes more than one block of the file into the I/O buffer at a time. This capability speeds file processing, though the buffer gets bigger. For relative and indexed files, you achieve a similar effect by increasing bucket sizes.

- Multiple buffers -- You can allocate I/O buffers for a relative or indexed file beyond RMS-11's minimum requirements: one for relative; two for indexed. If the file is not accessed for read/write sharing, RMS-11 uses the buffers to save in memory, or cache, buckets from the file, so that they do not have to be read again from disk if needed.

  For indexed files, RMS-11 caches the root buckets from indexes that are used, saving one I/O operation on every random record operation. However, for relative files, RMS-11 makes no distinction between buckets, saving them until it has to use the buffer.

- Mass insertion -- Specified before the insertion of a series of records already sorted in ascending order by primary key, this mode enables RMS-11 to store the records tightly and quickly in the file. Records can be mass inserted only at the logical end of an indexed file. Mass insertion significantly improves performance for single-key indexed files. However, with each additional key defined for the file, the percentage improvement is smaller.

CHAPTER 3

SEQUENTIAL FILE APPLICATIONS

This chapter discusses sequential file structure, design, and processing. Sequential file design consists generally of determining the specific attributes, including record size and format, that will allow you to store, retrieve, and process your data efficiently within the sequential file structure. Your task design, along with your file design, will determine your record and file processing options, including record access modes.

3.1 FILE STRUCTURE

Physical Structure -- Sequential files carry almost no RMS-11 overhead. The operating system's file management software stores attributes in the file directory. RMS-11 stores data records beginning with virtual block number (VBN) 1.

- If records cross block boundaries (span blocks), RMS-11 packs records into the file end-to-end, allowing for control information and padding.
- If you do not allow records to span blocks, RMS-11 packs records into each block, allowing for control information and padding.

NOTE

You will waste space in your file if both of the following are true:

- You do not allow records to span blocks.
- Your records do not exactly fit into a block.

To be compatible with other file management systems, RMS-11 flags space that is not used at the end of each block. When you allow records to span blocks, the only unused space starts after the last record in the file. Table 3-1 lists the end-of-block indicators.

Table 3-1: End-of-Block Indicators

Medium	Record Format	End-of-Block Indicator
Disk	All but stream	-1 in word following last valid byte
Disk	Stream	nulls (000) to end of file
Magtape	All	circumflex (^) to end of block

For disk sequential files, RMS-11 uses the end-of-file attribute, stored in the file directory, to determine where the valid data in a file ends. This attribute includes a VBN and a byte offset within this block. The virtual block containing the logical end-of-file may not be the last block allocated to the file.

RMS-11 reads the end-of-file attribute with the other file attributes when it opens a file. RMS-11 also updates the end-of-file in the file directory when it closes the file if the end-of-file changed while the file was open. The end-of-file changes if records were added to the end of the file or if the file was truncated.

Conceptual Structure -- In most cases, RMS-11 stores records in the sequence that programs write them, one after the other from the first record in the file to the last. For these files, RMS-11 can only access the records sequentially or, for disk files, randomly by record file address (RFA).

The exception to this structure is the case of disk sequential files with fixed-length record format. In this case, RMS-11 stores records in a series of fixed-length cells; this is similar to relative file organization (see Chapter 4). The cell size is the size of the fixed-length record. Only one record can be put into a cell, and RMS-11 assumes that each cell contains a record. RMS-11 numbers the cells consecutively from 1 to n, where n indicates the last cell in the file. A cell number indicates the location of the cell relative to the beginning of the file, and is associated with the record as a relative record number (RRN).

RMS-11 can access records in a fixed-format disk sequential file sequentially, randomly by RFA, or randomly by key (RRN).

NOTE

RMS-11 does not initialize the cells in a fixed-format disk sequential file, nor does it "know" whether a cell contains a valid record. Your application program must maintain this information.

3.2 RECORD SIZE

Records in disk sequential files are word aligned, which means that RMS-11 adds a pad byte to the end of any record with an odd number of bytes. RMS-11 uses this convention to maintain structural compatibility with FCS-11 sequential files.

You can define a sequential file so that RMS-11 writes records across the boundaries between blocks. Such a sequential file is optimally dense; all bytes within its allocated space are used, except at the end of the file where no data has been written.

Table 3-2 shows the maximum data size for records in a sequential file. They are adjusted for RMS-11 restrictions and overhead.

Table 3-2: Sequential File Data Sizes (in bytes)

Format	Maximum Size		Data Size Calculation
	With Block Spanning	Without Block Spanning	
Fixed	32,766	512	Your data + MOD(DS/2)[1]
Variable	32,765	510	Your data + 2 + MOD(DS/2)[1]
VFC	32,765	509	Fixed + variable + 2 + MOD(DS/2)[1]
Stream	None	511[2]	Data + terminator(s)

1.MOD(DS/2) is the remainder after the size of your data (DS) in bytes is divided by 2:

- MOD(DS/2) = 0 if the data size is an even number of bytes.
- MOD(DS/2) = 1 if the data size is an odd number of bytes.

For VFC, DS = fixed + variable

2. Assuming a 1-byte terminator character; however, if the terminator is CR-LF, then the maximum length without block-spanning records is 510 bytes. Note that these figures do not include the terminator characters.

3.3 FILE DESIGN

For sequential files, the primary design considerations are:

- Record format (see Section 2.4.1 for a description of the RMS-11 record formats)
- Data storage medium
- File allocation
- Contiguity

3.3.1 Data Storage Medium

Sequential files can be accessed on both disk and magnetic tape. When you select the medium for your file, you should consider the following:

- Speed of access -- How long can each record operation take? Tape is significantly slower than disk.
- Frequency of use -- How often do you use the file? If you use it once a month, a quarter, and so on, you could store the file on tape and save your disk for more immediate purposes.

- Transportability -- Do you need to use the file on different operating systems? RSTS/E disk structure is not compatible with IAS, RSX-11M/M-PLUS, or VAX/VMS, and vice-versa. If you need to use the file across these systems, you should consider using a magnetic tape file.

3.3.2 File Allocation

Disk file allocation involves two quantities:

- Initial allocation quantity -- the number of blocks assigned to a file when you create it.
- Default extension quantity -- the number of blocks added to a file each time RMS-11 automatically extends it.

3.3.2.1 Initial Allocation - Even with sequential files, where a file extension requires only an allocation of blocks by the operating system, total allocation of the file when you create it is much more efficient.

You calculate the allocation (ALQ), in blocks, for block-spanning records as follows:

```
ALQ = (NRF*RSZ)/512
```

where:

ALQ is the allocation quantity in blocks

NRF is the largest number of records that will be in the file at one time

RSZ is the size of the record in bytes

For variable-length or VFC records, use the average record size for RSZ, including 2 bytes for the count field.

For fixed-length records, use the actual record size for RSZ.

Be sure to round RSZ up to a multiple of 2 to account for word alignment.

This allocation can be done by RMSDES or by your application program, depending on the capabilities of your high-level language.

3.3.2.2 Default Extension Quantity - If the file cannot be totally allocated at creation time, you should establish a reasonable default extension quantity (DEQ) to minimize the number of (and the time spent on) file extensions. Even if the file is totally allocated when you create it, you should establish a reasonable DEQ in case the file gets bigger than planned. The time required for each file extension is significant, involving:

- A call to the file control processor
- Possible I/O operations to bring file control processor routines into memory

- I/O operations to read and change file directory information
- I/O operations to read and change the disk free-block bit map

A good basis for calculation is the number of records added to the file in a given period of time, such as a day; use the formula for allocation quantity in Section 3.3.2.1.

The DEQ can be set by RMSDES or by your application program, depending on the capabilities of your high-level language. If you do not specify a DEQ, it will default to zero whether you create the file with RMSDES or a high-level language. This means that RMS-11 will extend the file according to the operating system default for file extensions.

Example: You are inserting 1000 50-byte fixed-length records into a sequential file. Records do not span blocks; therefore, each block contains 10 records. The file is currently full; that is, no more records can be added without an extension.

- If DEQ is zero, RMS-11 extends the file according to operating system defaults, which are typically only a few blocks. Therefore, in this example, if the system default is 5 blocks, RMS-11 extends the file 20 times.
- If DEQ is 1, RMS-11 extends the file for every tenth PUT operation after the first, for a total of 100 extensions.
- If DEQ is 25, RMS-11 extends the file 4 times.
- If DEQ is 100 or more, RMS-11 extends the file only once.

3.3.3 Contiguity

Contiguity can significantly affect performance. Therefore, you should consider contiguity for a disk sequential file to minimize the time spent on each I/O operation.

If the blocks in a file are not contiguous, they may be on different parts of the disk, and thus require significant head movement to access the file contents.

Physical contiguity, however, ensures that the file is stored on one track or, at worst, adjacent tracks. Because the disk can read a track without moving the heads, file contiguity reduces head movement. This assumes that no other software is accessing the disk at the same time.

Contiguity also enhances virtual-to-logical-block mapping (see Chapter 8).

To ensure that the blocks in the file are physically contiguous, allocate the whole file when you create it (see Section 3.3.2.1) and specify that the allocation be performed contiguously.

3.4 ACCESS SHARING

Access sharing can be specified for disk sequential files, as described in the following sections. See Section 2.2.3 for general information on shared access.

3.4.1 Record Access to Sequential Files

Because of their internal structure, record-structured sequential files are not read/write sharable in the manner of relative and indexed files. Thus, a read/write sharing declaration for such a file is converted internally to a read-only sharing declaration before the file is processed.

As a result, multiple read-only accessors who have specified no-sharing, read-only sharing, or read/write sharing can access such a file concurrently as long as no read/write accessor is present; or a single accessor who has specified no-sharing, read-only sharing, or read/write sharing can access such a file as long as no other accessor of any kind is present. Other combinations are rejected: the access and sharing declarations are incompatible.

Limited sequential file sharing is possible, however, in the case of a single read/write accessor in combination with multiple read-only accessors, when the application programs involved (rather than RMS-11) can take responsibility for any interlocking required.

In this case, the read-only accessors must specify sharing with user-provided interlocks to gain access; the sharing declaration of the single read/write accessor is immaterial. Each read-only accessor cannot read beyond the logical end-of-file mark that existed at the time that accessor opened the file, and must recognize that inconsistent data may be returned if the single read/write accessor modifies data within the accessible portion of the file.

3.4.2 Block Access to Sequential Files

Sequential files can be read/write shared using block access, but for those accessors who specify read/write sharing, automatic file extensions will not occur and the logical end-of-file mark in the file header will be neither respected nor updated. (Again, this is because of the internal structure of sequential files.) Such read/write sharing uses the operating system's block-locking facilities to coordinate shared access.

Sequential files can also be shared in a noninterlocked manner, with user-provided interlocks. Because of operating system restrictions, the single read/write accessor must specify no-sharing or sharing with user-provided interlocks, and multiple read-only accessors must specify sharing with user-provided interlocks. These restrictions also prohibit concurrent access to the file by read/write-sharing accessors or an accessor who specified read-only sharing and read/write access.

When no write accessor is present, sequential files can be shared among multiple read-only accessors who have specified no sharing or read-only sharing.

3.5 RECORD AND FILE PROCESSING OF SEQUENTIAL FILES

The record and file processing capabilities described in RSX-11M/M-PLUS RMS-11: An Introduction are available for sequential files. This section discusses the operations and their implementation and restrictions with sequential files.

3.5.1 Record and Stream Operations

The following record and stream operations can be performed on sequential files:

CONNECT
DISCONNECT
FIND
FLUSH
GET
PUT
REWIND
TRUNCATE
UPDATE

In all record operations, RMS-11 establishes the current record context (if any) and the next record context (if applicable). If any record operation fails, RMS-11 normally sets the current record context to none and does not change the next record context.

NOTE

For more information on the RMS-11 error codes referred to in the following sections, see the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide.

3.5.1.1 CONNECT - A CONNECT operation affects the record context for the access stream as follows:

- Current record -- There is no current record. Any operation requiring a current record fails at this point.
- Next record -- If you did not specify that you were going to append records to the file, the next record is the first record in the file.

  If you did specify that you were going to append records to the file, the next record is the end-of-file.

3.5.1.2 DISCONNECT - A DISCONNECT operation destroys the current record context for the access stream. You cannot resume this context by reconnecting the stream.

3.5.1.3 FIND - To perform a FIND operation on a sequential file, RMS-11:

1. Determines the location of the record in the file according to the specified record access mode:
   - In sequential-access mode, location is indicated by the next record pointer.
   - In key-access mode, location is determined by the specified relative record number and match criterion. (This access mode is available for fixed-format disk sequential files only.)

- In RFA-access mode, location is determined by the specified RFA. (This access mode is available for disk files only.)

2. Reads the block containing the record, or the first part of the record if the record spans blocks, from disk into the task's I/O buffer, if it is not already in memory. The block may be in memory if the block was required by a previous operation.

3. For disk files, returns the RFA to the program, but does not transfer the record to the program's user buffer.

4. Returns the RRN for fixed-format disk sequential files.

If no valid record exists in the location specified, the response depends on the access mode:

- In sequential-access mode, the error code is ER$EOF, meaning that no record was located because there are no more records in the file.

- In RFA-access mode, the error code is either ER$RFA, if the RFA had an invalid format, or ER$EOF, if the RFA specified a location beyond the end of the file.

- In key-access mode for fixed-format disk sequential files, the error code is ER$KEY, if the key value had an invalid format, or ER$EOF, if the key value specified a location beyond the end of the file.

A FIND operation affects the record context for the access stream as follows:

- For a sequential-access FIND operation:

  - Current record is set to value of the record found, that is, the next record before the FIND operation started.

    Example: You have connected a stream to a sequential file without specifying that records will be appended to the file (see Section 3.5.1.1). There is no current record, but the next record is the first record in the file. If you execute a sequential FIND operation, the current record is set to the first record in the file.

  - Next record is set to the record virtually following the current record.

    Example: From the previous example, the next record is the second record in the file.

- For an RFA-access or key-access FIND operation:

  - Current record is set to the record found, that is, the record identified by the RFA or RRN.

  - Next record is unchanged.

Example: In the preceding example, you performed a sequential-access FIND operation after connecting the stream to the file. You now execute a FIND by RFA. The current record is set to the record specified, but the next record is not changed. Therefore, when you perform another sequential FIND operation, the current record is set to the second record in the file, not to the record following the one found by RFA.

You use a FIND operation instead of a GET operation for two reasons:

1. It is faster because the record is not moved to the user buffer. Although the time required to move a record from one part of memory to another is very short, do not expend it unnecessarily.

2. It does not change the next record in RFA or key access mode. This convention allows you to branch off sequential processing for updating or deleting, and keep your place in the file.

You can use a FIND operation in the following ways:

- To skip records in sequential access mode by initiating successive FIND operations.

- To establish a random starting point using RFA or key access mode. You could then initiate successive GET operations, where the first operation gets the record found by RFA or by RRN.

- To establish a current record for an UPDATE or TRUNCATE operation.

- To determine whether a record cell specified by RRN exists in a file (for fixed-format disk sequential files only).

3.5.1.4 FLUSH - A FLUSH operation does not affect the record context for the access stream.

3.5.1.5 GET - To perform a GET operation on a sequential file, RMS-11:

1. Determines the location of the record in the file according to the specified access mode:

   - In sequential-access mode, location is indicated by the next record pointer, if the get operation was not immediately preceded by a successful FIND operation, or the current record pointer set by an immediately preceding successful FIND operation.

   - Location is determined by the specified relative record number and match criterion in key-access mode (fixed-format disk sequential files only).

   - Location is determined by the specified RFA in RFA-access mode (disk files only).

2. Reads the block containing the record, or the first part of the record if the record spans blocks, from disk into the task's I/O buffer, if the block is not already in memory.

Example: Your records are 50 bytes long. When you read sequentially through the file, RMS-11 must request a disk I/O operation for every tenth GET operation that your program executes.

3. For disk files, returns the RFA to the program and moves the record from the I/O buffer to the specified user buffer in the program unless the program is operating in locate record transfer mode (see Section 3.5.2). If the buffer does not contain the entire record, RMS-11 reads more blocks into the I/O buffer and assembles the record in the program's user buffer, regardless of record transfer mode.

4. Returns the RRN for fixed-format disk sequential files.

If no valid record exists in the location specified, the response depends on the access mode:

- In sequential-access mode, the error code is ER$EOF, meaning that no record was located because there are no more records in the file.

- In RFA-access mode, the error code is either ER$RFA, if the RFA had an invalid format, or ER$EOF, if the RFA specified a location beyond the end of the file.

- In key-access mode for fixed-format disk sequential files, the error code is ER$KEY, if the key value had an invalid format, or ER$EOF, if the key value specified a location beyond the end of the file.

A GET operation affects the current record context for the access stream as follows:

- Current record is set to the record read.

- Next record is set to the record virtually following the current record.

Example: You have connected a stream to a sequential file without specifying that records will be appended to the file (see Section 3.5.1.1). There is no current record, but the next record is the first record in the file. If you execute a sequential-access GET operation, the current record is set to the first record in the file and the next record is the second record in the file.

3.5.1.6 PUT - To perform a PUT operation on a sequential file, RMS-11:

1. Determines whether the specified access mode is allowed. Sequential-access mode must be specified unless the file is a fixed-format disk file; in that case, key-access mode is allowed. RMS-11 returns the error code ER$RAC if an illegal access mode is specified.

2. Determines the destination of the record in the file according to the specified access mode:

   - In sequential-access mode, the next record pointer indicates the destination. The destination must be the end-of-file; if it is not, RMS-11 returns the error code ER$NEF.

Your program gets to the end of a sequential file by:

- Specifying that records will be appended to the file when the program connects the record access stream to the file (see Section 3.5.1.1).

- Initiating sequential FIND and/or GET operations until RMS-11 returns an ER$EOF error code.

- In key-access mode, the specified relative record number indicates the destination. Note that RMS-11 does not check the validity of the designated RRN: if the destination block is beyond the current end-of-file, RMS-11 will extend the file to the destination block.

3. Reads the destination block in the file into the I/O buffer, if the block is not already in memory. The block may be in memory if it was required by a previous operation.

4. Moves the record from the user buffer to the task's I/O buffer.

5. Writes the I/O buffer to disk only if the buffer is full. If there is no room for the block(s) in the file, RMS-11 extends the file (see Section 3.3.2) and then writes the buffer to disk.

6. For disk files, returns the RFA to the program.

7. Returns the RRN for fixed-format disk sequential files.

A PUT operation affects the context for the access stream as follows:

- For a sequential-access PUT operation:

  - Current record -- None. Any operation requiring a current record fails at this point.

  - Next record -- End-of-file. A sequential FIND or GET operation fails with error code ER$EOF.

- For a key-access PUT operation:

  - Current record -- None. Any operation requiring a current record fails at this point.

  - Next record -- Unchanged.

3.5.1.7 REWIND - A REWIND operation affects the record context for the access stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.

- Next record -- Set to the first record in the file.

3.5.1.8 TRUNCATE - A TRUNCATE operation declares end-of-file at the position of the current record. In doing so, the operation effectively deletes the current record and all records in the sequential file following that record.

The TRUNCATE operation requires a valid current record. It therefore should follow a successful GET or FIND operation; otherwise, RMS-11 returns the error code ER$CUR.

A TRUNCATE operation affects the context for the access stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.
- Next record -- End-of-file.

After a TRUNCATE operation, you can immediately add records to the file using PUT operations.

NOTE

The TRUNCATE operation does not reduce the actual allocated size of a sequential file on a disk: it merely specifies a new logical end-of-file mark.

3.5.1.9 UPDATE - In an UPDATE operation, RMS-11 moves the specified record from the task's user buffer to the I/O buffer, replacing the current record set by a previous GET or FIND operation. However, RMS-11 does not immediately write the buffer to the file. RMS-11 requests the file control processor to write the changed buffer over its original location on the disk only when the buffer must be replaced in memory by another operation.

Example: You get a record by RFA and update it. Then, you get another record by RFA. RMS-11 writes the buffer containing the first record you updated only when it must replace the data in the buffer to satisfy the second GET operation.

UPDATE operations have the following restrictions:

- The operation is valid only on disk sequential files. If you attempt it on magnetic tape files or unit record devices, RMS-11 returns the error code ER$IOP.
- The operation requires a valid current record. It therefore should follow a successful GET or FIND operation; otherwise, RMS-11 returns the error ER$CUR.
- The size of the record cannot change during an UPDATE operation. If it changes, RMS-11 returns the error code ER$RSZ.
- You cannot update stream records. If you attempt it, RMS-11 returns the error code ER$RFM.

None of these errors affects the original record in the file on disk.

An UPDATE operation affects the context for the access stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.
- Next record -- Unchanged.

3.5.2 Record Transfer Modes

You can manipulate records either in the I/O buffer or in your program's user buffer. Each of these options is called a record transfer mode. You can change record transfer mode at run time, even between record operations.

Figure 3-1 shows the I/O and user buffers.

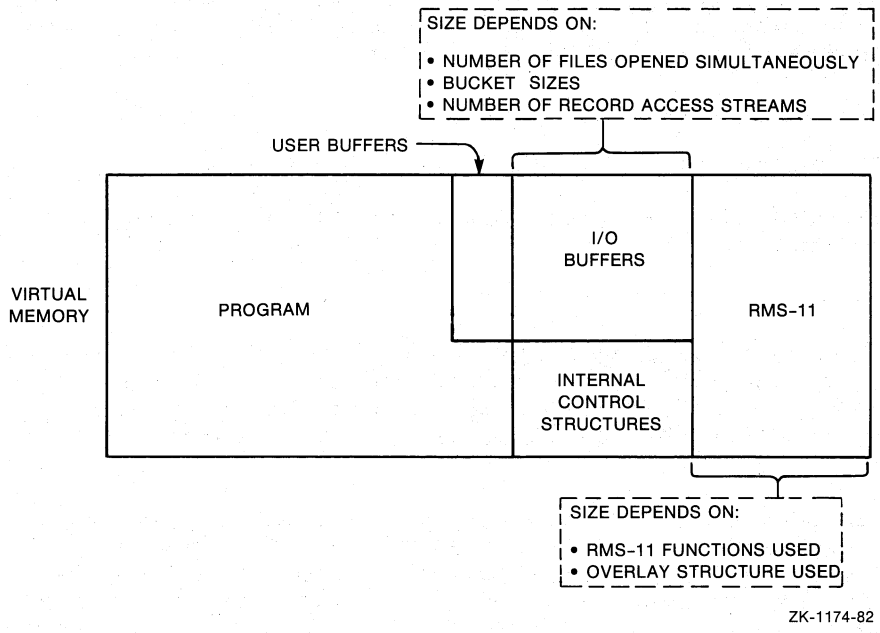


Figure 3-1: RMS-11 Task Structure

3.5.2.1 Move Mode - Move mode requires that each record be copied between the user and I/O buffers:

- On GET operations, RMS-11 moves the record from the I/O buffer to the user buffer before returning control to your program.
- On PUT and UPDATE operations, your program assembles the record to be written into the file in the user buffer. During the operation, RMS-11 moves the data into the I/O buffer before updating the file.

Move mode is the default record transfer mode for all programming languages and all file organizations.

3.5.2.2 Locate Mode - Locate mode enables your program to manipulate records in the I/O buffer, eliminating the data transfers between it and the user buffer. However, when you specify locate mode, RMS-11 uses it only when such usage does not compromise data integrity. Otherwise, RMS-11 uses move mode. Therefore, your program must still contain a user buffer.

Example: RMS-11 uses move mode instead of locate mode when records span buffers in a sequential file.

Example: RMS-11 uses move mode instead of locate mode if you opened the file indicating that you were going to perform UPDATE operations on it.

RMS-11's use of move mode instead of locate mode is transparent to your program as long as you use RMS-11 facilities to access the record data.

For sequential files, your program can both performs both GET and PUT operations in locate mode. See your high-level language documentation to determine whether the language supports locate mode and, if it does, what the programming techniques are.

3.5.3 I/O Techniques

You can use the following techniques to improve the performance of record operations.

3.5.3.1 Asynchronous Record Operations - Within each access stream, your program can perform any record operation either synchronously or asynchronously. In synchronous operations, RMS-11 returns control to your program after the operation ends, either successfully or with an error.

When you execute an asynchronous operation, RMS-11 may return control to your program before the operation is complete. The program continues processing while the physical transfer of data between disk and memory is carried out. However, you must not initiate another record operation on that stream until the first operation ends; otherwise, RMS-11 returns the error code ER$ACT. See your high-level language documentation for asynchronous techniques.

3.5.3.2 Deferred Write - The normal mode of operation for sequential files is similar to operations using deferred write with the other file organizations (see Chapters 4 and 7). Using this technique for sequential files does not change or improve performance.

3.5.3.3 Multiple Buffers - The multiple buffer capability is not available to sequential files.

3.5.3.4 Multiple Access Streams - RMS-11 allows each program to use only one stream on a sequential file because sequential files are not formatted to permit simple and economical sharing (see Section 3.4).

3.5.3.5 Multiblock Count - Your task can be set up so that more than one block from a disk sequential file is read or written at one time. This multiple-block I/O can improve processing because it tends to reduce the number of physical I/O operations. However, it also increases the size of the task, on a one-for-one basis; that is, for each increment of the multiblock count (MBC), the I/O buffer in the task grows by 512 bytes.

An MBC greater than 1 is therefore useful for sequential processing, including file population.

Example: You are using 50-byte records. During sequential processing, if the MBC is 1, RMS-11 requests a disk I/O operation for every tenth record operation your program executes, whether the operations are GET or PUT operations. If you set MBC to 5 for instance, RMS-11 requests a physical I/O operation for every 50 record operations.

3.5.4 File and Directory Operations

The following file and directory operations can be performed on sequential files:

CLOSE
CREATE
DISPLAY
ENTER
ERASE
EXTEND
OPEN
PARSE
REMOVE
RENAME
SEARCH

See your high-level language documentation for a description of the support provided.

CHAPTER 4

RELATIVE FILE APPLICATIONS

This chapter discusses relative file structure, design, and processing. Relative file design consists generally of determining the specific attributes, including record size and format, that will allow you to store, retrieve, and process your data efficently within the relative file structure. Your task design, along with your file design, will determine your record and file processing options, including record access modes.

4.1 FILE STRUCTURE

Physical Structure -- Relative files contain at least one block of RMS-11 information known as the prologue. The operating system's file management software stores attributes in the file directory. RMS-11 stores the prologue in VBN 1 -- unless the bucket size is 2, 4, or 8 blocks. In that case, RMS-11 makes the prologue equal to 1 bucket in size. Data records begin in the block following the prologue.

RMS-11 allocates relative files in bucket increments. The first bucket begins with the first data block. To support deleted record control, RMS-11 initializes each bucket (sets all bits to 0) when it allocates the blocks to the file.

The fixed-length cells are set up in each bucket starting with byte 0 and packed end-to-end, byte-aligned, until no more cells can fit in the bucket (no padding necessary). Cells cannot span bucket boundaries, although they can cross block boundaries within multiblock buckets. The first byte of each cell is used by RMS-11 to provide deleted record control.

Conceptual Structure -- RMS-11 stores records in a series of fixed-size cells. Only one record can be put into a cell, but all cells do not have to contain records. The cell size is based on the length you specify as the maximum for any record in the file. RMS-11 numbers the cells consecutively from 1 to **n**, where **n** indicates the last cell in the file. A cell number relates its location to the beginning of the file and is associated with the record in the cell, if any, as a relative record number.

RMS-11 can access records in a relative file either sequentially or randomly, both by relative record number (key) and by RFA.

4.2 RECORD SIZE

RMS-11 calculates the number of bytes in each record cell in the file (CL) of a relative record cell as follows:

CL = 1+RFO+DS+FSZ

where:

1 is a byte for RMS-11 overhead

RFO is bytes for record format overhead: 0 for fixed; 2 for variable or VFC

FSZ is the fixed control size for VFC format; 0 for other formats

DS is bytes of data

For variable-length or VFC record format, DS is the maximum record size set for the file.

Table 4-1 shows the maximum data sizes for records in a relative file. These are the sizes of your data; they are adjusted for RMS-11 restrictions and overhead.

Table 4-1: Relative File Data Sizes (in bytes)

Format	Maximum Size	Record Cell Size Calculation
Fixed	16,383	Data size + 1
Variable	16,381	Maximum record size + 3
VFC	16,381	Fixed + variable + 3

4.3 FILE DESIGN

For relative files, the primary design considerations are:

- Record format (See Section 2.4.1 for a description of the RMS-11 record formats)
- Bucket size
- File allocation
- Contiguity
- Maximum record number

4.3.1 Bucket Size

Buckets are the I/O units for relative files. Their size is therefore critical to the space required by a task and the speed with which the task performs. Sequential access, especially, benefits when there are multiple records per bucket. There is, of course, a trade-off: the larger the bucket size, the larger the task, but the faster the task reads data sequentially:

- Each block added to the bucket size increases the task size by 512 bytes for each access stream.

- The speed of an RMS-11 operation is closely proportional to the number of I/O operations involved. RMS-11 requests an I/O operation each time it requires a new bucket to locate a record. Therefore, the more record cells in a bucket, the fewer I/O operations RMS-11 needs to read a file sequentially.

  However, write sharing a relative file counteracts this optimization if your program has read-only access to the file. RMS-11 reads a bucket from disk during each GET operation -- even if the next record is in the bucket in memory -- because the bucket is not locked after each GET operation and a writing program may have changed the bucket since the record was last read.

Bucket size can be set by RMSDES or by your application program depending on the capabilities of your high-level language.

4.3.2 File Allocation

File allocation involves two quantities:

- Initial allocation quantity -- The number of blocks assigned to a file when you create it
- Default extension quantity -- The number of blocks added to a file each time RMS-11 automatically extends it

4.3.2.1 Initial Allocation - Total allocation of a file when you create it is the most efficient technique regardless of file organization, but with relative files initial allocation becomes most critical. Each allocation, whether at creation time or during an extension, requires RMS-11 to initialize the new buckets by setting all bits to zero. You can avoid time-consuming file extensions during normal processing by totally allocating the file when you create it or by explicitly extending the file when it is not being used for processing.

You calculate the allocation (ALQ), in blocks, as follows:

```
ALQ = PLG+(NRF/NRBKT)*BKS
```

where:

PLG is equal to 1 block or to BKS if BKS is 2, 4, or 8

NRF is equal to the maximum record number (MRN) or to the number of records that will be written into the file

BKS is the bucket size in blocks

NRBKT is the number of records in a bucket

You calculate NRBKT as follows:

```
NRBKT = (512*BKS)/(RSZ+RFO)
```

where:

RSZ is the size of the record in bytes:

- Data size for fixed-length records
- Maximum record length for variable-length records
- Size of the fixed-length control area plus the maximum size of the variable-length area for VFC records

RFO is the record format overhead:

- RFO = 1 byte for fixed-length records
- RFO = 3 bytes for variable-length and VFC records

This allocation can be done during file creation by RMSDES or by your application program, depending on the capabilities of your high-level language.

The allocation can also be done by using a PUT operation to write the "last record" into the file first; that is, the record whose relative record number is equal to the maximum record number (MRN). Before RMS-11 can write this record, it must allocate all record cells from 1 to MRN and initialize the new blocks. After the PUT operation, the relative file will be completely allocated.

4.3.2.2 Default Extension Quantity - If the file cannot be totally allocated at creation time, you should establish a reasonable default extension quantity (DEQ) to minimize the number of (and the time spent on) file extensions. Even if the file is totally allocated when you create it, you should establish a reasonable DEQ in case the file must become bigger than planned.

A good basis for calculation is the number of records that are added to the end of the file in a given time period, such as a day; use the formula for allocation quantity in Section 4.3.2.1.

The DEQ for the file can be set by RMSDES or by your application program, depending on the capabilities of your high-level language.

If you do not specify a DEQ, it defaults to zero. RMS-11 responds to a DEQ of zero by requesting 4 times the bucket size in blocks from the file control processor each time it automatically extends the file.

4.3.3 Contiguity

Contiguity can significantly affect performance. Therefore, you should consider contiguity for a relative file to minimize the time spent on each I/O operation. If the blocks in a file are not contiguous, they may be on different parts of the disk and thus require significant head movement to access the file contents.

Physical contiguity, however, ensures that the file is stored on a single track or, at worst, adjacent tracks. Because the disk can read an entire track without moving the heads, file contiguity reduces head movement. This assumes that no other software is accessing the disk at the same time.

Contiguity also enhances virtual-to-logical-block mapping (discussed in Chapter 8).

To ensure that the blocks in the file are physically contiguous, allocate the whole file when you create it (see Section 4.3.2.1).

4.3.4 Maximum Record Number

The MRN associated with a relative file limits the size of the file. RMS-11 will not put a record into a file with a relative record number greater than the assigned MRN. However, if an MRN is not set (that is, MRN is zero), RMS-11 only checks whether the record number is greater than zero before attempting to store a record in a relative file.

MRN determines the maximum useful size of a file because RMS-11 allocates a record cell for each record between relative record number 1 and the highest relative record number used. You can explicitly make the file larger than this maximum, but RMS-11 will not use the space. The actual size can be smaller than the size that would be set if a record with the MRN were written into the file.

You can calculate the file size (FSZ) in blocks from the largest relative record number actually present in the file:

```
FSZ = PLG+1+((LRN-1)/((BKS*512)/(RSZ+RFO)))
```

where:

PLG is the size of the prologue: BKS if BKS = 2, 4, or 8; otherwise, 1

LRN is the largest RRN actually present in the file

BKS is the bucket size in blocks

RSZ is the size of the record in bytes:

- Data size for fixed-length records
- Maximum record length for variable-length records
- Size of the fixed-length control area plus the maximum size of the variable-length area for VFC records

RFO is the record format overhead:

- RFO = 1 byte for fixed-length records
- RFO = 3 bytes for variable-length and VFC records

MRN can be set by RMSDES or by your application program, depending on the capabilities of your high-level language.

4.4 ACCESS SHARING

Access sharing can be specified for relative files as described in the following sections. See Section 2.2.3 for general information on shared access.

4.4.1 Record Access to Relative Files

Relative files allow fully interlocked read/write sharing, dependent upon the compatibility of the access and sharing declarations of multiple accessors, as follows:

- If you have requested read/write access, your request will be denied unless **all** other accessors have allowed read/write sharing. (Otherwise, your read/write access request will conflict with the sharing declaration of at least one other accessor.)
- If you have not permitted read/write sharing, your request for read/write access will be denied if any other read/write accessor is present. (In this case, the read/write accessor does not meet the requirements of your sharing declaration.)

4.4.2 Block Access to Relative Files

Because block access bypasses the record structure and interlocking algorithms used with relative files, read/write sharing cannot be permitted. Any read/write sharing declaration is converted internally to read-only before the file is processed (this is similar to record-accessed sequential files).

Thus, multiple read-only accessors (regardless of their sharing declarations) can share relative files concurrently using block access, as long as no read/write record accessor is present. Read-only block accessors can share files with read-only record accessors. In addition, a single read/write accessor can access a relative file using block access (regardless of sharing declaration) as long as no other accessor of any kind is present.

Other combinations are rejected: the access and sharing declarations are incompatible.

4.5 RECORD AND FILE PROCESSING OF RELATIVE FILES

The record and file processing capabilities described in RSX-11M/M-PLUS RMS-11: An Introduction are available for relative files. This section discusses the operations and their implementation and restrictions with relative files.

4.5.1 Record and Stream Operations

The following record and stream operations can be performed on a relative file:

CONNECT
DELETE
DISCONNECT
FIND
FLUSH
GET
PUT
REWIND
UPDATE

In all record operations, RMS-11 establishes the current record context (if any) and next record context (if applicable). If any record operation fails, RMS-11 normally sets the current record context to none and does not change the next record context.

NOTE

For more information on the RMS-11 error codes referred to in the following sections, see the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide.

4.5.1.1 CONNECT - A CONNECT operation affects the current record context for the access stream as follows:

- Current record -- There is no current record. Any operation requiring a current record fails at this point.
- Next record -- The next record is the first record cell in the file.

4.5.1.2 DELETE - In a DELETE operation, RMS-11 flags the current record cell to indicate that it contains a deleted record. RMS-11 does this by setting the RMS-11 control byte in the cell to a certain value. Then, RMS-11 writes the bucket over its original location on the disk, unless you have specified deferred write (see Section 4.5.3.2).

A DELETE operation requires a valid current record. Therefore, a DELETE operation should follow a successful GET or FIND operation; otherwise, RMS-11 returns the error code ER$CUR. This error does not affect the original record in the file on disk.

A DELETE operation affects the current record context for the access stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.
- Next record -- Unchanged.

4.5.1.3 DISCONNECT - A DISCONNECT operation destroys the current record context for the access stream. You cannot resume this context by reconnecting the stream.

4.5.1.4 FIND - To perform a FIND operation on a relative file, RMS-11:

1. Determines the location of the record in the file according to the specified access mode:

   - In sequential-access mode, location is indicated by the next record pointer.
   - In key-access mode, location is determined by the specified relative record number and match criterion.
   - In RFA-access mode, location is determined by the specified RFA.

2. Reads the bucket containing the indicated cell from disk into the task's I/O buffer, if the bucket is not already in memory. The bucket may be in memory if it was required by a previous operation.

3. Returns the RFA and the RRN to the program, but does not transfer the record to the program's user buffer.

   If the cell is empty or contains a deleted record, the response depends on the access mode:

   - In sequential-access mode, RMS-11 repeats steps 1 through 3, moving through cells until the MRN is exceeded (ER$MRN) or the end of the file is reached (ER$EOF).
   - In key-access mode, RMS-11 reacts according to the specified match criterion:
     - On an equal match, RMS-11 returns the error code ER$RNF.
     - On a greater-than or greater-than-or-equal match, RMS-11 internally adds 1 to the relative record number and repeats steps 1 through 3, until either the MRN is exceeded (ER$MRN) or the end of the file is reached (ER$RNF).
   - In RFA-access mode, RMS-11 returns the appropriate error code:
     - ER$RNF -- No valid record has ever existed at the specified location.
     - ER$DEL -- The control byte in the cell indicates that the record in it was deleted.

A FIND operation affects the record context for the access stream as follows:

- For a sequential-access FIND operation:
  - Current record is set to the relative record number of the record found, that is, the next record before the FIND operation started.

Example: You have connected a stream to a relative file. There is no current record, but the next record is the first record in the file. If you execute a sequential-access FIND operation, the current record is set to the first record in the file.

- Next record is set to a relative record number 1 higher than the relative record number for the current record.

Example: From the previous example, the next record is the second record cell in the file.

- For a key-access or RFA-access FIND operation:
  - Current record is set to the record found, that is, the record identified by the relative record number or RFA.
  - Next record is unchanged.

Example: In the preceding examples, you performed a sequential-access FIND operation after connecting the stream to the file. You now execute an RFA-access FIND operation. The current record is set to the record specified, but the next record is not changed. Therefore, when you perform another sequential-access FIND, the search will begin in the second record cell in the file, not in the cell following the one found by RFA.

You use a FIND operation instead of a GET operation for two reasons:

1. It is faster because the record is not moved to the user buffer. Although the time required to move a record from one part of memory to another is very short, there is no use expending it if you do not need to.
2. It does not change the next record in key-access mode or RFA-access mode. This allows you to branch off sequential processing for purposes of updating or deleting records, and keep your place.

You can use a FIND operation in the following ways:

- To skip records in sequential-access mode by initiating successive FIND operations.
- To establish a random starting point for sequential processing using RFA-access mode. You could then initiate successive GET operations, where the first operation gets the record found by RFA.
- To establish a current record for a DELETE or UPDATE operation.
- To determine the existence of a record by using a random access mode.

4.5.1.5 FLUSH - A FLUSH operation does not affect the record context for the access stream.

4.5.1.6 GET - To perform a GET operation on a relative file, RMS-11:

1. Determines the location of the record in the file according to the specified access mode:

   - In sequential-access mode, location is indicated by: (a) the next record pointer, if the GET operation was not immediately preceded by a successful FIND operation; or (b) the current record pointer set by an immediately preceding FIND operation.
   - In key-access mode, location is determined by the specified relative record number and match criterion.
   - In RFA-access mode, location is determined by specified RFA.

2. Reads the bucket containing the indicated cell from disk into the task's I/O buffer, if the bucket is not already in memory. The bucket may be in memory if it was required by a previous operation.

   Example: Your fixed-length records are 50 bytes long; bucket size is 2 blocks. When you read sequentially through the file, RMS-11 must request a disk I/O operation every twentieth GET operation that your program executes.

   NOTE

   If you have opened a relative file with read-only access and read/write sharing declarations, each GET operation causes an I/O operation.

3. Returns the RFA and the RRN to the program and moves the record from the I/O buffer to the specified user buffer in the program -- unless the program is operating in locate record transfer mode (see Section 4.5.2.2).

   If the cell is empty or contains a deleted record, the response depends on the access mode:

   - In sequential-access mode, RMS-11 repeats steps 1 through 3, moving through cells until the MRN is exceeded (ER$MRN) or the end of the file is reached (ER$EOF).
   - In key-access mode, RMS-11 reacts according to the specified match criterion:
     - On an equal match, RMS-11 returns the error code ER$RNF.
     - On a greater-than or greater-than-or-equal match, RMS-11 internally adds 1 to the relative record number and repeats steps 1 through 3, until either the MRN is exceeded (ER$MRN) or the end of the file is reached (ER$RNF).

- In RFA-access mode, RMS-11 returns the appropriate error code:
  - ER$RNF -- No valid record has ever existed at the specified location.
  - ER$DEL -- The control byte in the cell indicates that the record in it was deleted.

A GET operation affects the record context for the access stream as follows:

- Current record is set to the relative record number of the record read.
- Next record is set to a relative record number 1 higher than the relative record number for current record.

4.5.1.7 **PUT** - To perform a PUT operation on a relative file, RMS-11:

1. Determines the destination of the record in the file according to the specified access mode:
   - In sequential-access mode, the next record pointer indicates the destination.
   - In key-access mode, the specified relative record number indicates the destination.
2. Determines whether the bucket containing the indicated cell is in the file. If it is, RMS-11 goes to the next step. If it is not, RMS-11 extends the file until it has enough blocks for all buckets up to and including the required one. Then, RMS-11 initializes all newly allocated buckets.
3. Reads the bucket containing the indicated cell from disk into the task's I/O buffer, if the bucket is not already in memory. The bucket may be in memory if it was required by a previous operation.
4. Checks the indicated cell: if it already contains an existing, valid record, RMS-11 returns error code ER$REX; otherwise, RMS-11 goes to the next step.

   Note that in some cases, you may be able to update an existing, valid record in a cell. See your high-level language documentation.
5. Moves the record from the user buffer in the program to the task's I/O buffer.
6. Returns the RFA and the RRN to the program.
7. Writes the I/O buffer to disk, unless you have specified deferred write (see Section 4.5.3.2).

A PUT operation affects the record context for the access stream as follows:

- For a sequential-access PUT operation:
  - Current record -- None. Any operation requiring a current record fails at this point.

  - Next record -- The cell with a relative record number 1 higher than the relative record number of the record just inserted.

- For a key-access PUT operation:

  - Current record -- None. Any operation requiring a current record fails at this point.

  - Next record -- Unchanged.

4.5.1.8 REWIND - A REWIND operation sets the context of the access stream to the beginning of the relative file. In doing so, it affects the record context for the stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.

- Next record -- Set to the first record cell in the file.

4.5.1.9 UPDATE - In an UPDATE operation, RMS-11 moves the specified record from the task's user buffer to the I/O buffer, replacing the current record set by a previous GET or FIND operation. Then, RMS-11 writes the bucket over its original location on the disk, unless you have specified deferred write (see Section 4.5.3.2).

An UPDATE operation requires a valid current record. Therefore, an UPDATE operation should follow a successful GET or FIND operation; otherwise, RMS-11 returns the error code ER$CUR. This error does not affect the original record in the file on disk.

An UPDATE operation affects the current record context for the access stream as follows:

- Current record -- None. Any operation requiring a current record will fail at this point.

- Next Record -- Unchanged.

4.5.2 Record Transfer Modes

You can manipulate records either in the I/O buffer or in your program's user buffer. Each of these options is called a record transfer mode. You can change record transfer mode at run time, even between record operations. Figure 4-1 illustrates the RMS-11 task structure.

4.5.2.1 Move Mode - Move mode requires that each record be copied between the user and I/O buffers:

- On GET operations, RMS-11 moves the record from the I/O buffer to the user buffer before returning control to your program.

- On PUT and UPDATE operations, your program assembles the record to be written into the file in the user buffer and, during the operations, RMS-11 moves the data into the I/O buffer before updating the file.

Move mode is the default record transfer mode for all programming languages and all file organizations.

4.5.2.2 Locate Mode - Locate mode enables your program to manipulate records in the I/O buffer, eliminating the data transfers between it and the user buffer. However, when you specify locate mode, RMS-11 uses it only when such usage does not compromise data integrity. Otherwise, RMS-11 uses move mode. Therefore, your program must still contain a user buffer.

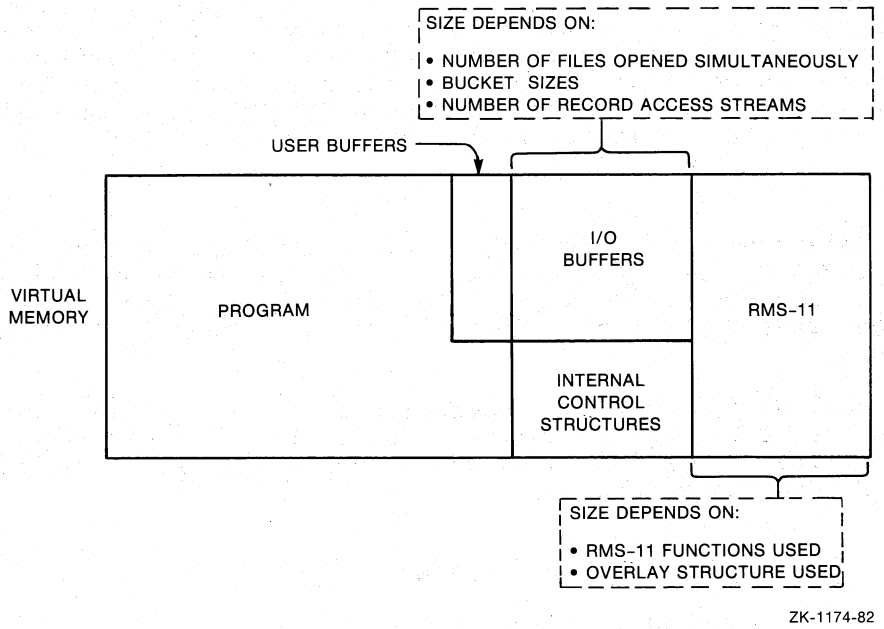


Figure 4-1: RMS-11 Task Structure

Example: RMS-11 uses move mode instead of locate mode when a relative file is shared.

Example: RMS-11 uses move mode instead of locate mode if you opened a file indicating you were going to perform UPDATE operations on it.

RMS-11's use of move mode instead of locate mode is transparent to your program as long as you use RMS-11 facilities to access the record data.

For relative files, your program can only perform GET operations in locate mode. See your high-level language documentation to determine whether the language supports locate mode and, if it does, what the exact programming techniques are.

4.5.3 I/O Techniques

You can use the following techniques to improve the performance of record operations.

4.5.3.1 Asynchronous Record Operations - Within each access stream, your program can perform any record operation either synchronously or asynchronously. In synchronous operations, RMS-11 returns control to your program after the operation ends, either successfully or with an error.

When you execute an asynchronous operation, RMS-11 may return control to your program before the operation is complete. The program continues processing while the physical transfer of data between disk and memory is carried out. However, you must not initiate another record operation on that stream until the first operation ends; otherwise, RMS-11 returns the error code ER$ACT. See your high-level language documentation for asynchronous techniques.

4.5.3.2 Deferred Write - Normally, each write-type record operation (DELETE, UPDATE, and PUT) results in a bucket being written to disk. This convention emphasizes data integrity: you know that when a write-type operation has ended successfully, the file reflects that operation.

However, you can improve the performance of sequential write-type operations by using deferred write. Basically, deferred write directs RMS-11 to write a bucket to disk only when RMS-11 must use the I/O buffer for some other purpose.

NOTE

Deferred write, although not illegal, is essentially invalidated while a relative file is being shared by multiple tasks or streams. In that environment, every write-type operation results in an I/O operation so that:

- The bucket locked by the previous GET or FIND (for UPDATE and DELETE operations) or by the PUT operation can be released.
- The new data is available to the other tasks or streams.

Therefore, if you perform sequential write-type operations on a nonshared relative file, deferred write improves performance. RMS-11 writes out the buffer only when it must read another bucket to complete an operation.

Example: Your records are 304 bytes long and the bucket size is 3 blocks. During sequential write-type operations, deferred write causes I/O operations per bucket to drop from 5 to 1.

Deferred write offers little or no benefit to random write-type operations or read-type operations in any access mode.

4.5.3.3 Multiple Buffers - When you open a relative file, RMS-11 normally allocates 1 bucket-sized I/O buffer in your task's address space. RMS-11 uses this buffer during record operations. However, you can direct RMS-11 to allocate more than the one buffer.

RMS-11 uses any extra buffers to keep, or cache, buckets in memory. When a record operation requires that a bucket be read from disk, RMS-11 checks its cache first. RMS-11 does not perform an I/O operation if both of the following are true:

- The requested bucket is already in memory.
- That bucket is still valid, that is, the file is not shared and/or the bucket has been kept locked.

You do not benefit from multiple buffers during sequential operations. You can improve performance with multiple buffers during random operations only if your program accesses the same buckets often.

4.5.3.4 Multiple Access Streams - RMS-11 allows each program to use multiple streams on a relative file.

4.5.4 File and Directory Operations

The following file and directory operations can be performed on relative files:

CLOSE
CREATE
DISPLAY
ENTER
ERASE
EXTEND
OPEN
PARSE
REMOVE
RENAME
SEARCH

See your high-level language documentation for a description of the support provided.

CHAPTER 5

INDEXED FILE STRUCTURE AND ACCESS

DIGITAL designed the RMS-11 indexed file organization to achieve the following goals:

- Content-addressable record access -- Each record in the file can be located on the basis of the values in designated portions of the data, called key fields.
- Uniform random access time -- Each record in the file can be located with approximately the same number of I/O operations, regardless of when it was added to the file.
- Alternate key capabilities (comply with ANSI COBOL Level 2) -- Each record in the file can be located via more than one key field.
- Very good performance on sequential access by primary key -- A program can sequentially read a reasonably designed indexed file by primary key almost as fast as it can sequentially read a sequential file.
- Good performance on sequential access by alternate keys -- Each record in the series can be accessed with (typically) one to three I/O operations.
- Unique record address for the life of the file (data base key concept) -- A record in a file can be located via a unique identifier (record file address) established by the PUT operation. The record may be deleted, but its unique identifier is never reused.
- Preserve the state of processing despite a system failure -- Normally, each logical write operation results in a physical transfer of data from memory to disk. Therefore, the file reflects each record inserted. However, you can override this mode with deferred write in some cases.

More importantly, RMS-11 performs record operations so that both of the following are true:

- File corruption is avoided or minimized even if a system failure occurs during a write-type record operation.
- Even if some corruption exists, user data can still be accessed.

NOTE

You should still reorganize your file if the system fails during write-type processing on an RMS-11 indexed file.

5.1 PHYSICAL FILE STRUCTURE

On disk, an indexed file consists of three kinds of blocks:

- Prologue -- RMS-11 information about the file, including attributes and key and area descriptions
- Index -- Index records for primary and alternate keys pointing the way to a data record
- Data -- Your data records and index data records

The prologue contains information about the keys and areas of the file. RMS-11 allocates at least one block for the key descriptors and at least one block for the area descriptors. RMS-11 uses more blocks as needed. Size calculations are discussed in Section 6.6.1.

Areas are portions of an indexed file that are treated independently for initial allocation, extensions, placement, and bucket sizes. Like subfiles, but invisible to the operating system, areas allow you to divide indexed files logically into separate units for each index and for the data records to improve performance; see Section 6.3 for more information on areas.

In addition, RMS-11 extends the prologue to an integral multiple of the area 0 bucket size, if the area 0 bucket size is 2, 4, or 8 blocks. See Section 6.5 for more information on bucket sizes.

The location of the index and data blocks is up to you:

- If the file is a single area, RMS-11 allocates data and index blocks in buckets as it needs them; they are therefore interspersed throughout the file.
- If the index and data are set up in separate areas, RMS-11 allocates each type of bucket from the appropriate area; the index is therefore set apart physically from the data portion of the file.

Figure 5-1 illustrates an indexed file both with and without areas.

RMS-11 formats buckets in an indexed file as it requires them for record storage. The RMS-11 control bytes are set to their initial values:

- 14 bytes, beginning with byte 0 of the bucket contain bucket control information.
- The last byte of the last block duplicates the first byte of the bucket for checking I/O completion.

RMS-11 packs index or data records, including record format overhead, into each bucket, beginning with byte 14, end-to-end and byte-aligned.

Figure 5-2 shows the RMS-11 bucket format.

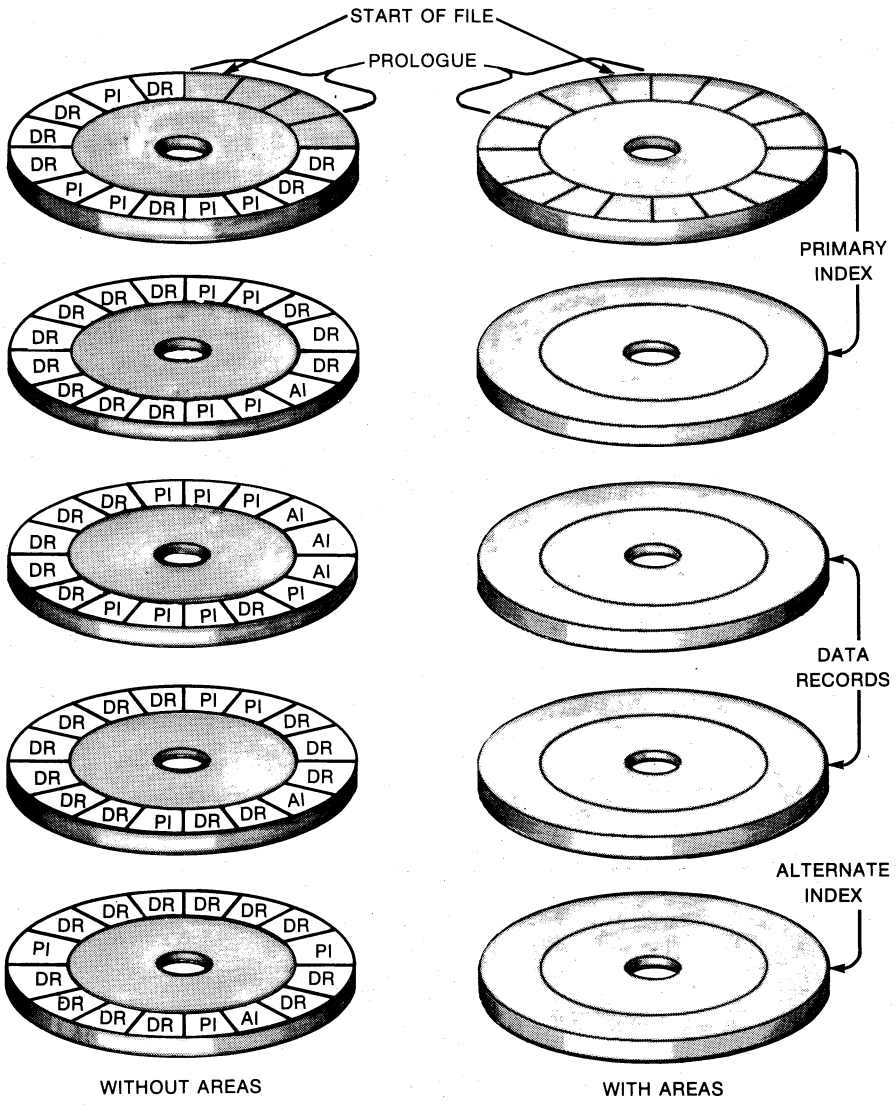


Figure 5-1: Indexed File with and without Areas

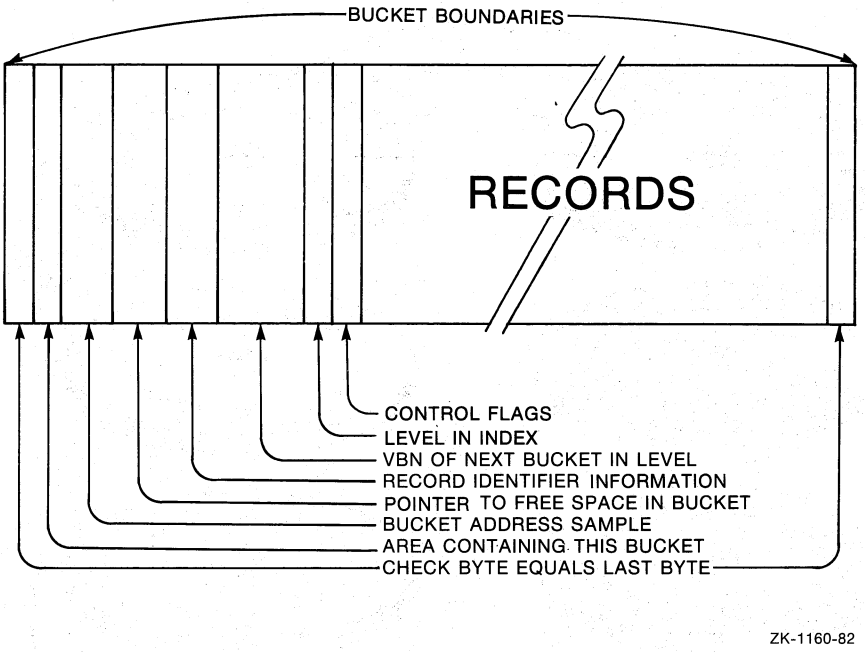


Figure 5-2: Formatted Bucket

5.2 CONCEPTUAL FILE STRUCTURE

No matter how it is laid out physically, the indexed file is conceptually a prologue plus a group of indexes, one per key. Each index consists of horizontal chains of buckets called levels. Figure 5-3 illustrates this structure as a pyramid.

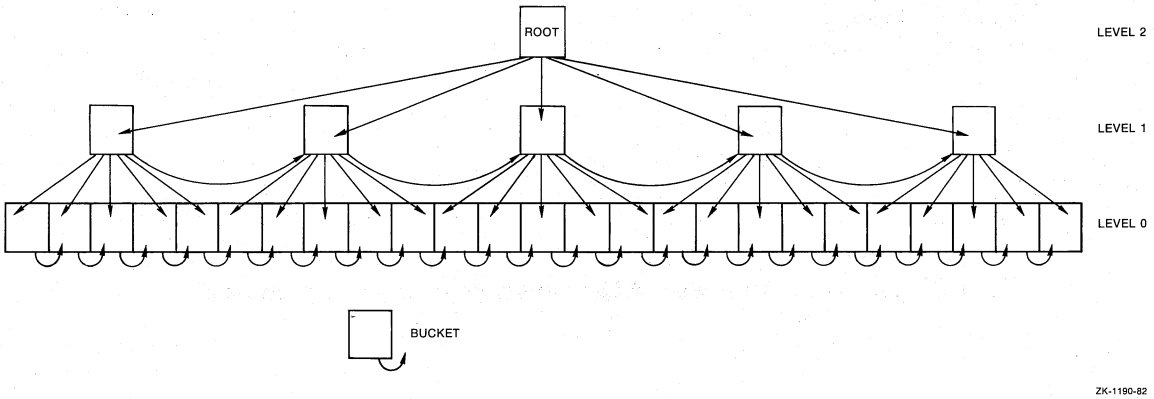


Figure 5-3: Index as a Pyramid

The lowest level of an index is level 0. The level number is incremented for each successive (and smaller) level, that is, level 1, level 2, and so on. The highest level in an index is a single bucket called the root; this bucket is the entry point to the index for random accesses using this key. Each index has at least two levels (0 and 1).

The depth of an index is equal to the level number of the root. An index depth relates to the time needed to randomly access any record in the file via that index.

5.2.1 Data

Level 0 of each index is called the data level; it consists of data buckets. In the primary index, level 0 contains buckets of your data records. In the alternate indexes, level 0 buckets contain pointers to your data records.

5.2.1.1 Level 0 of the Primary Index - RMS-11 physically orders data records by ascending primary key value along the bucket chain. The records having the lowest primary key value reside in the first bucket of the level and the records with the highest primary key values reside in the last bucket. RMS-11 preserves this order regardless of the insertion sequence of the records.

Each bucket in level 0 shares the following properties:

- The last data record in a bucket has an equal or higher key value than any other record in the bucket.
- The last data record in a bucket has a lower key value than the first record in the next bucket in the chain.

Each bucket thus has a high-key value, located in the last record of the bucket. This concept is the core of RMS-11 index file structure.

NOTE

RMS-11 places records with duplicate key values next to each other on a first-in, first-out (FIFO) basis. If these duplicate records cannot fit in the same bucket, RMS-11 stores the overflow in a continuation bucket. Continuation buckets are extensions of level 0 buckets and, as such, are not indexed. This extension storage preserves the high-key concept.

5.2.1.2 Level 0 of an Alternate Index - Level 0, the data level, of an alternate index contains secondary index data records (SIDRs). A SIDR consists of two elements:

1. An alternate key value from a data record stored in the primary data level. The SIDRs in the data level of each alternate index are stored in ascending order by this key value.

2. One or more pointers to data records in the primary data level. Multiple pointers occur when you allow duplicates for the alternate key and records with duplicate values for the key actually exist in the file.

Figure 5-4 illustrates the SIDR format.

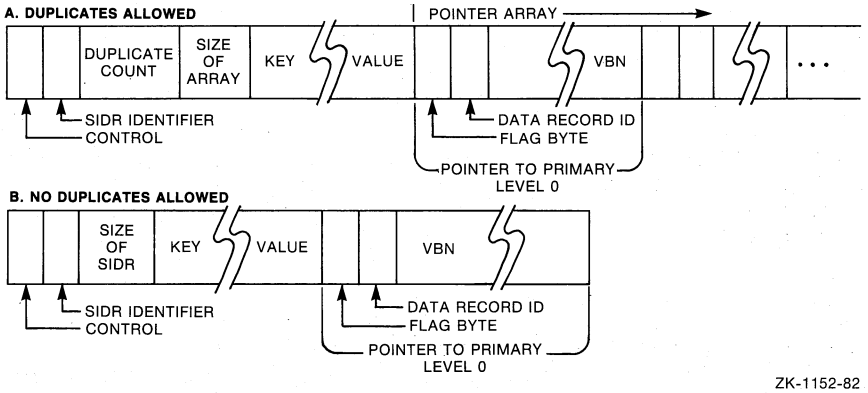


Figure 5-4: Format for Secondary Index Data Record

5.2.2 Indexes

Levels 1 and above in an index are called the index levels; they consist of index buckets. Index buckets contain index records that guide RMS-11 through the levels to the data records in primary level 0. An index record consists of two elements:

1. The high-key value of a bucket in the next lower level in the index. Because RMS-11 arranges these values in ascending sequence, there is a high-key value for index buckets also. However, the last high-key value in the last index bucket of a level is set to the highest possible key value, rather than the highest key value in the file. The associated pointer references the last bucket in the next lower level.

2. A pointer to the bucket associated with the high-key value.

Example: The buckets in level 1 of the primary index contain the high-key values of the data buckets in level 0. Then, level 2 contains the high-key values from level 1 and so on. Figure 5-5 shows an example of a primary index.

In other words, each bucket on a given level is represented by an index record in the next higher level. Thus, the number of buckets required on each successive level decreases exponentially until the root bucket is reached.

Example: If an index bucket can hold 10 index records, then:

- If level 0 contains 2000 data buckets:

  Level 1 contains 200 index buckets
  Level 2 contains 20 index buckets

Level 3 contains 2 index buckets
Level 4 contains 1 index bucket

- If level 0 contains 10,000 data buckets:

  Level 1 contains 1000 index buckets
  Level 2 contains 100 index buckets
  Level 3 contains 10 index buckets
  Level 4 contains 1 index bucket

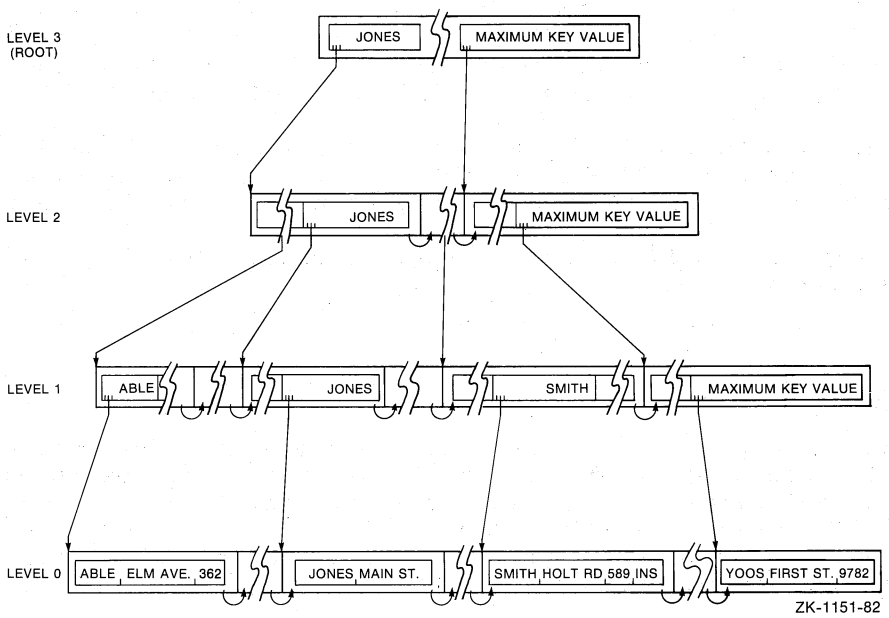


Figure 5-5: Example of a Primary Index

5.2.3 Random Access Using the RMS-11 Indexed File Structure

The following steps show how RMS-11 uses the indexed file structure to execute a random access operation. These steps constitute a process called "follow the index."

1. RMS-11 examines memory-resident index descriptors to find the location of the root for the specified index. Note that the root can be cached (see Section 7.4.3), eliminating the I/O operation to read the root in the next step.

2. RMS-11 reads the root and scans for the first value greater than or equal to the key value specified when the operation was initiated. If all else fails, the search will find the highest possible key value in the last index record.

3. RMS-11 reads the bucket indicated by the pointer associated with the selected key value and scans for the first key value greater than or equal to the value specified. RMS-11 repeats this step through the levels until level 0 is reached.

Example: Refer to Figure 5-5 for this example.

The specified primary key value is "YOOS."

RMS-11 determines the VBNs of the root bucket from the memory-resident index descriptors and requests the file control processor to read those blocks into an I/O buffer. RMS-11 scans the index records in the root. The first key value equal to or greater than "YOOS" is the maximum key value in the last record.

RMS-11 uses the bucket pointer in this index record to request another I/O operation. The file control processor reads the specified blocks into the I/O buffer, and RMS-11 scans them looking for a key value equal to or greater than "YOOS." Again, it finds no qualifying key value until the last record in the bucket, which contains the maximum key value. This index record points to a level 1 bucket.

Upon RMS-11's request, the file control processor brings the indicated bucket into memory. RMS-11 searches the bucket, terminating with the last record in the bucket, which contains the maximum key value.

The file control processor reads the indicated level 0 bucket at RMS-11's request.

5.2.4 Why this Structure?

Mechanical data storage devices make I/O operations the slowest part of file processing. Ideally, a file is read into memory when it is opened and maintained there, without additional I/O operations, until the file is closed. Some very small files allow this approach and are handled most efficiently by your own search techniques rather than by RMS-11's indexing facilities.

However, most indexed files are very much larger than the memory available for data buffering. Such files are therefore partitioned into pieces that can be read to memory. RMS-11 calls these pieces buckets. By definition, one I/O operation is required to access one bucket.

If no index to the data exists, a task must scan sequentially through the buckets of a file to find a specific record. Such a search, on the average, accesses half the buckets in the file. Figure 5-6 plots the time curves for various file searching techniques.

You can optimize nonindexed access by:

- Ordering the records by a key value
- Using a binary search technique

Then, the number of accesses required to find a record approaches log2 of the total number of buckets (see Figure 5-6). This better, but still mediocre, speed is realized on one of perhaps many keys.

The RMS-11 indexed structure uses buckets so that your programs can handle files more efficiently. In most cases, RMS-11 uses n+1 I/O operations to locate a record by primary key, where **n** is the depth of the file's primary index.

In a small file, this technique is not appreciably faster than a sequential scan. However, given typical key sizes, a primary index of depth 3 can represent from 1,000 to 125,000 buckets of data records, using only single-block buckets. Normally, four disk accesses are needed to get any record by primary key value.

Example: You want to search 50 buckets of data for records with specific primary key values. The average number of buckets you read during each search depends on your search technique (see Figure 5-6):

25.5	buckets for a nonindexed search of unsorted records
5+	buckets for a binary search of records sorted by primary key
2	buckets for an RMS-11 indexed search

5.3 PROCEDURES FOR PERFORMING RANDOM RECORD OPERATIONS

The procedures for performing random record operations on indexed files depend on the circumstances for the individual operation, the file's design, and whether alternate indexes must be updated.

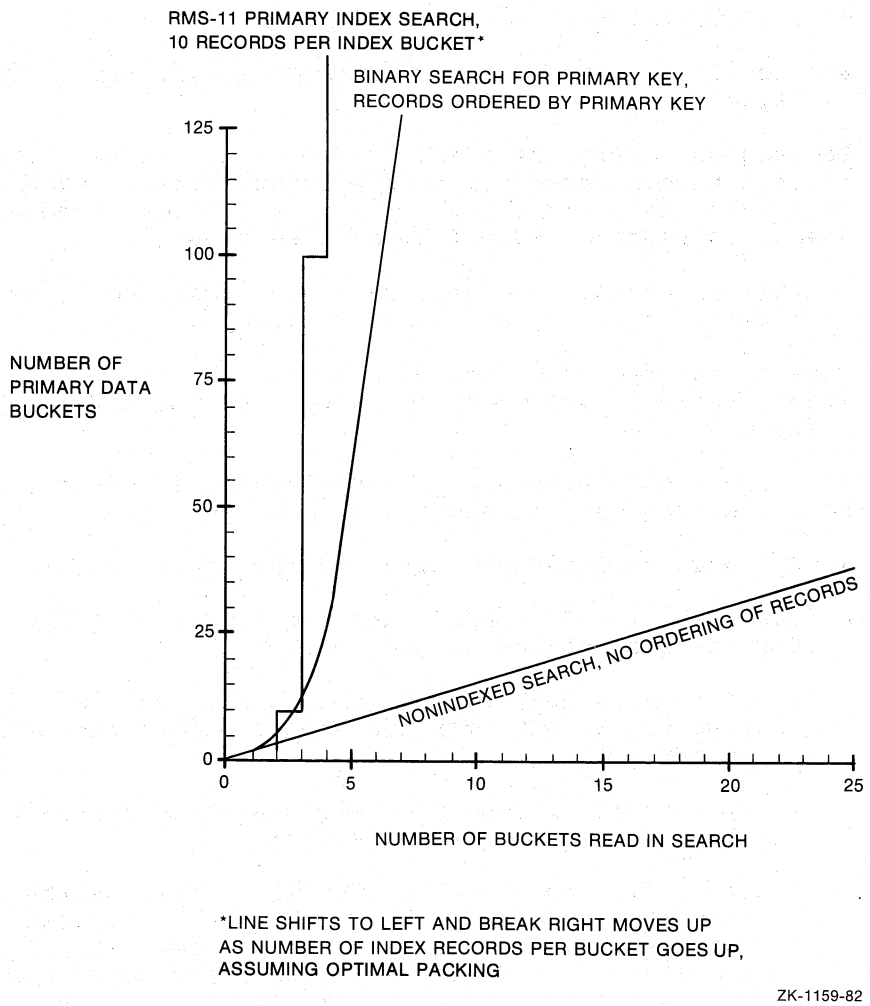


Figure 5-6: Search Time Curves

5.3.1 Writing a Record

When your program initiates a PUT operation, RMS-11 moves the data from the task to the proper bucket in level 0 of the primary index and updates all indexes involved with the record. This process can be simple, requiring minimal I/O operations. It can also be complex, requiring more procedures and data transfers. The complexity depends on whether there is enough room for the new record in its data bucket.

5.3.1.1 Simplest Case - In the simplest PUT operation, RMS-11 finds room in the target data bucket to insert the record. To execute the operation, RMS-11 performs the following steps:

1. Determines the value of the primary key field from the record.
2. Follows the primary index to the proper level 0 bucket.
3. Reads the level 0 bucket and sequentially scans for the first record with a primary key value greater than the specified value. RMS-11 then establishes a position before that record, or after the last existing record in the bucket if:
   - The key values are equal.
   - The first record in the next data bucket has a higher key value.
4. Compresses deleted records. RMS-11 can reuse bytes in a deleted record depending on the record format and whether you allow duplicates in the primary key field. Section 6.2.5 discusses reusing space from deleted records.
5. Determines whether the record to be inserted fits in the bucket (in this simplest case, it does).
6. Inserts the record at the established position. No primary index buckets are updated since no high-key value has changed.
7. If there are alternate keys, updates those indexes, using the following sequence of steps for each one:
   - Follows the alternate index to the proper level 0 bucket.
   - Reads the level 0 bucket and sequentially scans for the key value specified in the record:
     - If a value higher than the one specified is found, inserts a SIDR for the record before the SIDR for the higher value.
     - If a match is found, determines whether duplicates are allowed for the alternate key:

       If duplicates are allowed, RMS-11 follows the duplicate pointer array in the SIDR to the end, then inserts a pointer to the newly inserted record. This procedure preserves the first-in, first-out convention. After the last alternate key, RMS-11 returns a successful completion code to the program.

If duplicates are not allowed, RMS-11 returns to level 0 of the primary index, flags the newly inserted record as deleted, logically removing it, and returns an error code to the program.

Example: Refer to Figure 5-5 for this example.

RMS-11 examines the record in the user buffer of the record access stream initiating the PUT operation. The value in the primary key field is "JACKSON." RMS-11 locates the primary root and requests the file control processor to read the bucket into the I/O buffer associated with the stream. When that I/O operation completes, RMS-11 scans the bucket, looking for a key value equal to or greater than "JACKSON." It finds "JONES."

RMS-11 requests the bucket indicated by the pointer in the "JONES" index record. When RMS-11 scans this level 2 bucket, it finds that "JONES" again ends the search. Following the pointer in this index record, RMS-11 requests another bucket. Its search of the level 1 bucket ends in another "JONES" index record.

RMS-11 requests the level 0 bucket indicated by this last index record. It finds that a data record with a primary key value of "JONES" is the only occupant of the bucket. There are no deleted records to compress, so RMS-11 writes the "JACKSON" record before the "JONES" data record, moving the "JACKSON" record down in the bucket.

There are no alternate keys. RMS-11 returns a successful completion code to the program.

5.3.1.2 Bucket Splitting - If there is not enough room in the target data bucket for the record, RMS-11 allocates a new bucket and reorganizes the records in the old one between the two buckets. This procedure is called bucket splitting.

Bucket splitting is identical with the simplest case (Section 5.3.1.1) to step 5 where RMS-11 determines whether the new record fits in the bucket. When there is not enough room, RMS-11 does the following:

1. Reads the appropriate area descriptor from the file prologue. If enough blocks for a bucket are allocated for the area, RMS-11 formats the blocks into a bucket and updates the area descriptor to reflect the new bucket. Otherwise, RMS-11 requests the operating system to allocate enough blocks, and then formats them into a bucket and updates the area descriptor.

2. Splits the target bucket at the point where the record should be inserted. RMS-11 moves the records in the **high portion** of the bucket into the new bucket; these records have primary key values higher than those of the new record.

NOTE

When RMS-11 moves a record between buckets, it marks the record's original location with a record reference vector (RRV). An RRV is a copy of the record's header (both contain 7 bytes). RRVs preserve alternate key and RFA access, holding the original location of the record and pointing to its current location. Only one RRV is created for a record: if the record moves again, RMS-11 updates the RRV with the record's new location.

Since the original location of a record is filled, either with the record or a pointer to that record, RMS-11 does not have to update alternate indexes every time a record moves. This convention means one extra I/O operation may be needed to find or get a record via an alternate key, but it prevents a complex and costly index update for each bucket split.

3. Inserts the data record in the original target bucket. If the record will not fit, RMS-11 inserts it into the new bucket. If the record will not fit there either, RMS-11 will create another bucket (see step 1) and put the record there.

4. Updates the level 0 bucket chain to include the new bucket(s).

5. Returns to the primary root bucket and follows the index to the level 1 index bucket that points to the data bucket that split.

6. Inserts index record(s) for the new data bucket(s). If the index bucket splits, RMS-11 uses a procedure similar to this to move the index records and update the next higher level of the index. Splitting can occur all the way to the root where a new root is created and the file prologue updated.

7. If there are alternate keys, RMS-11 updates those indexes as described in step 7 of the simplest case (Section 5.3.1.1). Bucket splitting can occur in alternate indexes also.

5.3.1.3 Incremental Reorganization - The process of inserting each data record where it belongs in level 0 and updating the indexes when RMS-11 inserts the record is called incremental reorganization of the file.

Incremental reorganization has the following advantages:

- It eliminates reorganization periods where special software incorporates overflow areas into the main file and that file is not available for processing
- It ensures equal access time to old and new records
- It enables performance on sequential access by primary key to approach the speed of sequential access to a sequential file

This process has its costs: additional I/O operations occur when a bucket splits. But with good file design and file loading, bucket splitting (and the time for each bucket split) is minimal. Chapter 6 discusses these considerations in detail.

5.3.2 Getting and/or Finding a Record

To execute a key-access GET or FIND operation, RMS-11 performs the following steps:

1. Determines from the instruction initiating the operation the following criteria:
   - Key of reference, indicating which index to search and which key field within the data record to examine
   - Value to find
   - Match criterion (equal to, greater than, or both)
   - Number of characters to match
2. Follows the index to the proper level 0 bucket.
3. Reads the level 0 bucket, sequentially scanning for the first record with a value in the specified key field that matches the specified value according to the match criterion. This search can continue into other buckets:
   - If no such record is found, RMS-11 returns an error code.
   - If such a record is found, RMS-11:
     - Determines which index has been read:

       If it is the primary index, RMS-11 goes to the next step.

       If it is an alternate index, the record located is a SIDR. RMS-11 follows the SIDR pointer to the primary level 0 data record.
     - For a GET operation only, moves the record to the user buffer associated with the access stream performing the operation.
     - Sets the current context for the stream performing this operation. The effect of each record operation on context is described in Section 7.2.
     - Returns a successful completion code.

Example: Refer to Figure 5-5 for this example.

RMS-11 determines the key (and index) of reference and the value to find from the instruction initiating the operation. In this case, they are the primary key (key 0) and "ABI."

RMS-11 locates the primary root and requests the file control processor to read the bucket into an I/O buffer. RMS-11 sequentially scans the root for an index record whose key value is equal to or greater than "ABI." It finds "ABRAM."

RMS-11 requests the bucket indicated by the pointer in the "ABRAM" index record. When RMS-11 searches this level 2 bucket, it finds an index record containing the key value "ABNER." Following the pointer in this index record, RMS-11 requests another bucket. The search of the level 1 bucket ends in the key value "ABLE."

RMS-11 requests the level 0 bucket indicated by this last index record. RMS-11 changes its search criteria to that specified in the initiating instruction: it looks for a record where the first 3 bytes of the primary key field equal "ABI." Since the only record in the bucket contains "ABLE" in its primary key field, RMS-11 cannot satisfy the search requirements. It returns a "record not found" error code to the program.

5.3.3 Updating a Record

RMS-11 requires an UPDATE operation to be preceded by a GET or FIND operation, although some high-level languages hide this prerequisite.

To execute an UPDATE operation, RMS-11 performs the following steps:

1. Locates the key fields of the revised record in the user buffer associated with the access stream performing the operation. RMS-11 compares those key values with the values in the current record:
   - If the primary key value changed, RMS-11 returns an error code.
   - If an alternate key value changed, RMS-11 checks whether you allowed changes for that key:
     - If not, RMS-11 returns an error code.
     - If so, RMS-11 continues processing.
2. For each alternate key where the key value changed, RMS-11 performs the following steps to delete the pre-update value from the alternate index:
   - Reads the data bucket containing the current record, if that bucket is not in memory.
   - Saves the pre-update alternate key value from the current record.
   - Follows the index to the level 0 bucket that should contain the SIDR for the pre-update key value.
   - Reads the level 0 bucket and sequentially scans for the pre-update key value:
     - If a value higher than the one specified is found, RMS-11 goes to the next alternate index.

Example: RMS-11 scans a bucket, searching for a pre-update key value of "D." It finds a record with a key value of "E." Since "E" is greater than "D," RMS-11 ends the search and this step in the procedure.

- If a match is found, RMS-11 scans the duplicate pointer array in the SIDR to the entry for the record being updated and flags it as deleted.

NOTE

To allow keys to change, RMS-11 requires that you also allow duplicates. Therefore, if you allow alternate key values to change, there is a duplicate pointer array in the SIDR for each key value. However, you should refer to your high-level language documentation for specific information on your compiler's implementation of this capability.

3. Reads the data bucket containing the current record, if that bucket is not in memory. RMS-11 replaces the current record in the I/O buffer with the updated version in the user buffer.

4. Writes the bucket to the file.

5. For each alternate key where the key value changed, RMS-11 performs the following steps to insert the post-update value in the alternate index:

   - Reads the data bucket containing the current record, if that bucket is not in memory.
   - Follows the index to the level 0 bucket that should contain the post-update key value.
   - Reads the level 0 bucket and sequentially scans for the post-update key value:
     - If a value higher than the one specified is found, RMS-11 inserts a SIDR for the new record before the SIDR for the higher value.
     - If a match is found, RMS-11 follows the duplicate pointer array in the SIDR to the end, then inserts a pointer to the new record.

   After the last alternate key is updated, RMS-11 returns a successful completion code to the program.

5.3.4 Deleting a Record

RMS-11 requires a DELETE operation to be preceded by a GET or FIND operation, although some high-level languages hide this prerequisite.

To execute a DELETE operation, RMS-11 performs the following steps:

1. If there are alternate keys, RMS-11 updates those indexes as follows, using the same sequence of steps for each:

   - Reads the data bucket containing the current record, if that bucket is not in memory.
   - Follows the index to the level 0 bucket that should contain the SIDR for the key value in the deleted record.
   - Reads the level 0 bucket and sequentially scans for the specified key value:

     If a value higher than the one specified is found, RMS-11 goes to the next alternate index, if any.

     If a match is found, RMS-11 determines whether you have allowed duplicates:

     - If so, RMS-11 follows the duplicate pointer array in the SIDR to the entry for the record being deleted and flags it as deleted.
     - If not, RMS-11 deletes the SIDR.

2. Reads the data bucket containing the current record, if that bucket is not in memory.
3. Changes the flag byte in the header of the current record to indicate that the current record is deleted.
4. Writes the bucket to the file.
5. If the record has moved, reads the level 0 bucket containing the RRV. RMS-11 changes the flag byte in the RRV to indicate that the record is deleted.
6. Writes the bucket to the file.

RMS-11 does not compress a deleted record until it needs space to insert another user data record into the bucket (see Section 5.3.1). RMS-11 does not compress deleted RRVs.

NOTE

RMS-11 does not modify or reduce any index structure or allocation during a DELETE operation.

5.4 PROCEDURES FOR PERFORMING SEQUENTIAL RECORD OPERATIONS

Your program can use sequential access mode to perform the following record operations:

FIND
GET
PUT

During sequential-access GET and FIND operations, RMS-11 does not usually read an index to locate the specified record. Instead, RMS-11 uses the record context for the stream performing the operation to identify the proper data bucket.

For FIND operations, RMS-11 uses the next record pointer to identify the target bucket. For GET operations, RMS-11 uses the next record pointer, if the GET operation was not immediately preceded by a successful FIND operation. The current record pointer is used if the GET operation was immediately preceded by a successful FIND operation.

Next, RMS-11 requests the file control processor to move the target bucket into the I/O buffer, if that bucket is not in memory. If it has requested a SIDR bucket, RMS-11 then follows the appropriate pointer to the user data record.

During sequential-access PUT operations, RMS-11 compares the primary key value of the specified record with the primary key value of the last record written:

- If the specified record's primary key value is equal to or greater than the last record's primary key value, RMS-11 performs a key-access PUT operation (described in Section 5.3.1).
- If the specified record's primary key value is less than the last record's primary key value, RMS-11 returns an error code to the program.

5.5 I/O COST OF PERFORMING RECORD OPERATIONS

Table 5-1 provides simple algorithms for predicting the number of I/O operations any RMS-11 record operation requires:

- The value n = index depth of the indicated key; all indexes do not necessarily have the same depth.
- Algorithms do not include I/O operations caused by program or RMS-11 overlays, operating system overhead, or by file extensions (see Chapter 8).

Table 5-1: I/O Cost of Performing Record Operations

Record Operation	Primary Key	Each Alternate Key
Key-access GET or FIND		
Record in original location	n+1	n+2
RRV in original location (record moved)	n+1	n+3
Sequential-access GET or FIND	0-2 [a]	1-4 [b]
PUT		
Simplest case	n+2	n+2
Split in data level	2n+6 [c]	2n+6 [c]
Bucket split up entire index	(n**2+11n+20)/2 [d]	(n**2+11n+20)/2 [d]
UPDATE [e]		
Alternate key value did not change	1	0
Alternate key value changed	1	2(n+2) [f]
DELETE		
Record in original location	1	n+2 [g]
RRV in original location (record moved)	3	n+2 [g]

[a] Breaks down to:

0 or 1	I/O to position to current record
0 or 1	I/O to locate next record

[b] Breaks down to:

0 or 1	I/O to position to SIDR for current record
0 or 1	I/O to locate SIDR for next record
1 or 2	I/Os to retrieve user data record

[c] Breaks down to:

n+2	I/Os to read and write the old bucket
n+1	I/Os to read and write the level 1 index bucket
3	I/Os to write the new bucket and update the area descriptor in the prologue

[d] Breaks down to:

(n+2)+(n+1)+n+(n-1)+...+3+2

I/Os to return to the primary root and read and write updated buckets from level 0 to the root

3(n+1)	I/Os for each bucket split (see footnote c)
5	I/Os to create the new root

[e] Values assume record length does not change and cause bucket splitting.

[f] n+2 if either the old or new key value does not belong in the index; for example, the field contains the null key value defined for the key, or a variable-length record does not contain the whole key field.

[g] Value is different if one of the following is true:

- You specified the "fast delete" option (available in MACRO-11 only) when you initiated the DELETE operation. Then, RMS-11 does not update alternate indexes in which duplicate keys are allowed.
- RMS-11 has to scan a long duplicate array into one or more continuation buckets. Then, one I/O operation is needed for each additional bucket.

CHAPTER 6

INDEXED FILE DESIGN

Indexed file design ranges from the basic elements of your application (record definition and key selection) to the structure of the file and to the methods used to put the data into the file. This range includes:

1. Record size
2. Key selection
3. Areas
4. Placement control
5. Bucket size
6. Allocation
7. Population techniques

6.1 RECORD SIZE

You can use only fixed- and variable-length records in RMS-11 indexed files. RMS-11 calculates length (RL), in bytes, as follows:

RL = 7+RFO+DS

where:

7 is bytes for RMS-11 record header

RFO is bytes for record format overhead: 0 for fixed, 2 for variable

DS is bytes of data

Set your record size to reflect application requirements; do not adjust it to fit bucket size. For instance, if you are using 1-block buckets, you should not, if you can avoid it, set a record length so the records just fit into the buckets:

```
512     bytes in a block
-15     bytes of indexed file overhead per bucket
---
497     bytes left for records
- 7     bytes for the record header
---
490     bytes left for the data and record format overhead
```

This calculation seems ideal at first. However, when the record moves during a bucket split or RMS-11 deletes the record, and some RMS-11 overhead is left in the bucket, a normal data record cannot fit: the bucket is essentially useless, with up to 490 bytes of unused space.

If your application requires 490-byte records, you should use them, keeping the preceding limitation in mind and, perhaps, choosing a different bucket size.

NOTE

Records in an indexed file cannot span buckets and bucket sizes are limited by the operating system to 32 blocks. Therefore, the maximum record size, including overhead, is 16,369 bytes.

6.2 KEY SELECTION

A file's keys can take up significant space in an indexed file and can have a significant effect on the number of I/O operations needed to access the file. During key selection, you should consider the following:

- Number of keys
- Key data type
- Key size
- Position of key in record
- Key characteristics

6.2.1 Number of Keys

You can specify from 1 to 255 keys for an indexed file:

- One primary key that RMS-11 requires for every indexed file
- 254 alternate keys

There are overhead costs in key specification: For each key specified in an indexed file, RMS-11 builds an index. Since RMS-11 requires a primary key, you must accept that key's index overhead, but you should consider the cost before specifying an alternate key for the file:

- RMS-11 updates alternate indexes when your program:
  - Puts a new record into the file
  - Updates a record in the file and the alternate key values change
  - Deletes an existing record

  The time required for this update relates to the number of I/O operations needed to follow each alternate index from the root to level 0, to change or insert the SIDR, and to rewrite the bucket. RMS-11 can require additional time if one or more buckets in the index split.

- An index takes room in the file. You can estimate the disk space for an alternate index (see Section 6.6.1).

Whether the cost of each alternate key is bearable depends on your application. If the primary purpose of the application is to write, update, or delete records, each alternate key will noticeably burden the operations; therefore, the number of alternate keys should be kept to the minimum. Rarely used alternate access paths call for a separate program that sorts by the desired nonkey field and then processes the data.

However, if the primary purpose of your application is to get records from the file, then alternate keys do not burden processing. In fact, alternate keys give flexibility to information retrieval. However, the cost of the extra keys is borne on those few occasions when records are added to the file.

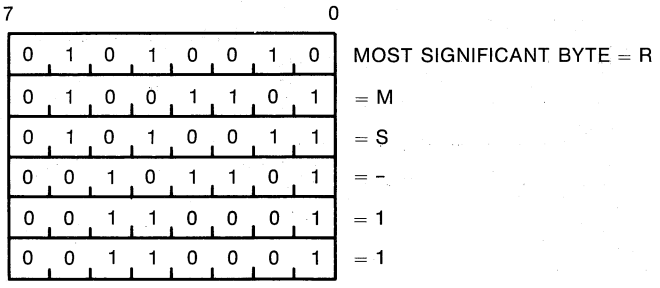
6.2.2 Key Data Types

Each key in an indexed file can be one of the following data types:

String
2-byte signed integer
4-byte signed integer
2-byte unsigned binary
4-byte unsigned binary
Packed decimal

6.2.2.1 String Type - RMS-11 interprets each character of the key in a byte by its binary contents. Permissible values are not limited to valid ASCII codes.

Example: The key value "RMS-11" is represented as follows:

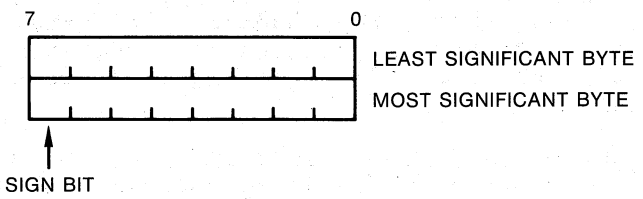


The first (lowest-addressed) byte of the key is the most significant byte of a string key for collating purposes. RMS-11 compares primary keys byte-by-byte, first-to-last, when it determines where the record should be placed in the file.

The maximum key value is all bits in each byte set to 1 (octal 377).

There is a cost in the number of bytes specified as the key length. For example, if you specify a key length of 12, each representation of the key in the data record and in the index takes 12 bytes.

6.2.2.2 Two-Byte Signed Integer Type - Each key requires 2 bytes; RMS-11 interprets the data in the following format:



ZK-1192-82

NOTE

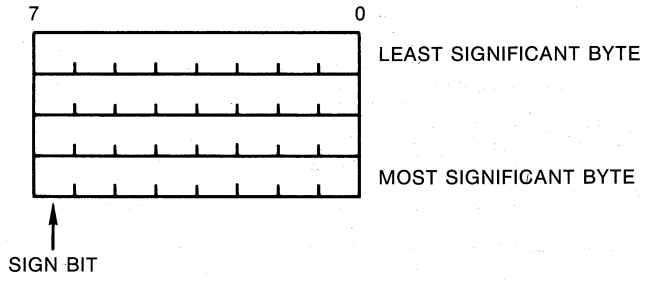
The least significant byte of an integer or binary key is the byte with the lowest address. Significance increases with address. Within a byte, the lowest significant bit is bit 0, and significance increases with position. See your PDP-11 Processor Handbook.

A 2-byte signed integer can represent the decimal values -32,768 through +32,767.

Maximum key value is +32,767.

The cost in key size is 2 bytes per representation.

6.2.2.3 Four-Byte Signed Integer Type - Each key requires 4 bytes; RMS-11 interprets the data in the following format:



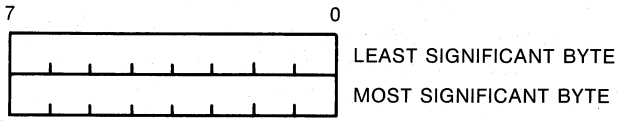
ZK-1193-82

A 4-byte signed integer can represent the decimal values -2,147,483,648 through +2,147,483,647.

Maximum key value is +2,147,483,647.

The cost in key size is 4 bytes per representation.

6.2.2.4 Two-Byte Unsigned Binary Type - Each key requires 2 bytes; RMS-11 interprets the data in the following format:



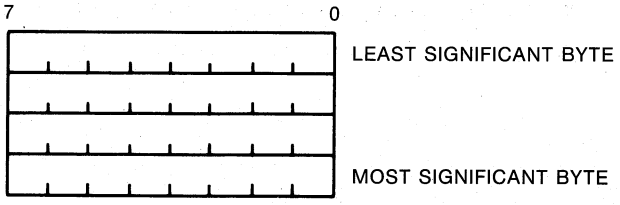
ZK-1194-82

A 2-byte unsigned binary value can represent the decimal values 0 through +65,535.

Maximum key value is 65,535.

The cost in key size is 2 bytes per representation.

6.2.2.5 Four-Byte Unsigned Binary Type - Each key requires 4 bytes; RMS-11 interprets the data in the following format:



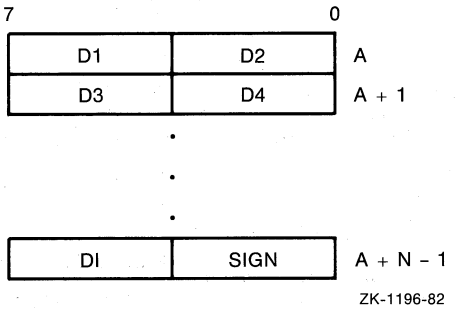
ZK-1195-82

A 4-byte unsigned binary value can represent the decimal values 0 through +4,294,967,295.

Maximum key value is 4,294,967,295.

The cost in key size is 4 bytes per representation.

6.2.2.6 Packed Decimal Type - RMS-11 recognizes 2 decimal digits of the key in each byte except the last. The key format takes the following form:



where:

A is an address: A, A+1,... are increasing (byte) addresses.

D1-DI are decimal digits: D1 is the most significant digit and DI is the least significant digit.

SIGN has a value of 10 through 15: + is represented by a 10, 12, 14, or 15; and - is represented by an 11 or 13.

N is the length of the key in bytes (maximum of 16)

I is the length of the digit string, an odd number in the range of 1 through 31, where I = 2N - 1

Maximum key value is 99 in each byte with the sign positive.

6.2.3 Key Size

Keys for indexed files have length restrictions according to their data types. Table 6-1 lists these restrictions.

Table 6-1: Key Data Types

Data Type	Length (bytes)
String	1-255
15-bit signed integer	2
31-bit signed integer	4
16-bit unsigned binary	2
32-bit unsigned binary	4
packed decimal	1-16

The cost of each key's size is borne in the data record and in the index: RMS-11 stores an entire key value in each index record.

6.2.4 Position of Key in Record

You can locate any key anywhere in the record:

- Alternate keys can precede the primary key.
- Keys can overlap each other. Note, however, that COBOL-81, in keeping with the ANSI standard, does not permit more than one key to start at the same position. The standard calls this leftmost correspondence.

You benefit from careful placement of keys within the record:

- Deleting a record -- When you allow duplicates in the primary key of variable-length records, RMS-11 compresses a deleted record by removing all data except:

  - The record header
  - Enough of the record to contain the primary key

  Therefore, you can optimize DELETE operations if you place the primary key at the beginning of the record. The closer the key is to the beginning of the record and the shorter the key, the fewer overhead bytes remain in the file.

  However, if you have fixed-length records or do not allow primary key duplicates, the position of that key in the record is not significant. See Section 6.2.5.1 for more information on duplicates.

- Writing a record -- You can optimize PUT operations for variable-length records, by placing alternate keys at the end of the record. Then, if no valid data is present in an alternate key field, you can shorten the record to exclude that field, thus reducing the record space in the data level as well as eliminating a reference to that record in the alternate index.

You can segment string keys; all other key data types must be contiguous bytes. You can specify up to eight segments in one string key, each segment with its own length; the total of the lengths cannot exceed 255 bytes. Note that some high-level languages do not make this capability available; see your high-level language documentation.

RMS-11 concatenates the segments you specify before performing any operations requiring a value for the key. RMS-11 defines a segment by byte position within the record and length in bytes. Therefore, the key segments you define with either a MACRO-11 program or RMSDES do not have to align with the data fields you define within the records: RMS-11 has no knowledge of the form of such files.

Example: You have an inventory application with a master product file. Within the product records, you have fields for vendor number, vendor's part code, and your part number, among others. You can define the following keys for the file regardless of the placement of the fields.

```
Primary key = vendor number + vendor part code
Alternate key 1 = vendor number + your part number
Alternate key 2 = vendor number
Alternate key 3 = your part number
```

For cost, see the preceding considerations about the placement of keys within a record, knowing that a key consists of all segments.

6.2.5 Key Characteristics

Key characteristics include:

- Duplicates
- Changes
- Null key

Characteristics are restricted according to key number:

Characteristics	Primary Key (0)	Alternate Keys (1+)
Duplicates	Allowed	Allowed
Changes	Not allowed	Allowed
Null key	Not allowed	Allowed

The combination of changes and duplicates is also restricted by key number:

Combination	Primary Key (0)	Alternate Keys (1+)
CHG+DUP	Error	Allowed
CHG+NODUP	Error	Error
NOCHG+DUP	Allowed	Allowed
NOCHG+NODUP	Allowed	Allowed

NOTE

COBOL-81 allows the CHG+NODUP combination for alternate keys. To enable this option, the COBOL-81 OTS uses a hidden FIND operation to check on duplicates each time an alternate key value changes on an UPDATE operation (REWRITE in COBOL-81).

6.2.5.1 Duplicates - If duplicates are allowed for a key, more than one record can have the same value in that key field. The overhead costs are:

- File space -- Duplicates have little effect on space usage as long as records are not frequently updated with changing key values or deleted. If anything, records with duplicate key values are stored more efficiently than records with nonduplicate values: fewer index records are required to cover data records with duplicate primary keys.

  In alternate indexes, one SIDR with one representation of the key value is needed to cover multiple data records with the same value in the key field.

- Writing a record -- RMS-11 stores records with duplicate key values for first-in, first-out access. Writing (and updating) records containing duplicate key values takes more time as the number of duplicates builds up.

A PUT operation can fail because duplicates are not allowed for one of the keys. If this is the primary key, RMS-11 has wasted little time since it has performed only the I/O operations to find the previous record with that value in the key field.

However, if you allowed no duplicates in one of the alternate keys, RMS-11:

1. Updates the primary index, including the data level.

2. Updates the preceding alternate indexes.

3. Discovers that it cannot insert the record because a record already exists with that key value.

4. Reverses the actions it has taken, removing all updates from the indexes it has already rewritten. Entries made in SIDR duplicate arrays are flagged as deleted and not compressed out of existence. However, RMS-11 cannot reverse bucket splits.

5. Returns an error code.

- Deleting a record -- If you do not allow duplicate values for the primary key, RMS-11 compresses a deleted record to a 2-byte indicator when it performs a DELETE operation. However, if you allow duplicate values for the primary key, RMS-11 keeps enough of the record to contain the entire primary key:

  1. If the format is fixed, the entire record remains in the file.

  2. If the format is variable, enough of a record remains in place to hold the entire primary key.

- If you do not allow duplicate values for an alternate key, RMS-11 removes the SIDR when it deletes the data record. However, if duplicates are allowed, the pointer remains in the SIDR array with the delete flag set.

- Updating a record -- If you allow duplicate values for the primary key, the length of a variable record cannot be changed during an UPDATE operation. In addition, updating records containing duplicate key values takes more time as the number of duplicates builds up. Finally, the SIDR pointers for deleted records are flagged as deleted, but not removed from the duplicate array.

- Summary -- Duplicates are not costly for write-type operations unless there are too many of them. Pick a key field that minimizes duplicates.

  Example: Fields where there are only two choices for entries, such as sex, are not good candidates for key fields.

6.2.5.2 Changes - The value of a primary key field cannot change during an UPDATE operation; however, you can allow the value in any alternate key field to change if you are willing to allow duplicate values in that key.

During any UPDATE operation, RMS-11 checks the characteristics of all keys and compares the new key values (in the record about to be rewritten) with the old values: if you do not allow changes in a key field, but changes have been made, RMS-11 immediately terminates the UPDATE operation with an error code.

Cost: If an alternate key value changes during an UPDATE operation, RMS-11 must trace the old SIDR and delete it, then insert the new one, starting with the root of the index for both processes. If the data does not change, however, RMS-11 does not update the alternate index.

6.2.5.3 Null Key - You can specify the null key characteristic for any alternate key. If RMS-11 finds that an alternate key field is filled with the null key value specified for that key, it does not insert an entry into the index for the record being written.

Zero is the null key value for the numeric key data types (integers, binaries, and packed decimal). The null key character for string keys can be any octal value (000 through 377) including an ASCII character: if all bytes in the key field contain this value, the key is considered null.

Cost: The use of a null key value can reduce the disk space that an alternate index occupies, but it also precludes accessing those records not entered in the index via that alternate key.

6.3 AREAS

You should divide an indexed file into areas. An area is a portion of the file that RMS-11 treats as an entity for:

- Initial allocation
- Extensions
- Bucket size
- Placement on disk

Areas allow you to gather logical elements of the file into groups of continuous ranges of VBNs. These VBNs can be mapped onto a contiguous set of logical blocks on disk. This tight sequence of VBNs is lost when RMS-11 extends an area.

NOTE

Unless you completely allocate each area when you create the indexed file, the division of the file into areas may not improve performance.

Areas can be set up for:

- Primary index level 0 (the data records)
- Primary index level 1 (the lowest index level)
- Primary index levels 2 and greater (the rest of the index)

- Alternate index level 0 (SIDRs)
- Alternate index level 1 (the lowest index level)
- Alternate index levels 2 and greater (the rest of the index)

Dividing a file into areas primarily saves I/O time. As explained in Section 5.1, in a single-area file, RMS-11 intersperses index and data buckets: index buckets are scattered among the data buckets. During each random record access, RMS-11 consults the appropriate index descriptor in memory and then directs (through the operating system) the disk head to read the root and levels 2 and greater, level 1, then the appropriate level 0 bucket. These buckets can be anywhere in the file, and the disk head can travel large distances several times to complete one access operation. Figure 6-1 shows an indexed file with one area. Figure 6-2 shows an example of a single-area indexed file.

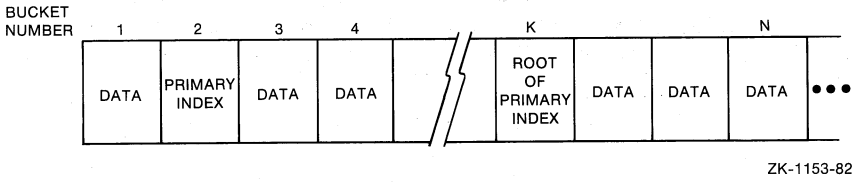


Figure 6-1: Single-Area Indexed File

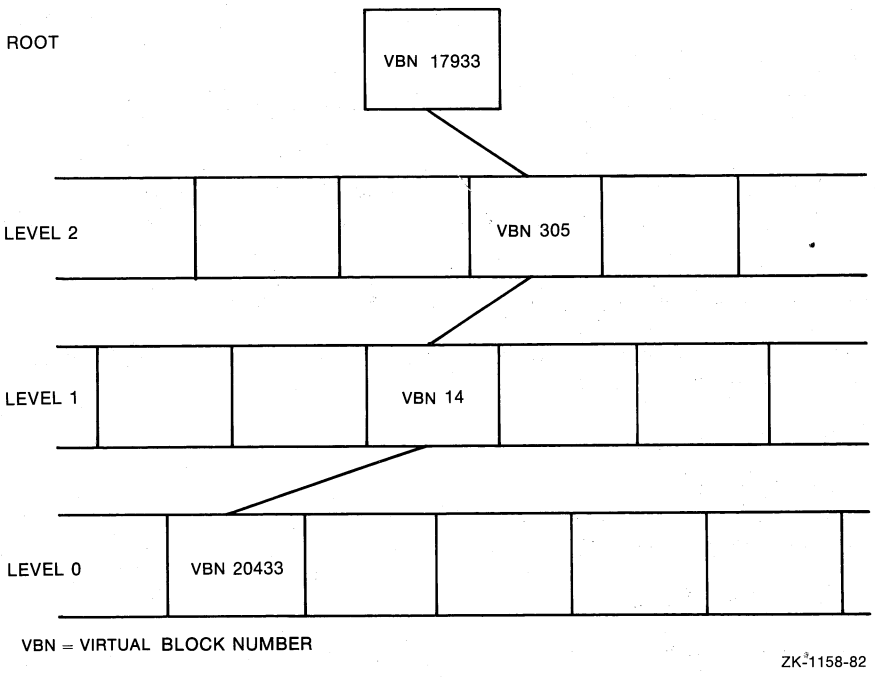


Figure 6-2: Example of Single-Area Indexed File

To randomly access a specific record in the file illustrated in Figure 6-2, RMS-11 makes the following I/O requests:

1. Read VBN 17933
2. Read VBN 305
3. Read VBN 14
4. Read VBN 20433

You can now realize how much the device has to move its read head to service one random access operation.

A multiarea file, on the other hand, can have all index buckets allocated contiguously (if enough blocks were initially allocated): all index information is available in one physical part of the disk. RMS-11 can then traverse an index with little or no head movement until it reads the indicated data bucket. In addition, a sequential read of the file moves the head mechanism smoothly through the physically contiguous area assigned to the primary index level 0. Figure 6-3 shows an indexed file with two areas.

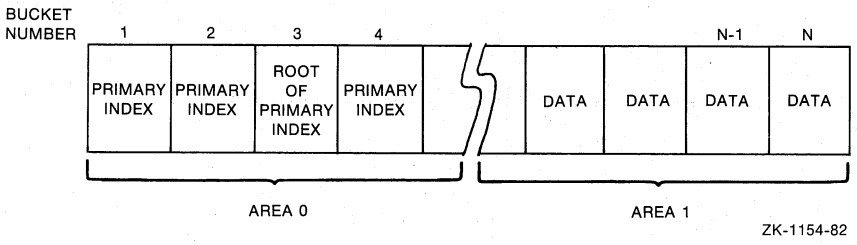


Figure 6-3: Two-Area Indexed File

To refine your file even more, place the lowest level of each index (level 1) in an area separate from the rest of the index (levels 2 and greater).

Figure 6-4 shows an example of a multiarea indexed file.

To randomly access a specific record in the file illustrated in Figure 6-4, RMS-11 makes the following I/O requests:

1. Read VBN 418
2. Read VBN 423
3. Read VBN 1537
4. Read VBN 14703

You can now realize how much the proper use of areas reduces disk head movement during a random access operation.

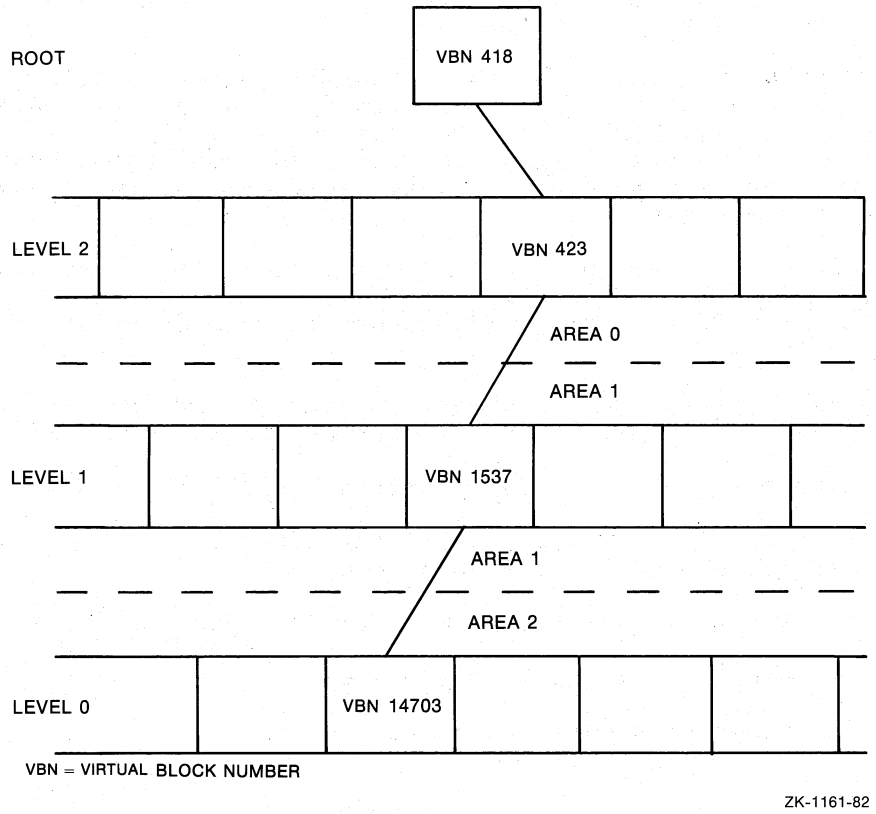


Figure 6-4: Example of Multi-Area Indexed File

When you specify and preallocate multiple areas, RMS-11 arranges them in order in the file: area 0 (including the file prologue) in the first virtual blocks of the file, then area 1, and so on. If you specify contiguity for the entire file, this control over the distribution of structural elements of the file is propagated from the virtual block sequence to the logical block sequence on the disk.

Contiguity is very important to performance. For more information on contiguity, Section 8.3.

6.4 PLACEMENT CONTROL

Placement control enables you to specify the location on a disk for a file or the areas of a file. You use placement control for the following reasons:

- To start a file or area at the first block of a track or cylinder so that the file or area can reside in one or more contiguous tracks or cylinders. This effort minimizes head movement during file access.

- To place the files used by a single application together on a disk. This effort reduces I/O time by minimizing head movement among the files.

  Example: You want to run a general ledger application that uses several files (an accounts file, a transaction file, and so on). The application consists of several tasks. So, you start with an initialized disk and copy the tasks onto it. Then, you create (and populate) your data files, placing them near the tasks.

  This effort reduces the distance the disk head moves to service I/O operations required by an RMS-11 program: disk-resident overlays (discussed in Chapter 8) and data file accesses.

  Note, however, you gain more improvement if you eliminate head contention by placing the individual files on separate disks.

You calculate track and cylinder starting block numbers as follows:

1. Read the documentation that came with your disk drive. Find and write down the following numbers:

   - Number of surfaces on a volume (or pack or disk)
   - Number of tracks on a surface
   - Number of sectors in a track

   NOTE

   On most DIGITAL disk drives, a sector equates to a logical block. For example, the following decimal numbers apply to an RP06 only:

   ```
   Number of cylinders per disk = 815
   Number of tracks per cylinder = 19
   Number of sectors per track = 22
   ```

2. Establish the starting logical block number (LBN) for each track on the disk by writing down the multiples of sectors-per-track. Since LBNs start with 0, tracks start at multiples of track length.

   Example: From the RP06 specifications, the first 10 tracks start at LBNs: 0, 22, 44, 66, 88, 110, 132, 154, 176, 198.

3. Multiply sectors-per-track by tracks-per-cylinder to get sectors-per-cylinder. Establish the starting LBN for each cylinder on the disk by writing down the multiples of sectors-per-cylinder.

   Example: For an RP06, the first 10 cylinders start at the following LBNs: 0, 418, 836, 1254, 1672, 2090, 2508, 2926, 3344, 3762.

After you decide where on the disk you want to place your file, you create the file using RMS-11 placement control. In the process, you place area 0 (which will position the whole file, if the file is contiguous) at the location you calculated.

If you are using a high-level language, you can specify placement control by using RMSDES. If you are programming in MACRO-11, you can specify placement control through the use of allocation XABs.

6.5 BUCKET SIZE

Buckets are the units of access for indexed files. Bucket size is critical to the virtual address space required by a task and to the speed with which a task performs. There is, of course, a trade-off: the larger a bucket, the larger the task, but the faster it reads data:

- The speed of an RMS-11 operation is closely proportional to the number of I/O operations involved. For indexed files, the number of data transfers during a random retrieval operation is approximately equal to the depth of the index (in most cases, one more than the depth). That number includes only the I/O operations directly related to the record operation; other data transfers can be required to service the operation, including overlays and system overhead (discussed in Chapter 8).

  Therefore, the larger the buckets, the shallower the index, and the faster the random retrieval operation. Without other considerations, you should pick the largest possible bucket. The maximum bucket size allowed is 32 blocks.

- The larger the bucket, the more records it can contain, and sequential access can require fewer I/O operations.

However, there are other considerations. RMS-11 requires two I/O buffers, each the size of the largest bucket, when it connects a record access stream to an indexed file. By making bucket size smaller, you reduce the size of the buffers your task requires. Depending on the record operations your program requires, that virtual address space may be better used in overlay structure optimization (discussed in Section 8.2).

Therefore, you should set bucket size to some lower value that still allows good performance; a reasonable goal is an index depth of 2 or 3 (root at level 2 or 3), although very large files can require four levels of index, in addition to the data level (level 0).

Each area can have its own bucket size, but normally you should use the maximum size for all buckets:

- You should consider more than the size of your data record (plus the 7-byte header) when you calculate primary data bucket size:

  - Records that move from one bucket to another leave a 7-byte pointer.

  - Deleted records leave from 2 bytes to enough to hold the primary key to the whole record.

  Therefore, you should consider the predominant activity in the file:

  - If you intend to populate the file and then only read from it, you do not consider activity overhead. You must populate the file with records in ascending order by primary key value (discussed in Section 6.7).

  - If you intend to populate the file and then insert and/or delete a lot of records, you should allow for those activities in your bucket size calculations.

6.5.1 Bucket Size for Primary Index

You can calculate bucket sizes in two steps.

Step 1:

Calculate the following quantities for different bucket sizes (1, 2, 3, and so on):

NIRBK = ((512*BKS)-15)/(PKL+BPL) (Equation 1a)

NDRBK = ((512*BKS)-15-AO)/(RSZ+RFO) (Equation 1b)

where:

NIRBK is the number of index records per level 1+ index buckets

NDRBK is the number of data records per level 0 bucket

BKS is the bucket size as number of blocks

PKL is the primary key length in bytes

BPL is the bucket pointer length:

BPL is 3 for pointers to the first 65,535 blocks in the file

BPL is 4 for pointers to the blocks numbered between 65,536 and (2**24)-1

BPL is 5 for pointers to the blocks numbered between 2**24 and (2**32)-1

RSZ is the size of the record:

- data size for fixed-length records
- average record length for variable-length records

RFO is the record format overhead:

RFO is 7 bytes for fixed-length records

RFO is 9 bytes for variable-length records

AO is activity overhead. If any noticeable number of bucket splits will occur (due to random record insertions or UPDATE operations that increase record sizes), specify a value of at least 7 (more if bucket splits will be common). If insertion and deletion activity will occur often, significantly larger values of AO may be desirable, as well as occasional file reorganizations to reclaim space and improve access performance.

When you load a file using RMS-11 bucket fill factors, you preallocate space in each bucket for future activity. In such a case, Equation 1a becomes:

NIRBK = (FF-15)/(PKL+BPL)

and Equation 1b becomes:

NDRBK = (FF-15)/(RSZ+RFO)

where:

FF is the appropriate fill factor in bytes and has been adjusted to leave extra space in each bucket to accommodate future activity overhead.

Step 2:

Select bucket size for data and index areas where the following equation is true:

NIRBK**n ≥ NRF/NDRBK (Equation 2)

where:

NRF is the number of data records in the file

n is the depth of the index

This equation portrays the exponential relationship between the number of data records in a file and the depth of its index. You use the values for NIRBK and NDRBK you calculated in step 1.

a. Set up a grid (see the example after step 2e).

b. For each value of NIRBK, calculate the left side of Equation 2, for n = 2, 3, and for very large files, 4.

c. For each value of NDRBK, calculate the right side of Equation 2.

d. Where the equation is true, that is, the left side is greater than or equal to the right side, you have a valid combination of bucket sizes. The bucket size used to calculate the left side may be equal to the size used to calculate the right side, but it does not have to be.

NOTE

You gain no advantage using different index and data bucket sizes. RMS-11 requires two I/O buffers, both the size of the largest bucket defined for the file.

In fact, PDP-11 COBOL users must not choose different index and data bucket sizes.

e. Select one of the valid combinations according to your application's requirements.

NOTE

- Equation 2 is true only for files where records are inserted in order by ascending value of the primary key. See also Section 6.7.

- Bucket size is a step function of index depth. Therefore, intermediate bucket sizes generally waste address space.

For example, given a file where:

BKS = 4 ⟶ index depth of 3
BKS = 8 ⟶ index depth of 2

then bucket sizes of 5, 6, and 7 blocks would not normally be used, although you might choose a bucket size of 5 blocks if future file activity seemed likely to increase the index depth otherwise.

Example: For a file containing 50,000 200-byte fixed-length records with a 15-byte primary key, use the equations in steps 1 and 2 to fill in the following grid:

- Calculate values for NIRBK using Equation 1a and bucket sizes 1 through 6. Drop the remainder; use only the integer part of the result.

- Calculate values for NDRBK using Equation 1b and bucket sizes 1 through 6. Drop the remainder; use only the integer part of the result.

- Calculate the number of data buckets in level 0 (NRF/NDRBK) corresponding to bucket sizes 1 through 6. Round the result up to the nearest integer.

- Calculate NIRBK**2 for the values of NIRBK corresponding to bucket sizes 1 through 6. Round the result up to the nearest integer.

- Calculate NIRBK**3 for the values of NIRBK corresponding to bucket sizes 1 through 6. Round the result up to the nearest integer.

BKS	1	2	3	4	5	6
NIRBK	27	56	84	112	141	169
NDRBK	2	4	7	9	12	14
NRF/NDRBK	25000	12500	7143	5556	4167	3572
NIRBK**3	19683	175616	592704	1014049	2803220	4826810
NIRBK**2	729	3136	7056	12544	19881	28561

- To determine the combinations of bucket sizes where Equation 2 is true, compare the values in the NIRBK**3 row one at a time to each of the values in the NRF/NDRBK row. Where the NIRBK**3 value is greater than or equal to NRF/NDRBK, a valid bucket size combination exists.

  Example: The first NIRBK**3 value is 19683. This is less than 25000, the first NRF/NDRBK value, but it is greater than 12500, the second NRF/NDRBK value. Therefore, index bucket size of 1 (from NIRBK**3 row) and data bucket size of 2 (from the NRF/NDRBK row) is a valid combination.

- Compare the values in the NIRBK**2 row one at a time to each of the values in the NRF/NDRBK row. Where the NIRBK**2 value is greater than or equal to NRF/NDRBK, a valid bucket size combination exists.

  Example: The first NIRBK**2 value is 729. This is too small to use, as is the second value in the row. However, the third value is 7056. This is less than 25000 (the first NRF/NDRBK value) as well as the next two values, but greater than 5556, the fourth NRF/NDRBK value. Therefore, index bucket size of 3 (from NIRBK**2 row) and data bucket size of 4 (from the NRF/NDRBK row) is a valid combination.

As a result of the comparisons in steps 6 and 7 above, Equation 2 is true in the following cases:

NIRBK3**

DBKS	IBKS	IOB (bytes)
1	2	2048
2	1	2048

NIRBK2**

DBKS	IBKS	IOB (bytes)
1	6	6144
2	4	4096
4	3	4096

where:

DBKS is the data bucket size from the NRF/NDRBK row

IBKS is the index bucket size from the NIRBK**n rows

IOB is the maximum I/O space required by the largest bucket size of the pair

The choice of bucket size pairs depends on what you need to optimize most in the application: task size or access time. After you choose, make data and index bucket sizes equal to the larger size selected.

6.5.2 Bucket Sizes for Alternate Indexes

The selection of bucket sizes for alternate indexes follows the same procedure as that of primary key bucket sizes.

Step 1:

The records-per-bucket equations for alternate indexes are:

```
NIRBK = ((512*BKS)-15)/(AKL+BPL)
```

and

```
NDRBK = ((512*BKS)-15)/(AKL+(DBPL*DF)+4+DO)
```

where:

AKL is the alternate key length in bytes

DBPL is the data bucket pointer length:

DBPL is 4 for pointers to the first 65,535 blocks in the file

DBPL is 5 for pointers to the blocks numbered between 65,536 and (2**24)-1

DBPL is 6 for pointers to the blocks numbered between 2**24 and (2**32)-1

DF is the duplicate factor:

DF is 1 if you allow no duplicates

DF is the average number of records with the same key values for any key value present in the file

NOTE

The DF factor does not compensate enough if DF is greater than the number of data records that fit in a bucket. RMS-11 must then use continuation buckets to store the records with duplicate values.

DO is the duplicate overhead:

DO is 0 if you allow no duplicates

DO is 4 if you allow duplicates

No record movement or space/deletion overhead occurs in index buckets.

Step 2:

RMS-11 cannot load buckets in alternate indexes as efficiently as in the primary index because alternate key values inevitably fall in random order (unless you use the RMSIFL utility described in the RSX-11M/M-PLUS RMS-11 Utilities manual). The ideal values resulting from the equations in Section 6.5.1 must be reduced by a packing efficiency factor, unless RMSIFL is used to load the file.

Studies have shown that the packing efficiency factor for alternate keys is normally about 0.5. However, this factor applies only to the lower levels of the index and to the data level, and not to the root. The packing efficiency of any index's root is always 1.

Therefore, the index depth equation for alternate indexes is:

```
(PF**n)*(NIRBK**n) ≤ NRF/NDRBK
```

where:

PF is the packing efficiency factor.

Example: Using the file in the primary key example and adding a 10-byte first alternate key, allowing no duplicates, the following grid can be filled in (NRF=50,000 since there is one SIDR per data record):

BKS	1	2	3	4	5	6
NIRBK	38	77	117	156	195	235
NDRBK	28	57	85	113	142	170
NRF/NDRBK	1811	892	592	443	354	295
0.125*NIRBK**3	6859	57067	200202	474552	926860	1622240
0.250*NIRBK**2	361	1483	3423	6084	9507	13807

The index depth equation for alternate indexes is true in the following cases:

NIRBK3**

DBKS	IBKS	IOB (bytes)
1	1	1024

NIRBK2**

DBKS	IBKS	IOB (bytes)
1	3	3072
2	2	2048

Do not choose a bucket size smaller than that selected for the primary index (Section 6.5.1).

6.5.3 Program Syntax

RMS-11 requires bucket size as a whole number of blocks. However, some high-level language compilers require or allow you to specify the bucket size in number of records. This syntax can lead to a different number of records per bucket than you are counting on.

Example: A BASIC-PLUS-2 program contains the following clause in an OPEN statement that creates an indexed file:

```
BUCKETSIZE 5%
```

The record format is fixed; record length is 100 bytes. The compiler makes the following calculation:

```
100      bytes for the data
+ 7      bytes for the record header
---
107      bytes for each record
x 5      records specified in a bucket
---
535      bytes for the records in a bucket
+15      bytes for the bucket overhead
---
550      bytes required to be in the bucket
```

A bucket must be a whole number of blocks long, so the compiler rounds the bucket size to 2 blocks and passes that to RMS-11 to create the file.

However, 2 blocks contain 1024 bytes; that leaves 1009 bytes for record storage after the bucket overhead is subtracted. Since each record is 107 bytes long, the buckets that were originally supposed to contain only 5 records now can contain 9 (1009/107).

Bucket size can be set by RMSDES or by your application program, depending on the capabilities of your high-level language.

6.6 FILE ALLOCATION

RMS-11 requests the file control processor to allocate blocks to a file at three different points in the file's life:

- When the file is created
- When RMS-11 must dynamically extend the file to complete an operation
- When you explicitly instruct RMS-11 to extend the file

The allocation of blocks to a file takes time, mainly I/O time as the operating system performs its function. If RMS-11 has to request an allocation every time it requires a new bucket, this time can be a significant factor in an application's performance, especially during file population.

You can help optimize performance by minimizing allocation overhead in the following areas:

- Initial allocation
- Default extension quantity

6.6.1 Initial Allocation

Total allocation of an indexed file when you create it is most efficient.

The total allocation for a file is the sum of the prologue and the allocations for the different indexes that make up the file; an index's allocation is the sum of the allocations for all levels in the index. You should start with the primary level 0 and "build" each level of each index on paper, as shown in the following steps.

1. Calculate the number of buckets in level 0 (NBK@0):

   NBK@0 = NRF/NDRBK

   where:

   NRF is the total number of records in the file

   NDRBK is the number of data records in a bucket in level 0 (see Section 6.5.1 for the method of determining this value)

   NOTE

   The method described in Section 6.5.1 assumes that you will put records into the file in order by ascending primary key value. However, if you will be loading the file in random primary key value order, you should divide the NDRBK value obtained using the method described in Section 6.5.1 by 2. You will need twice as many data buckets.

2. Calculate the number of buckets in level 1 (NBK@1):

   NBK@1 = NBK@0/NIRBK

   where:

   NBK@0 is the number of buckets in level 0 (calculated in Step 1)

   NIRBK is the number of index records per bucket in the index (see Section 6.5.1 for the method of determining this value)

   NOTE

   The method described in Section 6.5.1 assumes that you will put records into the file in order by ascending primary key value. However, if you will be loading the file in random primary key value order, you should divide the NIRBK value obtained using the method described in Section 6.5.1 by 2 for every index level but the root. You will need twice as many index buckets.

3. Calculate the number of buckets in level 2:

   NBK@2 = NBK@1/NIRBK

4. Continue this sequence of calculations until you reach the root level, that is:

   NBK@n = 1 = NBK@(n-1)/NIRBK

   where:

   NBK@n is the number of buckets in the root, which is 1 by definition

   n is the index depth

5. Calculate the allocation in blocks for each level:

   AQ@0 = NBK@0 * DBKS
   AQ@1 = NBK@1 * IBKS
   .
   .
   .
   AQ@n = IBKS

   where:

   AQ@n is the allocation quantity in blocks for level n (0 for level 0, 1 for level 1, and so on)

   DBKS is the data bucket size in blocks

   IBKS is the index bucket size

6. Calculate the allocation for each alternate index as shown in Steps 1 through 5; see Section 6.5.2, for equations.

NOTE

Alternate indexes are normally populated in random key value order. Therefore, you should divide the NDRBK and NIRBK values obtained using the method described in Section 6.5.2 by 2 except for the root level.

7. The total allocation quantity for the file (ALQ) is the sum of the index allocation quantities plus the prologue:

   ALQ = PLG + AQPK + AQAK1 + ... + AQAKn

   where:

   n is the last alternate key defined for the file

The prologue of an indexed file can be from 2 to 84 blocks long. The size is the sum of the key descriptor blocks and the area descriptor blocks:

- VBN 1 describes the primary key (and contains other attribute information).
- Each key descriptor block covers up to 5 alternate keys.
- Each area descriptor block covers up to 8 areas.

Finally, RMS-11 extends the prologue to an integral multiple of bucket size if the criteria described in Section 6.5 are met.

Example: Given an indexed file of 100,000 fixed-length user data records with the following attributes, calculate a reasonable initial allocation size in blocks:

Data size = 200 bytes

Primary key = 20-byte string; no duplicates allowed

Alternate key = 8-byte packed decimal; no duplicates allowed

Data bucket size = indexed bucket size = 3 blocks

Calculate the primary index first:

1. AO = 0, so

   NDRBK = ((512*3)-15)/(200+7) = 7 data records per bucket

   NBK@0 = NRF/NDRBK = 100000/7 = 14,286 buckets in level 0

2. NIRBK = ((512*3)-15)/(20+3) = 66 index records per bucket

   NBK@1 = NBK@0/NIRBK = 14286/66 = 217 buckets in level 1

3. NBK@2 = NBK@1/NIRBK = 217/66 = 4 buckets in level 2

   NOTE

   If the number of buckets in the level under the root is very much less than the number of index records that fit in a bucket, you may be able to use a smaller bucket size without increasing the index depth.

4. NBK@3 = NBK@2/NIRBK = 4/66 = 1 bucket in level 3, the root

5. AQ@0 = NBK@0*DBKS = 14286*3 = 42,858 blocks in level 0

   AQ@1 = NBK@1*IBKS = 217*3 = 648 blocks in level 1

   AQ@2 = NBK@2*IBKS = 4*3 = 12 blocks in level 2

   AQ@3 = NBK@3*IBKS = 1*3 = 3 blocks in level 3

   AQPK = 43,521 blocks in the primary index

Now calculate the alternate index (DF=1, DO=0):

1. NDRBK = ((512*3)-15)/(8+(4*1)+4)

   = 89 data records per bucket

   NBK@0 = NRF/NDRBK = 100000/89 = 1124*2 = 2,248 buckets in level 0

   The doubling compensates for a packing efficiency of 0.5.

2. NIRBK = ((512*3)-15)/(8+3) = 138 index records per bucket

   NBK@1 = NBK@0/NIRBK = 17*2 = 34 buckets in level 1

3. NBK@2 = NBK@1/NIRBK = 1 bucket in level 2, the root

4. AQ@0 = NBK@0*BKS = 2248*3 = 6,744 blocks in level 0

   AQ@1 = NBK@1*BKS = 34*3 = 102 blocks in level 1

   AQ@2 = NBK@2*IBKS = 1*3 = 3 blocks in level 2

   AQAK = 6,849 blocks in the alternate index

5. Finally:

   ALQ = PLG + AQPK + AQAK1 = 3 + 43,518 + 6,849, or

   ALQ = 50,370 blocks for the whole file

This allocation can be done by RMSDES or by your application program, depending on the capabilities of your high-level language.

6.6.2 Default Extension Quantity

If the file cannot be totally allocated at creation time, you should establish a reasonable default extension quantity (DEQ) to minimize the number of (and the time spent on) file extensions. Even if the file is totally allocated when it is created, you should establish a reasonable DEQ in case the file gets bigger than planned.

A good basis for calculation is the number of records that are added to the file in a given period of time, such as a day; use the formula for allocation quantity in Section 6.6.1. The DEQ should equal a multiple of the bucket size.

If you do not specify a DEQ, it defaults to zero whether you create the file with RMSDES or a high-level language. RMS-11 responds to a DEQ of zero by requesting 4 times the bucket size in blocks from the file control processor each time it automatically extends the file.

The DEQ for the file can be set by RMSDES or by your application program, depending on the capabilities of your high-level language.

6.7 POPULATION TECHNIQUES

File population entails a large burst of records written into the file after it has been created and before it is made available for normal processing. You can populate a file with the RMSIFL or RMSCNV utility programs, or with an application program, depending on the capabilities of your high-level language.

The aim of populating an RMS-11 indexed file is to avoid bucket splits and record movement during the population and during later use of the file. The techniques to achieve this goal are:

- Inserting records in ascending order by primary key
- Use of fill numbers

6.7.1 Ascending Order by Primary Key

The best way to populate an indexed file is to insert the records in ascending primary key value order. You do not need to insert the records all at once. This technique:

- Minimizes population time
- Avoids the creation of RRV records, allowing RMS-11 to fill buckets with data records and thereby find records with the least access time.

Contrast this technique with records loaded in descending order by primary key value. In that case, you introduce the packing efficiency factor **p** to the primary key equations. Normally, **p** is 1, when you insert records in ascending order and the factor drops out of the equation, as shown here:

```
NIRBK**n  ≥  NRF/NDRBK
```

But when **p** < 1, the equation becomes:

```
(p**n)(NIRBK**n)  ≥  NRF/NDRBK
```

Since **p** is a fraction, the introduction of this factor reduces the left side of the equation, at times dramatically, thereby potentially increasing:

- The index depth needed to cover a specific number of data records
- Frequency of bucket splitting (an important factor in the time required to populate an indexed file)

As mentioned in Section 6.5.2, alternate indexes are a prime example of packing inefficiency, a situation avoided only with the RMSIFL utility. The best general approximation for **p** in the case of alternate indexes is 0.5, the value used in Section 6.5.2.

You can populate a file with records in ascending order by primary key as follows:

- Use the RMSIFL utility. This utility:
  - Sorts your input file into ascending order by the output file's primary key, if the file is not already sorted that way
  - Transfers the records from the input file to the output file

  RMSIFL uses techniques not available to you to further improve the population of an indexed file.
- Use the RMSCNV utility, specifying the mass-insertion mode (/MA) switch.
- Write a MACRO-11 program to populate the file and specify:
  - In the FAB, deferred write when you open the file
  - In the RAB, when you connect to the file: mass-insertion mode and sequential access mode

See the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide for more information.

Be sure to sort your input records into ascending order by the indexed file's primary key before you run the program.

6.7.2 Random Insertions after File Population

If you will be inserting records into an indexed file after it is populated, you should consider ways to optimize these operations:

- If the new records to be inserted span the full range of primary key values, you should use a bucket fill size.
- If the inserted records are sorted into ascending order by primary key value and added at the logical end-of-file, you should use mass-insertion mode.

6.7.2.1 Bucket Fill Size - You can optimize for evenly distributed random insertions by leaving free space in buckets during the initial population of the file. To do this, you specify a bucket fill size as a set amount of bytes for each area in your file. Normally, RMS-11 ignores this number, but you can direct RMS-11 to obey it: RMS-11 then fills each bucket in the file to the level specified by the number.

Example: Your bucket size is 2 blocks; you set the bucket fill size to 768 bytes. When you tell RMS-11 to honor the fill size, it only uses 768 out of 1024 bytes in each bucket -- the buckets are logically three-quarters size.

You use the bucket fill size when you populate a file to improve its performance during normal operations: if free space is available in every bucket in the file, any record randomly inserted into the file is likely to fit without causing a bucket split.

The size of the bucket fill size depends on:

- The amount of insertion activity you expect.

  Allow room (including record header) for the number of records you will add to each bucket during normal operations. Occasional insertions might not warrant the use of bucket fill sizes, whereas heavy insertion can require room for multiple additional records in each bucket to optimize, but not eliminate, bucket splitting activity.

- The type of bucket (data or index) involved.

  Because of the difference in record sizes and frequency of insertion, data and index buckets should normally have different bucket fill sizes.

Example: The file contains 240-byte fixed-length records with a primary key field 24 bytes long. To optimize random insertions, the fill size for data buckets should therefore be at most: bucket length minus bucket overhead (15) minus record length (240) minus record overhead (7). This number leaves room for one data record.

This same bucket fill size for index buckets leaves room for 9 index records. A more reasonable bucket fill size for index buckets is: bucket length minus bucket overhead minus 2 times 27 bytes. This

number leaves room for 2 index records, where: the primary key length (24) plus the bucket pointer length (3) equals the index record length (27).

See Section 6.5.2 for a more complete discussion.

NOTE

RMS-11 ignores a bucket fill size of less than 50 percent of the bucket length and uses the 50 percent figure.

The bucket fill size for a file can be set by RMSDES or by your application program, depending on the capabilities of your high-level language.

6.7.2.2 Mass Insertion - You use mass-insertion mode when you have a series of records to add to an indexed file and:

- You have sorted the records into ascending order by the file's primary key.
- The lowest key value in the records is greater than the highest key value in the file; that is, the records will be inserted at the logical end-of-file.

While the mass-insertion bit is on, RMS-11 performs a PUT operation normally (see Section 5.3.1) except that it:

- Does not unlock the primary level 0 data bucket
- Keeps a pointer to the primary level 1 bucket that pointed to the proper level 0 bucket

These extra steps enable RMS-11 to:

- Write the next record without following the primary index (if the mass insertion bit is still on).
- Rapidly split the primary level 0 bucket when it is full: since RMS-11 has a pointer to the primary level 1 bucket that will contain the index record for the new bucket, it can update that bucket without following the index.

By using these techniques, RMS-11 can extend the primary level 0 bucket by bucket, packing records into the buckets in the order they are written. As each bucket becomes full, RMS-11 creates a new one, beginning with the next record inserted, and notes its existence in the primary level 1 index bucket.

NOTE

Mass insertion significantly improves performance for single-key indexed files. The percentage of improvement lessens with each additional key defined in the file.

You can enhance mass insertion performance by using deferred write (see Section 7.4).

CHAPTER 7

RECORD AND FILE PROCESSING OF INDEXED FILES

The record and file processing capabilities described in RSX-11M/M-PLUS RMS-11: An Introduction are available for indexed files. This chapter discusses the operations and their implementation and restrictions with indexed files.

7.1 ACCESS SHARING

Access sharing can be specified for indexed files as described in the following sections. See Section 2.2.3 for general information on shared access.

7.1.1 Record Access to Indexed Files

Indexed files allow fully interlocked read/write sharing, dependent upon the compatibility of the access and sharing declarations of multiple accessors, as follows:

- If you have requested read/write access, your request will be denied unless **all** other accessors have allowed read/write sharing. (Otherwise, your read/write access request will conflict with the sharing declaration of at least one other accessor.)

- If you have not permitted read/write sharing, your request for read/write access will be denied if any other read/write accessor is present. (In this case, the read/write accessor does not meet the requirements of your sharing declaration.)

7.1.2 Block Access to Indexed Files

Because block access bypasses the record structure and interlocking algorithms used with indexed files, read/write sharing cannot be permitted. Any read/write sharing declaration is converted internally to read-only before the file is processed (this is similar to record-accessed sequential files).

Thus, multiple read-only accessors (regardless of their sharing declarations) can share indexed files concurrently using block access, as long as no read/write record accessor is present. Read-only block accessors can share files with read-only record accessors. In addition, a single read/write accessor can access an indexed file using block access (regardless of sharing declaration) as long as no other accessor of any kind is present.

Other combinations are rejected: the access and sharing declarations are incompatible.

7.2 RECORD AND STREAM OPERATIONS

The following record and stream operations can be performed on indexed files. See also the discussions of read- and write-type record operations in Chapter 5.

CONNECT
DELETE
DISCONNECT
FIND
FLUSH
GET
PUT
REWIND
UPDATE

In all record operations, RMS-11 establishes the current record (if any) and next record (if applicable) context. If any record operation fails, RMS-11 normally sets the current record to none and does not change the next record.

NOTE

For more information on the RMS-11 error codes referred to in the following sections, see the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide.

7.2.1 CONNECT

A CONNECT operation affects the context for the access stream as follows:

- Current record -- There is no current record. Any operation requiring a current record fails at this point.
- Next record -- The next record is the first record in the file according to the collating sequence of the specified key of reference.

Example: In an indexed file with multiple keys, the next record varies by the key specified in the instruction initiating the CONNECT operation:

- If the primary key is specified, the next record is the first record in primary level 0, the one with the lowest primary key value in the file.
- If an alternate key is specified, the next record is indicated by the first SIDR in the alternate index's level 0; the record itself can be located anywhere in the primary level 0.

7.2.2 DELETE

In a DELETE operation, RMS-11 flags the header of the current record to indicate that it is a deleted record. The prerequisite GET or FIND operation brought the bucket containing the record into the I/O buffer.

Then, RMS-11 writes the bucket over its original location on the disk, unless you specified deferred write (see Section 7.4.2).

A DELETE operation affects the context for the access stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.
- Next record -- Unchanged.

7.2.3 DISCONNECT

A DISCONNECT operation destroys the context for the access stream. You cannot resume this context by reconnecting the stream.

7.2.4 FIND

Section 5.3.2 describes how RMS-11 performs a key-access FIND operation. Section 5.4 describes how RMS-11 performs a sequential-access FIND operation.

If the record does not exist or has been deleted, RMS-11 returns an error code depending on the access mode:

- In sequential-access mode, the error code is ER$EOF.
- In key-access mode, the error code is ER$RNF.
- In RFA-access mode, the error code is:
  - ER$RFA -- no valid record has ever existed at the specified location.
  - ER$DEL -- the record header indicates that the record was deleted.

A FIND operation affects the context for the access stream as follows:

- For a sequential-access FIND operation:
  - Current record -- Is set to the value of the record found.

    Example: You have connected a stream to an indexed file, specifying 0 as the key of reference. There is no current record, but the next record is the first record in primary level 0. If you execute a sequential-access FIND operation, the current record is set to this record.
  - Next record -- Is set to the record logically following the current record in the index of reference.

NOTE

RMS-11 enacts this logical sequence only when it actually accesses the next record:

1. RMS-11 locates the current record, reading a bucket if necessary.

2. RMS-11 locates the record logically following the current record, reading another bucket if necessary.

If the indexed file is shared, the actual record in the next record position can change between the operation that accesses the current record and the one that finds the next record.

Example: From the previous example, the next record is the record in the file with the next higher primary key value.

- For a key-access or RFA-access FIND operation:
  - Current record -- Is set to the record found, that is, the record identified by the RFA.
  - Next record -- Unchanged.

Example: In the previous examples, you did a sequential-access FIND operation after connecting the stream to the file. You now execute an RFA-access FIND operation. The current record is set to the record specified, but the next record is not changed. Therefore, if you do another sequential-access FIND operation, the current record will be set to the second record in primary level 0, not the record following the one found by RFA.

You use a FIND operation instead of a GET operation for two reasons:

1. It is faster because the record is not moved to the user buffer. Although the time required to move a record from one part of memory to another is very short, do not expend it if you do not need to.

2. It does not change the next record in key-access mode or RFA-access mode. This convention allows you to branch off sequential processing for the purpose of updating or deleting records, and keep your place.

You can use a FIND operation in the following ways:

- To skip records in sequential access mode by initiating successive FIND operations.
- To establish a current record for a DELETE or UPDATE operation.
- To determine the existence of a record by using key-access mode.

7.2.5 FLUSH

A FLUSH operation does not affect the context for the access stream.

7.2.6 GET

Section 5.3.2 describes how RMS-11 performs a key-access GET operation. Section 5.4 describes how RMS-11 performs a sequential-access GET operation.

If the record does not exist or has been deleted, RMS-11 returns an error code depending on the access mode:

- In sequential-access mode, the error code is ER$EOF.
- In key-access mode, the error code is ER$RNF.
- In RFA-access mode, the error code is:
  - ER$RFA -- No valid record has ever existed at the specified location.
  - ER$DEL -- The record header indicates that the record was deleted.

A GET operation affects the context for the access stream as follows:

- Current record -- Is set to the value of the record read.
- Next record -- Is set to the record logically following the current record in the index of reference.

NOTE

RMS-11 enacts this logical sequence only when it actually accesses the next record:

1. RMS-11 locates the current record, reading a bucket if necessary.
2. RMS-11 locates the record logically following the current record, reading another bucket if necessary.

If the indexed file is shared, the actual record in the next record position can change between the operation that accesses the current record and the one that finds the next record.

7.2.7 PUT

Section 5.3.1 decribes how RMS-11 performs a key-access PUT operation.

A PUT operation affects the context for the access stream as follows:

- For a sequential-access PUT operation:
  - Current record -- None. Any operation requiring a current record fails at this point.
  - Next record -- Undefined. The record retrieved by a sequential-access FIND or GET operation at this point is not specified.
- For a key-access PUT operation:
  - Current record -- None. Any operation requiring a current record fails at this point.
  - Next record -- Unchanged.

7.2.8 REWIND

A REWIND operation sets the context of the access stream to a logical beginning of the indexed file. In doing so, the operation affects the context for the stream as follows:

- Current record -- None. Any operation requiring a current record fails at this point.
- Next record -- is set to the first record in the file according to the specified key of reference.

7.2.9 UPDATE

In an UPDATE operation, RMS-11 moves the specified record from the task's user buffer to the I/O buffer, replacing the current record set by the previous GET or FIND operation. Then, RMS-11 writes the bucket over its original location on the disk. Section 5.3.3 describes the UPDATE operation in detail.

An UPDATE operation requires a valid current record. Therefore, an UPDATE operation should follow a successful GET or FIND operation; otherwise, RMS-11 returns the error code ER$CUR. This error does not affect the original record in the file on disk.

An UPDATE operation affects the context for the access stream as folows:

- Current record -- None. Any operation requiring a current record fails at this point.
- Next record -- Unchanged.

7.3 RECORD TRANSFER MODES

You can manipulate records either in the I/O buffer or in your program's user buffer. Each of these options is called a record transfer mode. You can change record transfer mode at run time, even between record operations. Figure 7-1 illustrates the RMS-11 task structure.

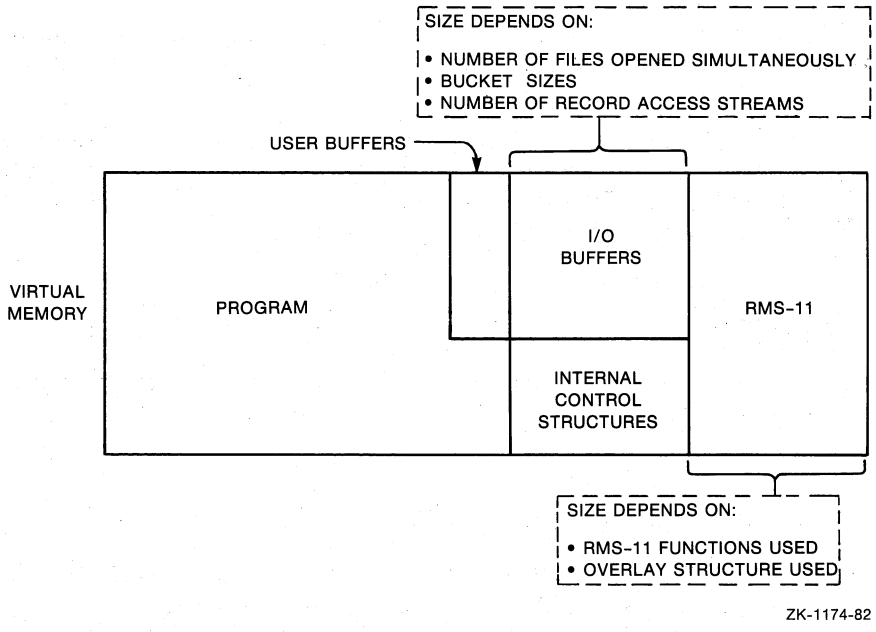


Figure 7-1: RMS-11 Task Structure

7.3.1 Move Mode

Move mode is the default record transfer mode for all programming languages and all file organizations.

- On GET operations, RMS-11 moves the record from the I/O buffer to the user buffer before returning control to your program.
- On PUT and UPDATE operations, your program assembles the record to be written into the file in the user buffer, and during the operation, RMS-11 moves the data into the I/O buffer before updating the file.

7.3.2 Locate Mode

Locate mode enables your program to manipulate records in the I/O buffer, eliminating the data transfers between it and the user buffer. However, when you specify locate mode, RMS-11 uses it only when such usage does not compromise data integrity. Otherwise, RMS-11 uses move mode. Therefore, your program must still contain a user buffer.

Example: RMS-11 uses move mode instead of locate mode when an indexed file is shared.

Example: RMS-11 uses move mode instead of locate mode if you opened the file indicating that you were going to perform UPDATE operations on it.

RMS-11's use of move mode instead of locate mode is transparent to your program as long as you use RMS-11 facilities to access the record data.

For indexed files, your program can only perform GET operations in locate mode. See your high-level language documentation to determine whether the language supports locate mode and, if it does, what the exact programming techniques are.

7.4 I/O TECHNIQUES

You can use the following techniques to improve the performance of record operations.

7.4.1 Asynchronous Record Operations

Within each record access stream, your program can perform any record operation either synchronously or asynchronously. In synchronous operations, RMS-11 returns control to your program after the operation completes, either successfully or with an error.

When you execute an asynchronous operation, RMS-11 may return control to your program before the operation is complete. The program continues processing while the physical transfer of data between disk and memory is carried out. However, you must not initiate another record operation on that stream until the first operation ends; otherwise, RMS-11 returns the error code ER$ACT. See your high-level language documentation for asynchronous techniques.

7.4.2 Deferred Write

Normally, each write-type record operation (DELETE, PUT, and UPDATE) results in a bucket being written to disk. This convention emphasizes data integrity: you know that when a write-type operation ends successfully, the file reflects that operation.

However, you can improve the performance of mass-insert sequential (by primary key) PUT or DELETE operations by using deferred write. Basically, deferred write directs RMS-11 to write a bucket out to disk only when RMS-11 must use the I/O buffer for some other purpose.

Example: Your records are 114 bytes long and the bucket size is 2 blocks. During sequential write-type operations, deferred write could cause I/O operations per bucket to drop from 9 to 1.

Deferred write offers little or no benefit to random write-type operations or read-type operations of any mode.

NOTE

Deferred write should only be used with mass-insert PUT operations. Although not illegal, deferred write is essentially invalidated while an indexed file is shared by multiple tasks -- except when you are also using mass insertion mode. In the non-mass-insertion, write-shared environment, every write-type operation results in an I/O operation so that:

- The bucket locked by the prerequisite GET or FIND (for UPDATE and DELETE operations) or by the PUT operation can be released.
- The new data is available to the other tasks or streams.

7.4.3 Multiple Buffers

When you open an indexed file, RMS-11 normally sets up two bucket-sized I/O buffers in your task's address space. RMS-11 uses both buffers for record operations. However, you can direct RMS-11 to use more than the two buffers.

RMS-11 uses any extra buffers to keep, or cache, index root buckets if either of the following is true:

- The file is shared only by tasks with read-only access.
- The file is not shared.

RMS-11 caches the roots as it uses them. Therefore, only keys specified or implicit in record operations have their index root buckets cached:

- During normal PUT operations, RMS-11 typically accesses all indexes in a file. You benefit from root caching only when the number of extra buffers equals or exceeds the number of indexes.
- During mass-insertion mode PUT operations, one extra buffer provides some benefit, regardless of sharing and number of indexes. If the file is not being shared, you benefit from root caching only when you provide one more extra buffer than indexes.
- During GET operations, RMS-11 accesses one index (associated with the key of reference). You benefit from root caching when you provide an extra buffer for each different key you reference.
- During UPDATE and DELETE operations, RMS-11 accesses the alternate indexes where a SIDR must be inserted or deleted. You benefit from root caching when you provide an extra buffer for each alternate index affected.

While root caching saves one disk read per index accessed, you may be able to employ the address space used for the extra buffers more profitably to optimize RMS-11 overlays (see Chapter 8).

7.4.4 Multiple Access Streams

RMS-11 allows each program to use multiple streams on an indexed file.

7.4.5 Sequentially Reading Write-Shared Files

If your task is trying to read sequentially by primary key an indexed file that is write-shared, you can improve performance by specifying write-access as well.

Example: Include in your BASIC-PLUS-2 OPEN statement the clauses ACCESS MODIFY and ALLOW MODIFY.

When there is a possibility that your task will update a record (established when it opened the file), RMS-11 locks the bucket when your task gets a record and holds the bucket in the task's I/O buffer. If your task then gets records sequentially, RMS-11 finds them in memory. When a record in a different bucket is specified, RMS-11 unlocks the previous bucket and repeats the procedure with the new one.

However, if your task opens a file in a read-only and write-sharing mode, RMS-11 does not retain the lock on the buckets read; RMS-11 reaccesses the file for each subsequent GET operation, although it does not start at the root and go down the index again.

7.5 FILE AND DIRECTORY OPERATIONS

The following file and directory operations can be performed on indexed files:

CLOSE
CREATE
DISPLAY
ENTER
ERASE
EXTEND
OPEN
PARSE
REMOVE
RENAME
SEARCH

See your high-level language documentation for a description of the support provided.

CHAPTER 8

TASK BUILDING AND COMMON OPTIMIZATION TECHNIQUES

Chapter 2 introduced four application design considerations. Two of those design considerations, sharing and ease of design, were discussed there. The others, speed and space, were the underlying concepts for the file and task design discussions in Chapters 4 through 7. They are also the prime considerations for the use of the techniques discussed in this chapter.

You can optimize the speed of and the space used by your application by:

- Improving the structure of each task. This includes:
  - The method of combining your program with RMS-11 routines (discussed in Section 8.1)
  - Program development, including the sequence of operations (discussed in Section 8.2)
- Using all features of the environment in which the task runs. Especially important is optimizing virtual-to-logical-block mapping (discussed in Section 8.3), but there are other factors as well (discussed in Section 8.4).

8.1 TASK BUILDING WITH RMS-11 ROUTINES

The software routines that perform the RMS-11 functions are distinct from your programming language. These routines must be combined with your program as follows:

1. A compiler or the assembler converts your program to object code. In the process, the RMS-11 routines that your program uses are listed as unresolved global references.
2. The task builder combines object modules into an executable task. It resolves the RMS-11 global references with the RMS-11 routines in either:
   - An object module library named RMSLIB.OLB
   - An RMS-11 resident library

   You must decide whether RMS-11 is to be overlaid or nonoverlaid when combined with your program to form a task. This section should guide your choice.
3. When the task builder is finished, your task is ready to run.

Figure 8-1 illustrates this sequence, from source program to object code to executable task.

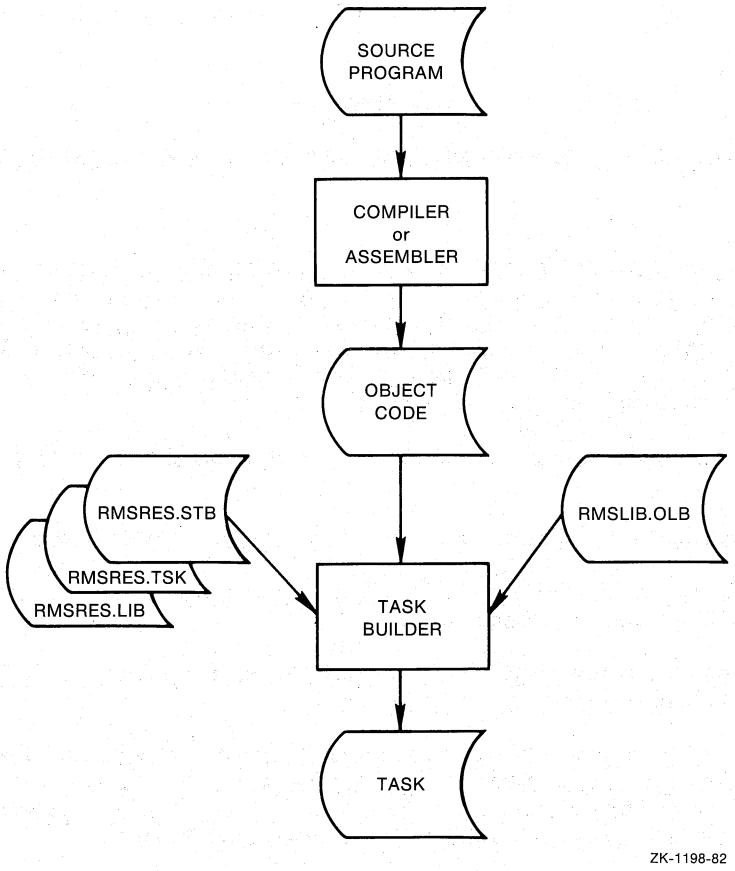


Figure 8-1: Source-to-Task Sequence

The RMS-11 routines that become part of your task can be overlaid or nonoverlaid. Overlays are task segments that can run independently; therefore, they do not have to be available to the task at the same time and can share address space. When a segment is needed, the operating system makes it available, replacing (overlaying) a segment no longer being used. By interchanging its parts, a task can run even though it is too large to be executed as one piece.

Nonoverlaid RMS-11: For synchronous operations, the task builder concatenates the RMS-11 routines with your program, that is, without overlays, if you add the following term to the command line:

```
,LB:[1,1]RMSLIB/LB
```

For asynchronous operations, use the following term:

```
,LB:[1,1]RMSLIB/LB:R0EXEC:R0RSET:R0WATB,RMSLIB/LB
```

The task builder extracts from RMSLIB.OLB only those routines required by your program. These routines contribute from 8KB to 44KB to the task size. Note that if other portions of your task are overlaid, you

can use nonoverlaid RMS-11 **only** if all references to RMS-11 take place in the root segment of your task.

Overlaid RMS-11: If the sum of your program plus RMS-11 code is greater than 64KB, there is not enough user address space for your task to run without overlays.

NOTE

Although you can overlay segments of your program, this section is devoted to the best use of RMS-11 overlays. Therefore, all references to "overlays" mean "overlays in RMS-11 routines."

Overlays can take one of two forms:

1. Disk-resident overlays

   The overlay segments are part of the task image, and they remain on disk until they are needed. When a routine is required, the operating system reads the overlay segment containing that routine into the task's address space, replacing a segment no longer needed. Section 8.1.1 discusses disk-resident overlays.

2. Memory-resident overlays

   The overlay segments are part of a task image maintained separately in memory. When a routine is needed, the operating system maps the segment into the task's address space with two of its active page registers (APRs). Section 8.1.2 discusses memory-resident overlays.

Figure 8-2 illustrates nonoverlaid and overlaid (disk resident and memory resident) task structure.

8.1.1 Disk-Resident Overlays

One disk-resident overlay can address others, which can address others, and so on. This chain of calls defines the overlay structure of a task. You describe this structure in a file with overlay description language (ODL) statements (described in your task builder manual).

You must generate an ODL file for each overlaid task and supply it to the task builder. However, you do not normally create ODL statements for the RMS-11 portion of your task, but instead refer to the RMS-11 ODL files provided on your system. The RMS-11 installation process provides overlay descriptions in two forms:

- A series of standard ODL files describing disk-resident RMS-11 overlay structures that require differing amounts of task address space. The larger structures may run faster; you should use the best one for your application.
- A prototype ODL file you can modify, making overlay segments larger if there is room in your address space, or eliminating them if your program does not use those functions.

The installation process places these files in account [1,1] on logical device LB:. If you are using asynchronous RMS-11 operations, you must select special RMS-11 modules, as indicated in these files.

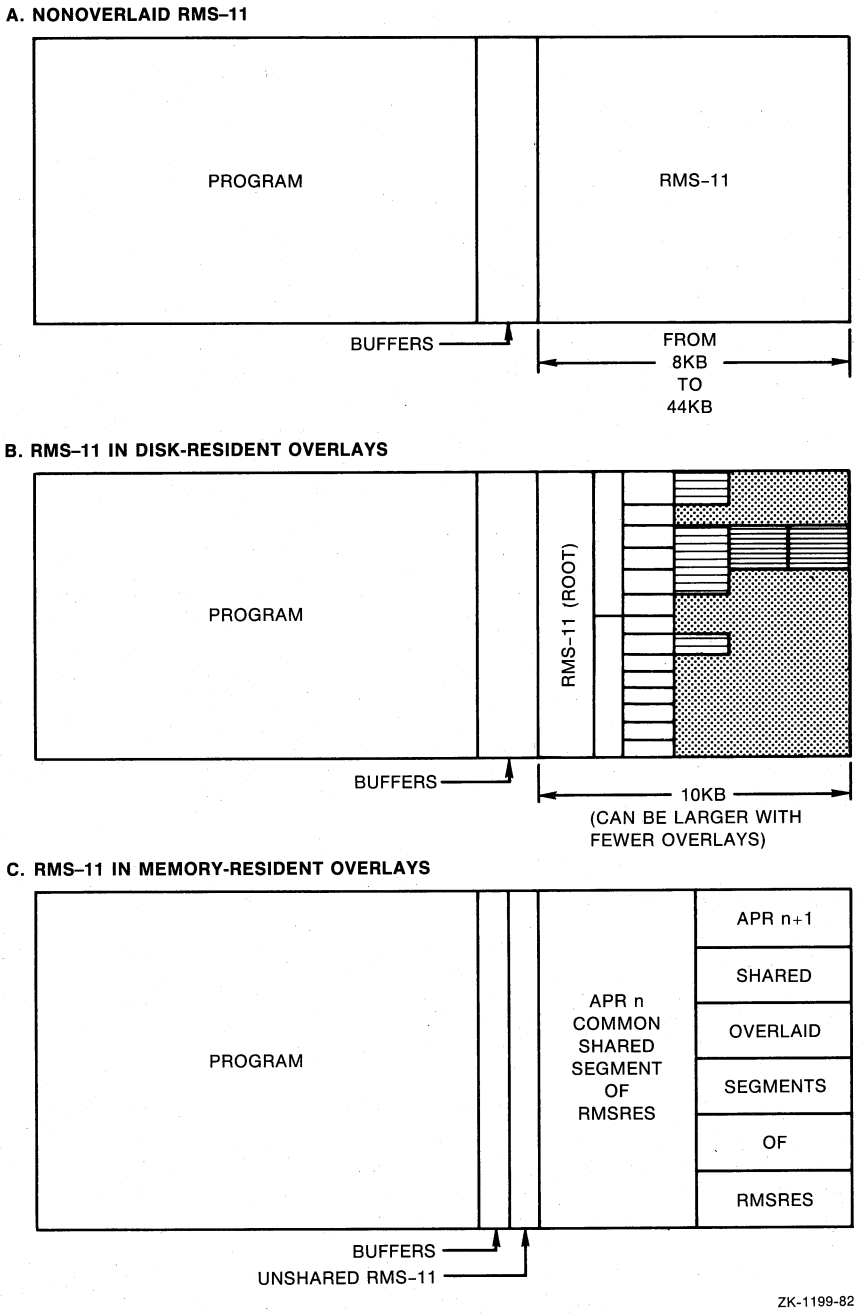


Figure 8-2: RMS-11 Tasks

Each high-level language has its method of generating the ODL file for your program and referencing the RMS-11 ODL files. They normally generate the following hierarchy of files:

- program-name.CMD

  You supply this indirect file to the task builder. The file contains the appropriate command lines(s) for the task builder and references a primary ODL file.

- program-name.ODL

  This primary ODL file determines the general structure of the task and references secondary ODL files, including RMS-11 ODL files, such as a standard file or your modification of the prototype file.

See your high-level language documentation for more details.

If you are a MACRO-11 programmer, however, you must write your own ODL file. Make sure the file contains the following terms, if you want to use RMS-11 disk-resident overlays:

- The factor names RMSROT and RMSALL in the .ROOT statement. RMSROT represents a set of concatenated modules that perform functions common to multiple RMS-11 operations. You must concatenate RMSROT with your program's root so that it is memory-resident while the task runs.

- An indirect reference to an RMS-11 ODL file, either a standard file or your customized version of the prototype, in the form:

  ```
  @file-name
  ```

  This RMS-11 ODL file resolves the references to RMSROT and RMSALL. For example:

  ```
                .ROOT   USRROT-RMSROT-USRSEG,RMSALL
  USRSEG:       .FCTR   (USR1,USR2,USR3)
  @LB:[1,1]RMS11X
                .END
  ```

8.1.1.1 ODL Files - DIGITAL provides the following standard ODL files. Do not change these files; make a copy in your own directory if you want to modify one for your own use.

RMS11S.ODL Structured to add about 6.5KB to the task size, this file provides only sequential and relative file organization routines in 11 overlay segments.

RMS11S.ODL is designed to use minimal virtual address space for the support provided. Because of this, file operation performance and performance where GET, PUT, and/or UPDATE operations on sequential files are intermixed will be slower than when using other ODL structures.

RMS11X.ODL Structured to add about 10KB to the task size, this file provides sequential, relative, and indexed file organization routines in 35 overlay segments.

RMS11X.ODL is designed to use minimal virtual address space for the support provided. Because of this, performance for record operations on indexed files will usually be slower than when using the RMS12X or DAP11X ODL structures.

RMS12X.ODL Structured to add about 12KB to the task size, this file provides sequential, relative, and indexed file organization routines in 13 overlay segments.

RMS12X.ODL is designed to offer a good compromise between performance for record operations on indexed files and use of task virtual address space.

DAP11X.ODL Structured to add about 14KB to the task size, this file provides sequential, relative, indexed, and (on systems with the required DECnet support) remote access facilities in 16 overlay segments.

DAP11X.ODL is designed to use minimal virtual address space for the support provided. For local access, however, it uses the efficient structure contained in RMS12X.ODL.

RMS12S.ODL Structured to add about 9KB to the task size, this file provides only sequential and relative file organization routines in 5 overlay segments.

RMS12S.ODL is designed to offer a good compromise between performance and use of task virtual address space.

8.1.2 Memory-Resident Overlays

The RMS-11 resident libraries contain RMS-11 routines in re-entrant executable code. Tasks that use RMS-11 can be built with global references resolved in the resident library RMSRES, if this library is present in your system.

While it is executing one of these tasks, the operating system uses two of the task APRs to map references from the task to the resident library. Therefore, any time the task requires an RMS-11 routine, the operating system changes the APRs to point to the segments of the resident library that contain the routines for the operation.

This mapping is called memory-resident overlaying. Because the overlay segments are in memory, the operating system does not perform an I/O operation to provide the routines (as it does with disk-resident overlays).

8.1.2.1 Task Building against the RMS-11 Resident Library - You build tasks, directing the task builder to resolve global references with a library, with one of the following sequences of commands:

```
TKB>command-string
TKB>/
ENTER OPTIONS:
TKB>LIBR=RMSRES:RO
TKB>//
```

or, to use RMSRES in a cooperating cluster of libraries that share the same set of task APRs:

```
TKB>command-string
TKB>/
ENTER OPTIONS:
TKB>CLSTR=LIB1,LIB2,RMSRES/RO
TKB>//
```

NOTE

Not all libraries can be clustered with RMSRES. See your high-level language, FCS-11, or other documentation for details. Clustering RMSRES may decrease performance for some applications.

See your task builder manual for a description of the command string.

You must also do one of the following:

- Specify LB:[1,1]RMSRLX.ODL as the RMS-11 secondary ODL file name in your primary ODL file.
- Merge the contents of RMSRLX.ODL into your own ODL file or into your task builder command string.

If your system provides the required DECnet support and the RMS-11 library DAPRES, and you want to use the RMS-11 remote access facilities, include the entry DAPRES in the task builder CLSTR option (after the RMSRES entry), as described above, and use LB:[1,1]DAPRLX.ODL instead of RMSRLX.ODL.

If you are using asynchronous RMS-11 operations, you must select special RMS-11 modules, as indicated in RMSRLX.ODL or DAPRLX.ODL.

On RSX-11M-PLUS systems that support supervisor mode, you may instead choose to use RMSRES as a supervisor-mode library. Because this configuration uses two otherwise idle supervisor-mode APRs to map most of the RMS-11 code, the impact of the RMS-11 code on your user-mode virtual address space is reduced to the absolute minimum; there also may be slight performance advantages over the clustered RMS-11 configuration.

To use RMSRES as a supervisor-mode library, use the following sequence of commands:

```
TKB>command-string
TKB>/
ENTER OPTIONS:
TKB>RESSUP=LB:[3,54]RMSRES/SV:0
TKB>//
```

See your task builder manual for a description of the command string.

You must also include the modules

```
LB:[1,1]RMSLIB/LB:R0EXSY:R0AUTS:R0IMPA
```

in the root of your task, using either the task builder command string or an ODL file.

In addition, you should include the module

```
LB:[1,1]RMSLIB/LB:RMSSYM
```

if your task requires global definitions of the user-visible RMS-11 symbols. If your task uses asynchronous operations, replace the module R0EXSY above with R0EXEC. To include remote access (DAP) support, replace the module R0AUTS above with R0AULS, and include DAPRES as a LIBR or CLSTR option in the task builder command sequence. If you are using DAPRES and the RMS-11 asynchronous facilities together, you must also include

```
LB:[1,1]SYSLIB/LB:AUTOT
```

in the task.

If you are using resident libraries, a BPT trap will be generated and R0 will contain the value 175744 (the error code ER$LIB). This can happen if not all segments of the library are installed or if the version numbers of one or more segments do not match the root sgement, the RMSDAP code, or the task itself. See your system manager to properly install the library.

8.1.2.2 Using RMS-11 Operations from within Your Own Resident Library - You can invoke RMS-11 operations from within a resident library if you task build that library to include the module R0RMSC from RMSLIB.OLB and to exclude the following symbols using the task builder GBLXCL option:

```
.SAVR1,$RMENT,$RMREM,$RMSEA,$RMERA,$RMOPE,
$RMPAR,$RMCRE,$RMREN,$RMDSP,$RMEXT,$RMCLO,
$RMCON,$RMDIS,$RMGET,$RMPUT,$RMUPD,$RMDEL,
$RMFIN,$RMTRU,$RMFRE,$RMREL,$RMFLU,$RMRWI,
$RMNXT,$RMSPA,$RMREA,$RMWRI,$RMWAI
```

Such a resident library may be clustered with the RMS-11 resident libraries only if it contains absolutely no pure or impure data (such as, RMS-11 structures and call parameter blocks, RMS-11 internal structures and buffers, file specifications, key or record buffers, and so on) that RMS-11 needs during its processing. If such a library is a default member of the cluster and has a non-null root segment, it must not contain an RMS-11 get-space routine or the completion routine for any asynchronous RMS-11 operation in its root segment.

When you build tasks that use your library, you include RMS-11 resident library support as described in Section 8.1.2.1.

If instead you want to use RMS-11 disk-resident overlays, you can do so as described in Section 8.1.1, and must also include ORG$ statements in the task's root segment appropriate to the needs of your library. To use nonoverlaid RMS-11 routines, you must tailor and assemble your own copy of the source module LB:[1,1]R0RMS1.MAC, include it in the root of your task, and build nonoverlaid RMS-11 as described in Section 8.1.

NOTE

In RSX-11M-PLUS systems that support supervisor mode: under no circumstances can RMS-11 operations be used in a supervisor-mode resident library.

8.1.2.3 Deciding Between Types of Overlays - You should normally use the RMS-11 resident libraries whenever possible, for the following reasons:

- Program execution speed will typically be faster than with disk-resident overlaid RMS-11, and nearly as fast as with nonoverlaid RMS-11.
- Virtual address space required in your program will usually be less than with nonoverlaid RMS-11, and may be less than with disk-resident overlaid RMS-11, if you are able to cluster RMS-11 with other libraries or, on RSX-11M-PLUS systems that support supervisor mode, use RMSRES as a supervisor-mode library.
- Your tasks will build significantly faster and take up significantly less space on disk than with other RMS-11 configurations.
- You will usually not need to rebuild your tasks when enhancements or corrections to RMS-11 are issued.
- Because the RMS-11 resident libraries can be shared among multiple programs, using them will often result in more efficient use of the system's physical memory.

Reasons that you might not use RMS-11 resident libraries include:

- Your system manager has not included them in your system, perhaps because very little system memory is available and RMS-11 is seldom used.
- You require indexed file organization support but your RSX-11M system manager has chosen the optional version of RMSRES for RSX-11M that does not include this support.
- The special virtual address requirements of your task do not permit the use of RMS-11 resident libraries.

8.2 PROGRAM DEVELOPMENT

You should consider performance while you are writing an application program:

- Your program's flow of operations can either cooperate with or fight against the RMS-11 code structure.
- Task-building consumes a significant portion of your machine resources. Minimize that time when you can.

8.2.1 Flow of Operations Should Reflect RMS-11 Code Structure

The overlay process causes a significant portion of the I/O performed for a program with disk-resident overlays. Using RMSRES, an RSX-11M-PLUS system may be forced to perform I/O paging operations to access RMS-11 code segments if physical memory is in short supply. You should structure the task to maximize the time each segment stays in memory and thus minimize the number of I/O operations. You do this by placing similar RMS-11 operations together in your program. This process also makes you aware of the nature of the operations your program is performing:

- File-related operations

  File-related operations are generally required at the beginning and end of processing. Therefore, they are fairly easy to group.

  Example: Open all files that the program uses and set up all record access streams at the beginning of the program.

  Example: Disconnect record access streams and close all the files at one time, probably at the end of the program.

  NOTE

  Most high-level languages automatically perform CONNECT and DISCONNECT operations during the execution of file open and close statements.

- Record operations

  The primary overlay or (on RSX-11M-PLUS systems) paging burden of your task comes from record operations. However, the nature of processing often dictates the placement of record operations in your program. Therefore, the type and sequence of these operations direct your optimization of the ODL files (see Section 8.1.1).

  Example: If your task uses GET operations to read records from a sequential file, and then uses PUT operations to write records to an indexed file, you could reduce the number of overlays required for those specific operations.

  Example: If your task uses GET operations to read records from an indexed file and UPDATE operations to modify the records, you should optimize those operations.

  Whenever possible, perform operations on only one type of file organization at a time.

8.2.2 Task Builder Considerations

The task builder contructs a task and ensures that the task's overlays, if any, work properly. To do this, the task builder must know the task's overlay structure if you use disk-resident overlays: you supply this information by means of an ODL file.

To reduce the time that the task builder needs to build your task, you can reduce the number of overlays in the task (see Section 8.1.1). Each overlay adds time to task building because it requires that a symbol table be built and then resolved.

NOTE

If you use memory-resident overlays (resident library), you reduce task builder overhead needed to process overlay segments.

You can also reduce task building time by not requesting a map. If you really need a map for debugging, specify a short one (the default).

Note that the use of overlaid I- and D-space segments can increase task building time.

8.3 VIRTUAL-TO-LOGICAL-BLOCK MAPPING

When RMS-11 issues a data transfer request, it specifies a starting virtual block number (VBN) and the size of the request in bytes to the operating system. The system maps the VBN onto a logical block number (LBN) that it must use to find the block on disk. To do this, the system uses a set of retrieval pointers, called a window, to the file. The operating system creates a window in its part of memory by reading the first set of pointers from disk when a task opens a file. These pointers specify blocks on disk, and from the structure and content of the pointers for a file, the system equates virtual blocks to logical blocks.

8.3.1 Retrieval Pointers on Disk

The file directory contains the retrieval pointers for a file. The representation depends on your operating system.

The file control processor stores retrieval pointers in a file header, using enough file headers to cover the file. A file header can contain up to 102 pointers. Each pointer consists of:

- The number of blocks the pointer maps
- The LBN where the group of blocks starts

The largest group of blocks that can be covered by one pointer is 256 blocks. Therefore, one file header can map a maximum of 26,112 logical blocks.

8.3.2 Retrieval Pointers in Memory

The operating system keeps one window in memory for each file. If that window does not contain the retrieval pointer that covers the virtual block requested by RMS-11, the system must bring more pointers into memory in a process called window turning.

Window turning normally requires an I/O operation. The operating system builds file control block (FCBs) in memory when a task opens a file. An FCB contains information about one file header, including the range of virtual blocks covered by the header's retrieval pointers. Whenever the system has to turn a window, it consults the FCBs for the file to determine which file header contains the appropriate retrieval pointer. The file control processor then reads that block from disk, requiring only one I/O operation (unless the software needs one or possibly two overlays).

Example: An evaluation of one application revealed that window turning during record operations accounted for nearly 30 percent of the I/O operations.

8.3.3 Optimizing Window Turning

When you reduce window turning, you improve performance.

You can reduce the I/O operations associated with window turning by increasing a window size, maximizing contiguity, using areas in indexed files, or increasing the size of the F11ACP.

Increase Window Size: Use one of the following methods to increase the number of retrieval pointers the system keeps in memory for the file:

- Initialize the disk volume that will contain the RMS-11 file with a window size greater than the default of seven pointers per window. See your system documentation for initialization procedures.
- Mount the volume containing the RMS-11 file using the /WIN switch to specify a window size greater than the volume default. See your system documentation for volume mounting procedures.
- Use a MACRO-11 subroutine that sets the RTV field in the FAB for the file. See the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide.

In each of these methods, you can specify a number of window pointers, as follows:

- If you specify a -1, the system tries to make the window large enough to map the entire file, using up to 81 retrieval pointers.
- If you specify a positive number of pointers, the system uses that number, up to a maximum of 127 pointers in each window.

The initialization and mount methods apply to all files on a disk. These methods cause the system to use more executive memory than when you set window size for an individual file in a program.

Maximize Contiguity: To maximize contiguity, you should make the file contiguous or, if that is not possible, reduce the number of extents in the file, making each extent as large as possible.

One retrieval pointer in memory can map up to 65,536 logical blocks. When the file control processor reads a pointer from the header, the software determines whether the extent mapped by the pointer is logically contiguous with the extent covered by the preceding pointer in the window. If it is, the file control processor adds the extent size to the size field of the pointer in the window, then it reads the next pointer. If the two extents are not contiguous, it adds the new pointer to the window. This compaction extends across file headers.

In this way, any file can be mapped with a default window if the file is sufficiently contiguous.

Use Areas in Indexed Files: Areas localize successive block requests and reduce window turning.

Increase the Size of the F11ACP: The large version of the F11ACP does not require overlays of its own routines to perform window turning, whereas the small version does.

8.4 OTHER OPTIMIZATIONS

You can improve the environment in which your RMS-11 task runs by:

- Allocating more resources to the task
- Improving disk usage

8.4.1 Allocating More Resources to the Task

You can improve the performance of a task by giving it more of the system to use, more CPU time, more memory, and so on. You take those resources away from other jobs, unless the system is not used to capacity.

The techniques for allocating system resources vary by operating system. Each of the following techniques affects system throughput by changing the number of I/O operations your task requires to complete its work.

- Priorities
- Checkpointing
- Round-robin scheduling
- Swapping

8.4.2 Disk Usage

You should consider the devices that store your data and task images when you are optimizing the performance of an application. Efforts at improving disk usage often result in significant increases in performance.

- Use the fastest disk drives available because the physical I/O operation causes the most significant portion of I/O time.

- Minimize I/O request overhead:

  - Reduce I/O request queues, using private packs if necessary.

  - Use a disk that has exclusive use of its disk controller. If other disks must share the controller, a disk driver that supports "overlapped seeks" is desirable.

- If your system has multiple disk drives which are not heavily used by other people, spread an application's files, including disk-resident overlaid tasks, across the devices. Thus, while a job runs, one disk device does not access more than one file. You can also put data files on a disk device other than the one containing a disk-resident overlaid task image.

  If you are using the resident library, and not overlays, you do not consider the task file, unless your code or your language run-time facilities are overlaid.

- Combine free blocks on a disk into one contiguous group using the DSC utility. By eliminating fragmentation, you are increasing the chances that file extents are contiguous, even if they are not requested that way. The more contiguous the file, the fewer the disk head moves required to access it.

APPENDIX A

FILE SPECIFICATION PARSING

A.1 STANDARD FILE SPECIFICATION SYNTAX

A full file specification consists of the following elements, in the order listed:

device
directory
name
type
version

A.1.1 Device

The device element of the file specification names the device on which the file resides. For unit-record devices, such as terminals and lineprinters, this is the **only** significant element in the file specification.

The device specification consists of 2 alphabetic characters specifying the device name, followed by 0 to 3 octal numeric characters specifying the device unit number, followed by a colon (:). If you use lowercase characters to specify the device name, RMS-11 will convert them to uppercase before passing them to the operating system. The device unit number must not exceed 377 octal; if no unit number is given, RMS-11 will specify unit number 0.

Note that RSX-11M/M-PLUS systems allow definition of logical device names that conform to the above description. RMS-11 processes such logical device names as well as physical device names.

For example:

db2: and DB02: Are equivalent

SY: and sy00: Are equivalent

A.1.2 Directory

The directory element of the file specification names the directory through which the file can be found on the device. For ANSI magnetic tape files, this element is not significant (see Section A.2).

The directory specification can take either of the following forms:

```
[group,member]
```

or

```
<group,member>
```

Note that the delimiting characters ([] or <>) and the comma (,) must appear as shown. The group and member subelements each consist of a 1-to-3 digit octal number in the range of 0 to 377 octal. In situations where wildcarding is permitted, you can substitute a single asterisk (*) character for the group and/or member subelement to indicate that all such subelements are acceptable.

You can explicitly request the current default directory by specifying [] or <> as the directory specification.

For example:

[27,36] and <027,036>	Are equivalent
[27,*]	Indicates all members in group 27
[]	Indicates the current default directory

For compatibility with other systems, RMS-11 access methods can process directory specifications of the "named" directory format. However, this format does not conform to RSX-11M/M-PLUS file specification conventions and, in general, named directories cannot be processed by RSX-11M/M-PLUS software.

A.1.3 Name

The name element of the file specification provides the name by which the file is known in the directory. The name specification is a 0-to-9 character alphanumeric string. RMS-11 will convert lowercase alphabetic characters to uppercase before passing them to the operating system. In situations where wildcarding is permitted, you can substitute asterisk (*) and percent (%) characters in this string: the asterisk character matches any string (including the null string), and the percent character matches any single character.

For example:

MyFile	Will be interpreted as MYFILE
*	Will match all names
My*le	Will match all names beginning with MY and ending in LE
my	Will match all names containing MY
%%my*	Will match all names containing MY as the third and fourth characters
	Will be interpreted as the null name of 0 length

A.1.4 Type

The type element of the file specification is the type by which the file is known in the directory. The type specification consists of a period (.) followed by a 0-to-3 character alphanumeric string. RMS-11 will convert lowercase alphabetic characters to uppercase before passing them to the operating system. In situations where wildcarding is permitted, you can substitute asterisk (*) and percent (%) characters in this string. The asterisk character matches any string (including the null string) and the percent character matches any single character.

For example:

.dat Will be interpreted as .DAT

.* Will be interpreted as all types

.da* Will be interpreted all types beginning with DA

.%* Will be interpreted all types except the null type

. Will be interpreted as the null type

A.1.5 Version

The version element of the file specification provides the version number by which the file is known in the directory. The version specification consists of a semicolon (;) followed by a 0-to-5 digit octal number in the range of 0 to 77777 octal. In situations where wildcarding is permitted, you can substitute a single asterisk (*) character for the octal number to indicate that all versions are acceptable. In situations where you are specifying a file that already exists, you can substitute the two characters "-1" for the octal number to specify the lowest-numbered version of the file that is known to the directory.

You can specify a version number of 0 or the null version to indicate either of the following:

1. The highest-numbered version of the file that is known to the directory, when the file already exists

2. A version number one greater than the highest-numbered version of the file (if any) known to the directory, when you are creating a new directory entry (with the CREATE or ENTER operation, or with the RENAME operation, using the new file name)

For example:

;5 and ;0005 Are equivalent

;* Indicates all versions

;-1 Indicates the lowest-numbered version

; Indicates the null version; this is equivalent to ;0

For compatibility with other systems, RMS-11 access methods can process version specifications beginning with a period (.) instead of a semicolon (;) when the presence of a type specification eliminates ambiguity. However, this format does not conform to RSX-11M/M-PLUS

file specification conventions and, in general, version specifications of this form cannot be processed by RSX-11M/M-PLUS software.

NOTE

When performing ERASE, RENAME, or REMOVE operations within a wildcard loop whose file specification contains ;0 (or the null version) or ;-1 as the version **and** contains wildcards in the name or type, the behavior of the loop after the first such operation in any directory will depend upon the ordering of the versions in that directory. This is because entries in the directory are deleted during the loop's operation, while the determination of the highest- or lowest-numbered version of a given file must be made during each pass through the loop.

In addition, the addition of entries to a directory during a wildcard loop may result in encountering these new entries during subsequent iterations of the loop.

Examples of a full file specification follow:

```
LB:[1,1]RMSLIB.OLB;1
```

```
LB:[1,*]RMS*.*;0
```

The highest-numbered version of each entry on logical device LB: in group 1 with a name that begins with "RMS"

A.2 ANSI MAGNETIC TAPE FILE SPECIFICATION SYNTAX

The file specification format specific to magnetic tapes consists of the following elements, in the order listed:

device
directory
quoted-string
version

A.2.1 Device

The device element is the same as described in Section A.1.1. The device must be a magnetic tape device.

A.2.2 Directory

The directory element is the same as described in Section A.1.1. This element has no meaning for ANSI magnetic tape files, and will be ignored if present.

A.2.3 Quoted String

RMS-11 treats a quoted string as a unit representing both the name and type elements of a standard file specification. This mechanism is used to allow expression of tape file names up to 17 characters in length which include the full set of ANSI "a" characters (some of which would otherwise be ignored or treated as element delimiters in a standard file specification).

You specify an ANSI name by enclosing the name in quotation characters ("name"). If the name itself contains full quotation characters ("), you must also precede each such character with an additional full quotation (") character. RMS-11 will convert any lowercase alphabetic characters to uppercase, strip the full-quotation characters that you have added, and pass the result to the operating system without further modification (including ANSI "a" characters such as SPACE).

For example:

"My File" Will be interpreted as MY FILE

""""Don't Panic""" Will be interpreted as "DON'T PANIC"

A.2.4 Version

The version element of a magnetic tape file specification is as described in Section A.1.1. A version specification of ;0, ;-1, or the null version will be interpreted as any version for magnetic tape files. An example of an ANSI magnetic tape file specification follows:

MM1:"John's file" Specifies any version of JOHN'S FILE on device MM1:

The standard file specification format described in Section A.1.1 can also be used with magnetic tapes; this is usually desirable to promote file transport to nontape devices and file accessibility by the widest possible range of software. See Appendix G of the IAS/RSX-11 I/O Operations Reference Manual for additional information concerning the use of names in ANSI magnetic tape files.

A.3 GENERATION OF A FULL FILE SPECIFICATION

When you specify the target file for an RMS-11 operation, RMS-11 generates a full file specification in the following manner:

1. RMS-11 parses the file name string to determine which elements are present. You need not provide a full file specification in the file name string; however, any elements present must be syntactically correct and in the proper order. RMS-11 ignores any NULL, SPACE, or TAB characters that may be present in the string unless they occur within an ANSI magnetic tape quoted-string name.

2. RMS-11 parses the default name string to determine which elements are present. You need not provide a full file specification in the default name string; however, any elements present must be syntactically correct and in the proper order. RMS-11 ignores any NULL, SPACE, or TAB characters that may be present in the string unless they occur within an ANSI magnetic tape quoted-string name.

3. If the file name string does not provide a full file specification, RMS-11 obtains missing elements from the default name string; if any elements are absent in the result of this merge, RMS-11 provides default values for them as follows:

   - device -- defaulted to the device to which the specified logical channel is currently assigned; if the specified logical channel is not assigned to any device, defaulted to SY:

   - directory -- defaulted to the current directory

   - name, type, version -- defaulted to null

4. If you have asked RMS-11 to use information from the NAM block, RMS-11 uses this information to override elements in the full file specification obtained above. This mechanism is described in Chapter 3 of the RSX-11M/M-PLUS RMS-11 Macro Programmer's Guide.

APPENDIX B

REMOTE FILE AND RECORD ACCESS VIA DECNET

If suitable DECnet facilities exist on your system and on the target system, RMS-11 will allow file and record access to files residing on other network nodes. Note, however, that these nodes must include an RMS-11-based file access listener (FAL); see Section B.2.

For most purposes, remote access is indistinguishable from local access, although performance may not be equivalent. The following general limitations apply:

- RMS-11 generally does not support remote functions (for example, to a VAX-11 node) that are not supported locally.
- Certain RMS-11 functions (wildcard support; the PARSE, SEARCH, ENTER, REMOVE, and RENAME operations; and transmission of device, directory, and file identifiers) are not supported by the data access protocol (DAP), and thus cannot be executed remotely.
- Certain FALs do not support the full set of RMS-11 functions expressible via DAP, and thus further limit remote access. For example, the current RSTS/E RMS-11 FAL does not support record access to indexed files.
- High-level languages may not allow expression of the file specification required to establish contact with a remote node.

To operate upon a remote file, you must include the RMS-11 remote access code when you build your task, and your program must include a node specification for the remote file. To include the remote access code, you must link your program with the RMSDAP modules either by using the disk-resident overlaid version (see Section 8.1.1) or by referencing the DAPRES resident library (see Section 8.1.2).

NOTE

RMS-11 uses the file access block (FAB) logical channel number as the link-id for remote access. Users performing remote access external to RMS-11 should be careful not to use the same link-ids.

B.1 REMOTE NODE SPECIFICATION

You must include a remote note specification at the beginning of the file name string or the default name string you provide to an OPEN, CREATE, or ERASE operation. In addition, your file name string and default name string must conform to the DIGITAL Command Language (DCL) file specification syntax rules, and the file specification that results from merging the file name string and default name string must conform to the file specification conventions of the target node as well.

In general, a full remote file specification consists of the following elements, in the order (and with the delimiters) given:

```
node::device:[directory]name.type;version
```

Elements beyond the node element must conform to the conventions of the target node, as well as to DCL syntax. If the file name string does not provide all six elements, RMS-11 obtains missing elements from the default name string. Elements that are still missing after this merge are defaulted according to the conventions of the target system.

An alternative remote file specification format is:

```
node::"quotedstring"
```

where **quotedstring** is any file specification that conforms to target system conventions. For example, this provides a means of passing certain RSTS/E logical names ($, %, and so on) that do not conform to DCL conventions.

If the quoted string itself contains any quotation mark (") characters, you must insert an additional quotation mark character before each; these additional quotation mark characters will be stripped from the string when the string is passed to the target system. Any elements not present in the quoted string will be defaulted according to target system conventions.

RMS-11 treats specifications of this format as complete, indivisible specifications. If one occurs in the file name string, no elements from the default name string will be used; if one occurs in the default name string, it will be ignored unless the file name string is empty.

The node element takes the form:

```
node"user password"::
```

where **node** is the destination node name and **"user password"** is an optional access control string containing login information (user identifier and password, separated by a space character) that meets target system login conventions.

If the login information is provided, the device and directory defaults and access privileges of the remote account are acquired. Otherwise, the device and directory defaults and access privileges of the default DECnet account on the target system are acquired.

B.2 REMOTE ACCESS ENVIRONMENTS

RMS-11-based FALs are currently available on VAX/VMS, RSTS/E, and RSX-11M/M-PLUS systems.

The version of DAP that you use must be at least Version 5.6 or greater. This means that you must have at least DECnet/E Version 2.0, DECnet for RSX-11M Version 3.1, DECnet for RSX-11M-PLUS Version 1.1, or DECnet/VAX Version 2.0.

B.3 REMOTE ACCESS POOL CONSIDERATIONS

Remote block access, unlike local block access, requires an internal I/O buffer for record operations, as well as for the initial OPEN or CREATE operation. This buffer is reserved to the file while the file is open, and must be 548 bytes in size.

Similarly, for sequential files with a specified maximum record size (or actual largest record) greater than 476 bytes, an internal I/O buffer 36 bytes larger than this maximum is required while the file is open. For record-accessed relative and indexed files, an internal I/O buffer equal to the bucket size is required while the file is open. Other pool requirements are equal to or less than those for local access.

INDEX

Access, 1-7
  block, 1-20, 3-6
  random, 1-7, 1-9
    to indexed files, 5-7, 5-9
  See also Access modes
  See also Shared access
  sequential, 1-7 to 1-8
    to indexed files, 5-16
Access control, 1-9
Access declarations, 2-6
  indexed files, 7-1
  read-only, 2-6
  read/write, 2-6
  relative files, 4-6
  sequential files, 3-6
Access modes
  block, 1-20, 3-6
  FIND operation
    indexed files, 7-3
    relative files, 4-8
    sequential files, 3-7
  GET operation
    indexed files, 7-5
    relative files, 4-10
    sequential files, 3-9
  PUT operation
    indexed files, 7-6
    relative files, 4-11
    sequential files, 3-10
  random
    by key, 1-12
    by RFA, 1-12
    by VBN, 1-20
  record, 1-12
  See also Random access
  See also Sequential access
  sequential
    blocks, 1-20
    records, 1-12
Access sharing
  See Shared access
Access streams, 1-15
  multiple, 2-17
    and shared access, 2-9 to 2-10
    indexed files, 7-10
    relative files, 4-15
    sequential files, 3-15
Active page registers
  See APRs
Address
  record, 1-9
  See also RFA
Allocation, 1-19
  indexed files, 6-22
    DEQ, 6-26
    initial, 6-22
  relative files, 4-3
    DEQ, 4-4
    initial, 4-3
  sequential files, 3-4
    DEQ, 3-4
    initial, 3-4
Alternate indexes
  See Indexes
Alternate keys, 1-7, 1-11
Applications, 1-13
  optimization, 2-11
  See also Designing applications
APRs, 8-3
  and memory-resident overlays, 8-6
  supervisor mode, 8-7
Areas, 6-10, 8-13
  contiguity, 6-13
  definition, 5-2
  multiarea indexed files, 6-12
  single-area indexed files, 6-11
Asynchronous operations, 8-2, 8-4, 8-7 to 8-8
  I/O techniques, 2-17
  indexed files, 7-8
  relative files, 4-14
  sequential files, 3-14
Attributes, 1-2, 1-18
  block spanning, 1-19
  bucket size, 1-19
  contiguity, 1-19
  file allocation, 1-19
  file organization, 1-18
  file specification, 1-18
  keys, 1-19
  medium, 1-18
  MRN, 1-19
  placement control, 1-19
  protection, 1-18
  record format, 1-18
  record size, 1-18
  record-output handling, 1-19

Backing up files, 1-10, 1-12
Binary keys
  2-byte unsigned, 6-5
  4-byte unsigned, 6-5
Block access, 1-20, 3-6
  remote, B-3
  see also Shared access
Block spanning, 1-19, 3-1
  variable length records, 2-16
Blocks, 1-16
  See also Control blocks
  spanning, 1-16, 3-1
Bucket fill size
  populating indexed files, 6-28
Bucket format
  indexed files, 5-2
Bucket locking

and shared access, 2-8
FIND operation, 2-8
GET operation, 2-8
Bucket size
calculation
alternate indexes, 6-19
primary index, 6-16
indexed files, 6-15
prologue, 5-2
relative files, 4-2
Bucket splitting, 5-11
RRV, 5-12
Buckets
continuation, 5-5
high-key value, 5-5 to 5-6
I/O units, 1-16
index, 5-6
relative files, 4-1
size, 1-19
Buffers
cache, 4-15
I/O, 1-14
size, 2-5
multiple, 2-18, 3-14, 4-15
user, 1-14
Building tasks, 1-13
See also Task builder

Cache, 4-15
indexed files, 7-9
Cells
relative files, 4-1
sequential files, 3-2
Changeable keys, 6-9
CLOSE operation, 1-18
Clustered libraries, 8-7 to 8-9
Compatibility
file specification, A-2, A-4
Compilers
See High-level languages
Compressing deleted records, 5-10, 5-16, 6-7, 6-9
CONNECT operation, 1-15
indexed files, 7-2
relative files, 4-7
sequential files, 3-7
Context, 1-9, 1-15
CONNECT operation, 3-7, 4-7, 7-2
current record, 1-9, 1-15
DELETE operation, 4-7, 7-3
DISCONNECT operation, 3-7, 4-8
FIND operation, 3-8, 4-8, 7-3
FLUSH operation, 3-9, 4-10, 7-5
GET operation, 3-10, 4-11, 7-5
next record, 1-9, 1-15
PUT operation, 3-11, 4-11, 7-6
REWIND operation, 3-11, 4-12, 7-6
TRUNCATE operation, 3-12
UPDATE operation, 3-13, 4-12, 7-6
Contiguity, 1-19, 8-12
indexed files, 6-13
relative files, 4-4
sequential files, 3-5
Continuation buckets, 5-5
Control blocks, 1-14, 2-2
Converting files, 1-10, 1-12
CREATE operation, 1-18
and shared access, 2-6
file versions, A-3
remote access, B-2 to B-3
Creating files, 1-10, 1-12
Current record
See Context
Cylinder
See Placement control

DAP (data access protocol)
See DECnet
DAPRES
remote access code, B-1
remote access support, 8-7
Data access protocol (DAP)
See DECnet
Data storage
and file organization, 2-5
Data types
keys, 6-3
2-byte unsigned binary, 6-5
2-byte-signed integer, 6-4
4-byte signed integer, 6-4
4-byte unsigned binary, 6-5
and segmenting, 6-7
packed decimal, 6-6
string, 6-3
DCL, B-2
DECnet, 8-6 to 8-8, B-1
and CREATE operations, B-2 to B-3
and ENTER operations, B-1
and ERASE operations, B-2
and file organization, B-3
and OPEN operations, B-2 to B-3
and PARSE operations, B-1
and REMOVE operations, B-1
and RENAME operations, B-1
and SEARCH operations, B-1
block access, B-3
DAPRES resident library, B-1
disk-resident overlaid code, B-1
FALs
on different systems, B-3
file specification, B-2
defaults, B-2
node, B-2
quoted string, B-2
I/O buffers, B-3
indexed files, B-3
relative files, B-3
sequential files, B-3
limitations on functions, B-1

Default extension quantity
  See DEQ
Defaults, 2-2
  compilers, 2-2
  DEQ, 4-4
  file specification, A-6, B-2
  operating system, 2-2
    DEQ, 3-5
  RMS-11, 2-2
Deferred write, 2-17, 6-27
  and mass insertion, 6-29
  indexed files, 7-8
  relative files, 4-14
  sequential files, 3-14
DELETE operation, 1-15
  and key position, 6-7
  and shared access, 2-6
  compressing records, 5-10, 5-16, 6-7, 6-9
  deferred write, 7-9
  duplicate keys, 6-9
  indexed files, 5-15, 7-3, 7-9
  optimizing, 6-7
  relative files, 4-7
Depth
  indexed files, 5-5
DEQ
  default
    indexed files, 6-26
    relative files, 4-4
    sequential files, 3-5
  indexed files, 6-26
  relative files, 4-4
  sequential files, 3-4
Designing applications, 2-1
  considerations, 2-3
    ease of design, 2-10
    shared access, 2-5
    space, 2-4, 8-1
    speed, 2-3, 8-1
  when to design, 2-2
Designing files, 1-10, 1-12
  indexed, 6-1
  relative, 4-2
  See also Indexed file organization
  See also Relative file organization
  See also Sequential file organization
  sequential, 3-3
Devices
  and record formats, 2-15
  disk, 1-12
  file specification, A-1, A-4
  magnetic tape, 1-12
  See also Medium
DIGITAL Command Language
  See DCL
Directory
  default, A-6
  file specification, A-1, A-4
Directory operations, 1-17
  See also Operations
DISCONNECT operation, 1-15
  indexed files, 7-3
  relative files, 4-8
  sequential files, 3-7
Disk
  See Medium
Disk-resident overlays, 1-13, 8-3, 8-8 to 8-9
  remote access code, B-1
DISPLAY operation, 1-18
Displaying files, 1-10, 1-12
Duplicate keys, 6-8

End-of-block indicators
  sequential files, 3-2
End-of-file
  sequential files, 3-2
ENTER operation, 1-17
  file versions, A-3
ENTER operations
  remote access, B-1
ER$ACT
  asynchronous operations, 3-14, 4-14, 7-8
ER$CUR
  DELETE operation, 4-7
  TRUNCATE operation, 3-12
  UPDATE operation, 3-12, 4-12, 7-6
ER$DEL
  FIND operation, 4-8, 7-3
  GET operation, 4-11, 7-5
ER$EOF
  FIND operation, 3-8, 4-8, 7-3
  GET operation, 3-10, 4-10, 7-5
  PUT operation, 3-11
ER$IOP
  UPDATE operation, 3-12
ER$KEY
  FIND operation, 3-8
  GET operation, 3-10
ER$LIB
  and resident libraries, 8-8
ER$MRN
  FIND operation, 4-8
  GET operation, 4-10
ER$NEF
  PUT operation, 3-10
ER$RAC
  PUT operation, 3-10
ER$REX
  PUT operation, 4-11
ER$RFA
  FIND operation, 3-8, 7-3
  GET operation, 3-10, 7-5
ER$RFM
  UPDATE operation, 3-12
ER$RLK
  and shared access, 2-10
ER$RNF
  FIND operation, 4-8, 7-3
  GET operation, 4-10 to 4-11,

7-5
ER$RSZ
UPDATE operation, 3-12
ERASE operation, 1-18
remote access, B-2
wildcard loops, A-4
Error codes, 3-7, 4-7, 7-2
ER$ACT, 3-14, 4-14, 7-8
ER$CUR, 3-12, 4-7, 4-12, 7-6
ER$DEL, 4-8, 4-11, 7-3, 7-5
ER$EOF, 3-8, 3-10 to 3-11, 4-8, 4-10, 7-3, 7-5
ER$IOP, 3-12
ER$KEY, 3-8, 3-10
ER$LIB, 8-8
ER$MRN, 4-8, 4-10
ER$NEF, 3-10
ER$RAC, 3-10
ER$REX, 4-11
ER$RFA, 3-8, 3-10, 7-3, 7-5
ER$RFM, 3-12
ER$RLK, 2-10
ER$RNF, 4-8, 4-10 to 4-11, 7-3, 7-5
ER$RSZ, 3-12
EXTEND operation, 1-18
and shared access, 2-6

F11ACP, 8-13
FALs, B-1
RSTS/E, B-1, B-3
RSX-11M/M-PLUS, B-3
VAX/VMS, B-3
FCBs, 8-12
FCS-11
sequential file compatibility, 3-2
File access listener
See FAL
File allocation
See Allocation
File control blocks
See FCBs
File control processor, 1-16
retrieval pointers, 8-11
window turning, 8-12
File name
default, A-6
file specification, A-2
File operations, 1-18
See also Operations
File organizations, 1-11, 1-18
and data storage, 2-5
and file design, 2-11
and optimizations, 8-10
and record formats, 1-20, 2-15
and standard ODL files, 8-5
indexed, 1-5, 1-11
relative, 1-4, 1-11
See also Indexed file organization
See also Relative file organization
See also Sequential file organization
selection, 2-11
sequential, 1-3, 1-11
File sharing
See Shared access
File specification, 1-18 to 1-19
default name string, A-5 to A-6
defaults, A-6
directory, A-6
name, A-6
node, B-2
type, A-6
version, A-6
file name string, A-5
magtape, A-4 to A-5
device, A-4
directory, A-4
quoted string, A-5
version, A-5
NAM block, A-6
node, B-2
quoted string, B-2
remote, B-2
standard, A-1
device, A-1
directory, A-1
name, A-2
type, A-3
version, A-3
wildcards, A-2 to A-3
File structure
See structure
File type, A-3
default, A-6
File version, A-3, A-5
default, A-6
Files, 1-2
attributes, 1-2
backing up, 1-10, 1-12
converting, 1-10, 1-12
creating, 1-10, 1-12
designing, 1-10, 1-12
displaying, 1-10, 1-12
loading, 1-10, 1-12
maintaining, 1-10, 1-12
processing, 1-16
restoring, 1-10, 1-12
See also Indexed file organization
See also Relative file organization
See also Sequential file organization
FIND operation, 1-15
and bucket locking, 2-8
and shared access, 2-6, 2-10
and stream records, 2-16
deferred write, 7-9
indexed files, 7-3
key access, 7-3
random by key, 5-13

RFA access, 7-3
sequential access, 5-17, 7-3
key access, 1-16
relative files, 4-8
key access, 4-8
RFA access, 4-8
sequential access, 4-8
RFA access, 1-16
sequential access, 1-16
sequential files, 3-7
key access, 3-7
RFA access, 3-7
sequential access, 3-7
Fixed-length record format, 1-11, 2-15
FLUSH operation, 1-15
indexed files, 7-5
relative files, 4-10
sequential files, 3-9
Formats
See Record formats
Four-byte signed integer keys, 6-4
Four-byte unsigned binary keys, 6-5
FREE operation, 1-15
and shared access, 2-10

GET operation, 1-15
and bucket locking, 2-8
and ODLs, 8-5
and shared access, 2-6, 2-10
and stream records, 2-16 to 2-17
deferred write, 7-9
indexed files, 7-5, 7-9
key access, 7-5
random by key, 5-13
RFA access, 7-5
sequential access, 5-17, 7-5
key access, 1-16
locate mode, 3-14
indexed files, 7-8
relative files, 4-13
move mode
indexed files, 7-7
relative files, 4-12
sequential files, 3-13
relative files, 4-10
key access, 4-10
RFA access, 4-10
sequential access, 4-10
RFA access, 1-16
sequential access, 1-16
sequential files, 3-9
key access, 3-9
RFA access, 3-9
sequential access, 3-9

High-key value
buckets, 5-5 to 5-6
High-level languages
access streams, 1-15
and asynchronous operations, 3-14
and file design, 1-19
and ODL files, 8-5
and remote access, B-1
and shared access, 2-7
asynchronous operations, 4-14
bucket fill size, 6-29
bucket size, 4-3, 6-21
defaults, 2-2
DEQ, 3-5, 4-4, 6-26
file and directory operations, 3-15, 4-15, 7-10
file operations, 8-10
initial allocation, 3-4, 4-4, 6-26
key characteristics, 6-8
keys, 6-7
locate mode, 3-14, 4-13
MRN, 4-5
placement control, 6-14
populating files, 6-26
restrictions, 1-13, 1-15, 2-1

I/O
and performance, 2-3
and record operations, 5-17
paging operations, 8-10
window turning, 8-11
I/O buffers, 1-14
application design, 2-5
remote access, B-3
indexed files, B-3
relative files, B-3
sequential files, B-3
I/O techniques, 2-17
asynchronous operations, 2-17
deferred write, 2-17
indexed files, 7-8
asynchronous operations, 7-8
deferred write, 7-8
multiple access streams, 7-10
multiple buffers, 7-9
sequential reads, 7-10
mass insertion, 2-18
MBC, 2-17
multiple access streams, 2-17
multiple buffers, 2-18
relative files, 4-14
asynchronous operations, 4-14
deferred write, 4-14
multiple access streams, 4-15
multiple buffers, 4-15
sequential files, 3-14
asynchronous operations, 3-14
deferred write, 3-14
MBC, 3-15
multiple access streams, 3-15

multiple buffers, 3-14
I/O units, 1-14
blocks, 1-16
buckets, 1-16
Incremental reorganization, 5-12
Index buckets, 5-6
Index records, 5-6
Indexed file organization, 1-5
access declarations, 7-1
allocation, 6-22
DEQ, 6-26
initial, calculation, 6-22
alternate keys, 1-11
and remote access, B-3
areas, 5-2, 6-10
multiarea files, 6-12
single-area files, 6-11
asynchronous operations, 7-8
bucket fill size, 6-28
bucket format, 5-2
bucket size, 6-15
calculation, alternate indexes, 6-19
calculation, primary index, 6-16
prologue, 5-2
bucket splitting, 5-11
compressing deleted records, 5-10, 5-16, 6-7, 6-9
CONNECT operation, 7-2
contiguity, 6-13
data records, 5-2
deferred write, 6-27, 7-8
DELETE operation, 5-15, 6-7, 7-3, 7-9
duplicate keys, 6-9
depth, 5-5
design, 5-1, 6-1
allocation, 6-22
areas, 6-10
bucket size, 6-15
keys, 6-2
placement control, 6-13
populating files, 6-26
record format, 6-1
record size, 6-1
directory operations, 7-10
DISCONNECT operation, 7-3
file operations, 7-10
FIND operation, 7-3, 7-9
random by key, 5-13
sequential access, 5-17
FLUSH operation, 7-5
GET operation, 7-5, 7-7 to 7-9
random by key, 5-13
sequential access, 5-17
I/O techniques, 7-8
incremental reorganization, 5-12
index buckets, 5-6
index levels, 5-4, 5-6
index records, 5-2, 5-6
indexes
data level, 5-5
keys, 1-5, 6-2
changeable, 6-9
characteristics, 6-8
data types, 6-3
duplicates, 6-8
null, 6-10
number of, 6-2
position, 6-7
size, 6-6
level 0
alternate indexes, 5-5
primary indexes, 5-5
locate mode, 7-7
mass insertion, 6-28 to 6-29
move mode, 7-7
multiple access streams, 7-10
multiple buffers, 7-9
placement control, 6-13
populating files, 6-26
primary key order, 6-27
random insertions, 6-28
primary keys, 1-11
prologue, 5-2
PUT operation, 6-7, 7-5, 7-7, 7-9
duplicate keys, 6-9
mass insertion, 7-9
random by key, 5-10
random access, 5-7, 6-12
record operations, 7-2
random access, 5-9
sequential access, 5-16
record transfer modes, 7-6
locate mode, 7-7
move mode, 7-7
REWIND operation, 7-6
root, 1-7, 5-5
search times, 5-8
See also Indexes
sequential reads, 7-10
shared access, 7-1
block access, 7-1
record access, 7-1
sharing declarations, 7-1
stream operations, 7-2
structure
conceptual, 5-4
physical, 5-2
UPDATE operation, 5-14, 7-6 to 7-9
changeable keys, 6-10
duplicate keys, 6-9
Indexes, 5-2
alternate
bucket size calculation, 6-19
level 0, 5-5
SIDRs, 5-5
depth, 5-5
levels, 5-4, 5-6
data, 5-5
primary

bucket size calculation, 6-16
level 0, 5-5
root, 5-5
Initial allocation
indexed files, 6-22
relative files, 4-3
sequential files, 3-4
Integer keys
2-byte signed, 6-4
4-byte signed, 6-4

Key access
to indexed files, 5-7, 5-9
to relative files, 4-1
to sequential files, 3-2
Keys, 1-5, 1-19, 6-2
alternate, 1-7, 1-11
characteristics, 6-8
changeable, 6-9
duplicates, 6-8
null, 6-10
data types, 6-3
2-byte signed integer, 6-4
2-byte unsigned binary, 6-5
4-byte signed integer, 6-4
4-byte unsigned binary, 6-5
packed decimal, 6-6
string, 6-3
number of, 6-2
position, 6-7
primary, 1-7, 1-11
segmented, 6-7
size, 6-6

LBN, 8-11
See also Placement control
Levels
indexed files, 5-4
data, 5-5
level 0, alternate indexes, 5-5
level 0, primary indexes, 5-5
Libraries
clustered, 8-7 to 8-9
object module, 8-1
resident, 8-1, 8-8
RMSRES, 8-6, 8-9
supervisor mode, 8-9
Loading files, 1-10, 1-12
Locate mode
indexed files, 7-7
relative files, 4-13
sequential files, 3-14
Logical block number
See LBN

MACRO-11, 1-13
and ODL files, 8-5
key segments, 6-7
placement control, 6-14
populating files, 6-27
Magtape
See Medium
Maintaining files, 1-10, 1-12
backing up files, 1-10, 1-12
converting files, 1-10, 1-12
designing and creating files, 1-10, 1-12
displaying files, 1-10, 1-12
loading files, 1-10, 1-12
restoring files, 1-10, 1-12
Mass insertion, 2-18, 6-28, 7-9
populating indexed files, 6-29
Match criteria
random access, 5-13
Maximum record number
See MRN
MBC, 2-17
sequential files, 3-15
Medium
and I/O time, 2-3
and record formats and file organizations, 1-20
and variable-length format, 2-15
disk, 1-18
file specification, A-1
retrieval pointers, 8-11
usage, 8-13
magtape, 1-18
file specification, A-4 to A-5
placement control, 6-13
sequential files, 3-3
Memory-resident overlays, 1-13, 8-3, 8-6
Modes
See Access modes
See Record transfer modes
Move mode
indexed files, 7-7
relative files, 4-12
sequential files, 3-13
MRN, 1-19
relative files, 4-4 to 4-5
Multiblock count
See MBC
Multiple access streams
and shared access, 2-9 to 2-10
I/O techniques, 2-17
indexed files, 7-10
relative files, 4-15
sequential files, 3-15
Multiple buffers, 2-18
indexed files, 7-9
relative files, 4-15
sequential files, 3-14

NAM block
file specification, A-6
Name
file, A-2
default, A-6
Next record

  See Context
No sharing
  sharing declaration, 2-7
Node
  remote file specification, B-2
Nonoverlaid routines, 1-13,
  8-2, 8-8 to 8-9
Null keys, 6-10

Object code
  assembling, 1-13, 8-1
  compiling, 1-13, 8-1
Object module libraries, 8-1
ODL, 8-3
ODL files
  DAPRLX, 8-7
  prototype, 8-3
  RMSRLX, 8-7
  standard, 8-3, 8-5
    DAP11X, 8-6
    RMS11S, 8-5
    RMS11X, 8-5
    RMS12X, 8-6
OPEN operation, 1-18
  remote access, B-2 to B-3
Operating systems, 1-13
  allocating system resources,
    8-13
  block locking, 3-6
  compatibility, 3-4
  defaults, 2-2
    DEQ, 3-5
  FALs, B-3
  protection codes
    and shared access, 2-5
  remote access
    RSTS/E, B-3
    RSX-11M/M-PLUS, B-3
    VAX/VMS, B-1, B-3
Operations
  asynchronous, 2-17, 3-14,
    4-14, 7-8, 8-2, 8-4,
    8-7 to 8-8
  CLOSE, 1-18
  CONNECT, 1-15
    indexed files, 7-2
    relative files, 4-7
    sequential files, 3-7
  CREATE, 1-18
    and shared access, 2-6
    file versions, A-3
    remote access, B-2 to B-3
  DELETE, 1-15
    and key position, 6-7
    and shared access, 2-6
    deferred write, 7-9
    duplicate keys, 6-9
    indexed files, 5-15, 7-3
    relative files, 4-7
  directory, 1-17
    indexed files, 7-10
    relative files, 4-15
    sequential files, 3-15
DISCONNECT, 1-15
  indexed files, 7-3
  relative files, 4-8
  sequential files, 3-7
DISPLAY, 1-18
ENTER, 1-17
  file versions, A-3
  remote access, B-1
ERASE, 1-18
  remote access, B-2
  wildcard operations, A-4
EXTEND, 1-18
  and shared access, 2-6
file, 1-18
  indexed files, 7-10
  optimizations, 8-10
  relative files, 4-15
  sequential files, 3-15
FIND, 1-15
  and bucket locking, 2-8
  and shared access, 2-6, 2-10
  and stream records, 2-16
  deferred write, 7-9
  indexed files, 5-13, 5-17,
    7-3
  key access, 1-16
  relative files, 4-8
  RFA access, 1-16
  sequential access, 1-16
  sequential files, 3-7
FLUSH, 1-15
  indexed files, 7-5
  relative files, 4-10
  sequential files, 3-9
FREE, 1-15
  and shared access, 2-10
GET, 1-15
  and bucket locking, 2-8
  and ODLs, 8-5
  and shared access, 2-6, 2-10
  and stream records,
    2-16 to 2-17
  deferred write, 7-9
  indexed files, 5-13, 5-17,
    7-5
  key access, 1-16
  locate mode, 3-14, 4-13, 7-8
  move mode, 3-13, 4-12, 7-7
  relative files, 4-10
  RFA access, 1-16
  sequential access, 1-16
  sequential files, 3-9
OPEN, 1-18
  remote access, B-2 to B-3
optimizations, 8-10
PARSE, 1-17
  remote access, B-1
PUT, 1-15
  and key position, 6-7
  and ODLs, 8-5
  and shared access, 2-6
  and stream records, 2-17
  deferred write, 7-9

duplicate keys, 6-9
indexed files, 5-10, 5-17, 7-5, 7-9
initial allocation, 4-4
key access, 1-16
locate mode, 3-14
mass insertion, 7-9
move mode, 3-13, 4-12, 7-7
relative files, 4-11
sequential access, 1-16
sequential files, 3-10
record, 1-10, 1-15
and I/O costs, 5-17
and indexed files, 5-9, 5-16
indexed files, 7-2
optimizations, 8-10
relative files, 4-7
sequential files, 3-7
REMOVE, 1-17
remote access, B-1
wildcard operations, A-4
RENAME, 1-17
file versions, A-3
remote access, B-1
wildcard operations, A-4
REWIND, 1-15
indexed files, 7-6
relative files, 4-12
sequential files, 3-11
SEARCH, 1-17
remote access, B-1
stream, 1-15
indexed files, 7-2
relative files, 4-7
sequential files, 3-7
synchronous, 8-2
TRUNCATE, 1-15
and shared access, 2-6
sequential files, 3-12
UPDATE, 1-15
and ODLs, 8-5
and record size, 2-16
and shared access, 2-6, 2-10
and stream records, 2-17
changeable keys, 6-10
deferred write, 7-9
duplicate keys, 6-9
indexed files, 5-14, 7-6
move mode, 3-13 to 3-14, 4-12 to 4-13, 7-7 to 7-8
relative files, 4-12
sequential files, 3-12
WAIT, 1-15
WRITE
and shared access, 2-6
Optimizations
allocating system resources, 8-13
application design, 2-11
DELETE operation, 6-7
disk usage, 8-13
operations, 8-10
overlays, 8-1
program development, 8-9
PUT operation, 6-7
task building, 8-10
virtual-to-logical block mapping, 8-11
window turning, 8-12
Organizations
See File organizations
Overlay description language
See ODL
Overlays, 8-2
and disk usage, 8-14
disk-resident, 1-13, 8-3, 8-8 to 8-9
memory-resident, 1-13, 8-3, 8-6

Packed decimal keys, 6-6
Paging, 8-10
PARSE operation, 1-17
PARSE operations
remote access, B-1
Performance
See Speed
Placement control, 1-17, 1-19
calculating starting LBN, 6-14
cylinder, 6-13 to 6-14
indexed files, 6-13
sector, 6-14
track, 6-13 to 6-14
Populating indexed files, 6-26
primary key order, 6-27
random insertions, 6-28
bucket fill size, 6-28
mass insertion, 6-29
Primary indexes
See Indexes
Primary keys, 1-7, 1-11
Processing blocks, 1-20
Processing files, 1-16
indexed files, 7-10
relative files, 4-15
sequential files, 3-15
Processing records, 1-10, 1-15
indexed files, 7-2
relative files, 4-7
sequential files, 3-7
Program development
optimizing, 8-9
Prologue, 1-19
indexed files, 5-2
bucket size, 5-2
relative files, 4-1
Protection, 1-18
Prototype ODL files, 8-3
PUT operation, 1-15
and key position, 6-7
and ODLs, 8-5
and shared access, 2-6
and stream records, 2-17
deferred write, 7-9
duplicate keys, 6-9
indexed files, 7-5, 7-9

key access, 5-10, 7-6
mass insertion, 7-9
sequential access, 5-17, 7-6
initial allocation, 4-4
key access, 1-16
locate mode, 3-14
move mode
indexed files, 7-7
relative files, 4-12
sequential files, 3-13
optimizing, 6-7
relative files, 4-11
key access, 4-11
sequential access, 4-11
sequential access, 1-16
sequential files, 3-10
key access, 3-10
sequential access, 3-10

Quoted string
file specification, A-5
remote file specification, B-2

Random access, 1-7, 1-9
by key, 1-12
FIND, 1-16
GET, 1-16
PUT, 1-16
by RFA, 1-12
FIND, 1-16
GET, 1-16
by VBN, 1-20
match criteria, 5-13
to indexed files, 5-7, 5-9, 6-12
to relative files, 4-1
to sequential files, 3-2
Read-only
access declaration, 2-6
sharing declaration, 2-7
Read/write
access declaration, 2-6
sharing declaration, 2-7
Record access, 1-12
See also Shared access
Record access modes
See Access modes
Record file address
See RFA
Record formats, 1-11, 1-18
and file organizations, 1-20, 2-15
fixed length, 1-11, 2-15
indexed files, 6-1
relative files, 4-2
sequential files, 3-3
stream, 1-11, 2-16
undefined, 1-11, 2-17
variable length, 1-11, 2-15
and medium, 2-15
VFC, 1-11, 2-16
Record operations, 1-15
See also Operations
Record reference vector
See RRV
Record size, 1-18
indexed files, 6-1
relative files, 4-2
sequential files, 3-2
UPDATE operation, 2-16
Record transfer modes
indexed files, 7-6
relative files, 4-12
sequential files, 3-13
Record-output handling, 1-19
Records, 1-1
data, 5-2
index, 5-2, 5-6
processing, 1-15
See also Record formats
Relative file organization, 1-4, 1-11
access declarations, 4-6
allocation
DEQ, 4-4
initial, 4-3
and remote access, B-3
asynchronous operations, 4-14
buckets, 4-1
cells, 4-1
CONNECT operation, 4-7
contiguity, 4-4
deferred write, 4-14
DELETE operation, 4-7
design, 4-1
allocation, 4-3
bucket size, 4-2
MRN, 4-5
record format, 4-2
directory operations, 4-15
DISCONNECT operation, 4-8
file operations, 4-15
FIND operation, 4-8
FLUSH operation, 4-10
GET operation, 4-10, 4-12 to 4-13
I/O techniques, 4-14
MRN, 4-4
multiple access streams, 4-15
multiple buffers, 4-15
prologue, 4-1
PUT operation, 4-11 to 4-12
random access
by key, 4-1
by RFA, 4-1
record operations, 4-7
record size, 4-2
record transfer modes, 4-12
move mode, 4-12
REWIND operation, 4-12
sequential access, 4-1
shared access, 4-6
block access, 4-6
record access, 4-6
sharing declarations, 4-6
stream operations, 4-7

structure, 4-1
conceptual, 4-1
physical, 4-1
UPDATE operation, 4-12 to 4-13
Relative record number
See RRN
REMOVE operation, 1-17
wildcard loops, A-4
REMOVE operations
remote access, B-1
RENAME operation, 1-17
file versions, A-3
wildcard loops, A-4
RENAME operations
remote access, B-1
Resident libraries, 8-1, 8-8
RMSRES, 8-6, 8-9
Restoring files, 1-10, 1-12
Retrieval pointers, 8-11
in memory, 8-11
on disk, 8-11
REWIND operation, 1-15
indexed files, 7-6
relative files, 4-12
sequential files, 3-11
RFA, 1-12
relative files, 4-1
sequential files, 3-2
RFA access
to relative files, 4-1
to sequential files, 3-2
RMS-11 defaults, 2-2
RMS-11 File Back-Up Utility
See RMSBCK
RMS-11 File Conversion Utility
See RMSCNV
RMS-11 File Design Utility
See RMSDES
RMS-11 File Display Utility
See RMSDSP
RMS-11 File Restoration Utility
See RMSRST
RMS-11 Indexed File Load Utility
See RMSIFL
RMS-11 resident library
See RMSRES
RMSBCK, 1-12
RMSCNV, 1-12
populating files, 6-26 to 6-27
RMSDES, 1-12, 1-18
bucket fill size, 6-29
bucket size, 6-22
relative files, 4-3
DEQ, 3-5, 4-4, 6-26
initial allocation, 3-4, 4-4, 6-26
key segments, 6-7
MRN, 4-5
placement control, 6-14
RMSDSP, 1-12, 1-19
RMSIFL, 1-12
populating files, 6-26 to 6-27
RMSRES
and I/O paging operations, 8-10
and overlays, 8-6
clustered, 8-7
overlays, 8-9
supervisor mode, 8-7
task building, 8-6
RMSRST, 1-12
Root, 1-7
indexed files, 5-5
RRN
relative files, 4-1
sequential files, 3-2
RRV, 5-12, 5-16
RSTS/E
remote access, B-3
RSX-11M/M-PLUS
remote access, B-3

SEARCH operation, 1-17
SEARCH operations
remote access, B-1
Search times
indexed files, 5-8
Seconday index data records
See SIDR
Sector
See Placement control
Segmented keys, 6-7
Sequential access, 1-7 to 1-8
FIND, 1-16
GET, 1-16
PUT, 1-16
to blocks, 1-20
to indexed files, 5-16
to records, 1-12
to relative files, 4-1
to sequential files, 3-2
Sequential file organization, 1-3, 1-11
access declarations, 3-6
allocation
DEQ, 3-4
initial, 3-4
and remote access, B-3
asynchronous operations, 3-14
cells, 3-2
CONNECT operation, 3-7
deferred write, 3-14
design, 3-1
allocation, 3-4
contiguity, 3-5
medium, 3-3
record format, 3-3
directory operations, 3-15
DISCONNECT operation, 3-7
end-of-block indicators, 3-2
end-of-file, 3-2
FCS-11 compatibility, 3-2
file operations, 3-15
FIND operation, 3-7
FLUSH operation, 3-9
GET operation, 3-9,

3-13 to 3-14
I/O techniques, 3-14
MBC, 3-15
multiple access streams, 3-15
multiple buffers, 3-14
PUT operation, 3-10, 3-13
random access
by key, 3-2
by RFA, 3-2
record operations, 3-7
record size, 3-2
record transfer modes, 3-13
locate mode, 3-14, 4-13
move mode, 3-13 to 3-14
REWIND operation, 3-11
sequential access, 3-2
shared access, 3-5
block access, 3-6
record access, 3-6
record structured files, 3-6
with undefined records, 3-6
sharing declarations, 3-6
stream operations, 3-7
structure, 3-1
conceptual, 3-2
physical, 3-1
TRUNCATE operation, 3-12
UPDATE operation, 3-12 to 3-14
user-provided interlocks, 3-6
Shared access, 1-17
access declarations, 2-6
read-only, 2-6
read/write, 2-6
and high-level languages, 2-7
application design consideration, 2-5
bucket locking, 2-8
deferred write
to relative files, 4-14
multiple access streams, 2-9 to 2-10
programming considerations, 2-10
sharing declarations, 2-7
no sharing, 2-7
read-only, 2-7
read/write, 2-7
user-provided interlocks, 2-7
system protection codes, 2-5
to indexed files, 7-1
block access, 7-1
record access, 7-1
to relative files, 4-6
block access, 4-6
record access, 4-6
to sequential files, 3-5
record structured, 3-6
with undefined records, 3-6
Sharing
See Shared access
Sharing declarations, 2-7
indexed files, 7-1
no sharing, 2-7
read-only, 2-7
read/write, 2-7
relative files, 4-6
sequential files, 3-6
user-provided interlocks, 2-7
SIDR, 5-5, 7-9
changeable keys, 6-10
duplicate keys, 6-8 to 6-9
Space
application design consideration, 2-4, 8-1
data storage, 2-5
I/O buffer size, 2-5
task size, 2-5
Spanning blocks, 1-16, 1-19, 3-1
variable length records, 2-16
Speed
application design consideration, 2-3, 8-1
Standard ODL files, 8-3, 8-5
and file organization, 8-5
DAP11X, 8-6
RMS11S, 8-5
RMS11X, 8-5
RMS12X, 8-6
Stream operations, 1-15
See also Operations
Stream record format, 1-11, 2-16
terminators, 2-16
Streams
See Access streams
String keys, 6-3
segmented, 6-7
Structure
indexed files
conceptual, 5-4
physical, 5-2
relative files, 4-1
conceptual, 4-1
physical, 4-1
sequential files, 3-1
conceptual, 3-2
physical, 3-1
Supervisor mode, 8-7, 8-9
Synchronous operations, 8-2

Task
executable, 8-1
size, 2-5
structure, 1-13
Task builder, 1-13
and optimizations, 8-10
and overlay structure, 8-10
and remote access code, B-1
and RMS-11 routines, 8-1
and RMSRES, 8-6
Terminals, A-1
Terminators
stream records, 2-16
Track
See Placement control
TRUNCATE operation, 1-15

and shared access, 2-6
sequential files, 3-12
Two-byte signed integer keys, 6-4
Two-byte unsigned binary keys, 6-5
Type
file, A-3

Undefined record format, 1-11, 2-17
Unit-record devices, A-1
UPDATE operation, 1-15
and ODLs, 8-5
and record size, 2-16
and shared access, 2-6, 2-10
and stream records, 2-17
changeable keys, 6-10
deferred write, 7-9
duplicate keys, 6-9
indexed files, 5-14, 7-6, 7-9
move mode, 3-14
indexed files, 7-7 to 7-8
relative files, 4-12 to 4-13
sequential files, 3-13
relative files, 4-12
sequential files, 3-12
User buffers, 1-14
User-provided interlocks, 3-6
sharing declaration, 2-7
Utilities, 1-12, 2-11
RMSBCK, 1-12
RMSCNV, 1-12
populating files, 6-26 to 6-27
RMSDES, 1-12, 1-18
bucket fill size, 6-29
bucket size, 4-3, 6-22
DEQ, 3-5, 4-4, 6-26
initial allocation, 3-4, 4-4, 6-26
key segments, 6-7
MRN, 4-5
placement control, 6-14
RMSDSP, 1-12, 1-19
RMSIFL, 1-12
populating files, 6-26 to 6-27
RMSRST, 1-12

Variable with fixed control
See VFC
Variable-length record format, 1-11, 2-15
and medium, 2-15
VAX/VMS
remote access, B-1, B-3
VBN, 8-11
access, 1-20
areas, 6-10
Version
file, A-3, A-5
default, A-6

VFC record format, 1-11, 2-16
Virtual block number
See VBN
Virtual-to-logical block mapping
optimizations, 8-11

WAIT operation, 1-15
Wildcards
file specification, A-2 to A-3
loops, A-4
Windows, 8-11
turning, 8-11
areas, 8-13
contiguity, 8-12
F11ACP size, 8-13
optimizations, 8-12
window size, 8-12
WRITE operation
and shared access, 2-6

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

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

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

Do Not Tear - Fold Here

Cut Along Dotted Line

Printed in U.S.A.