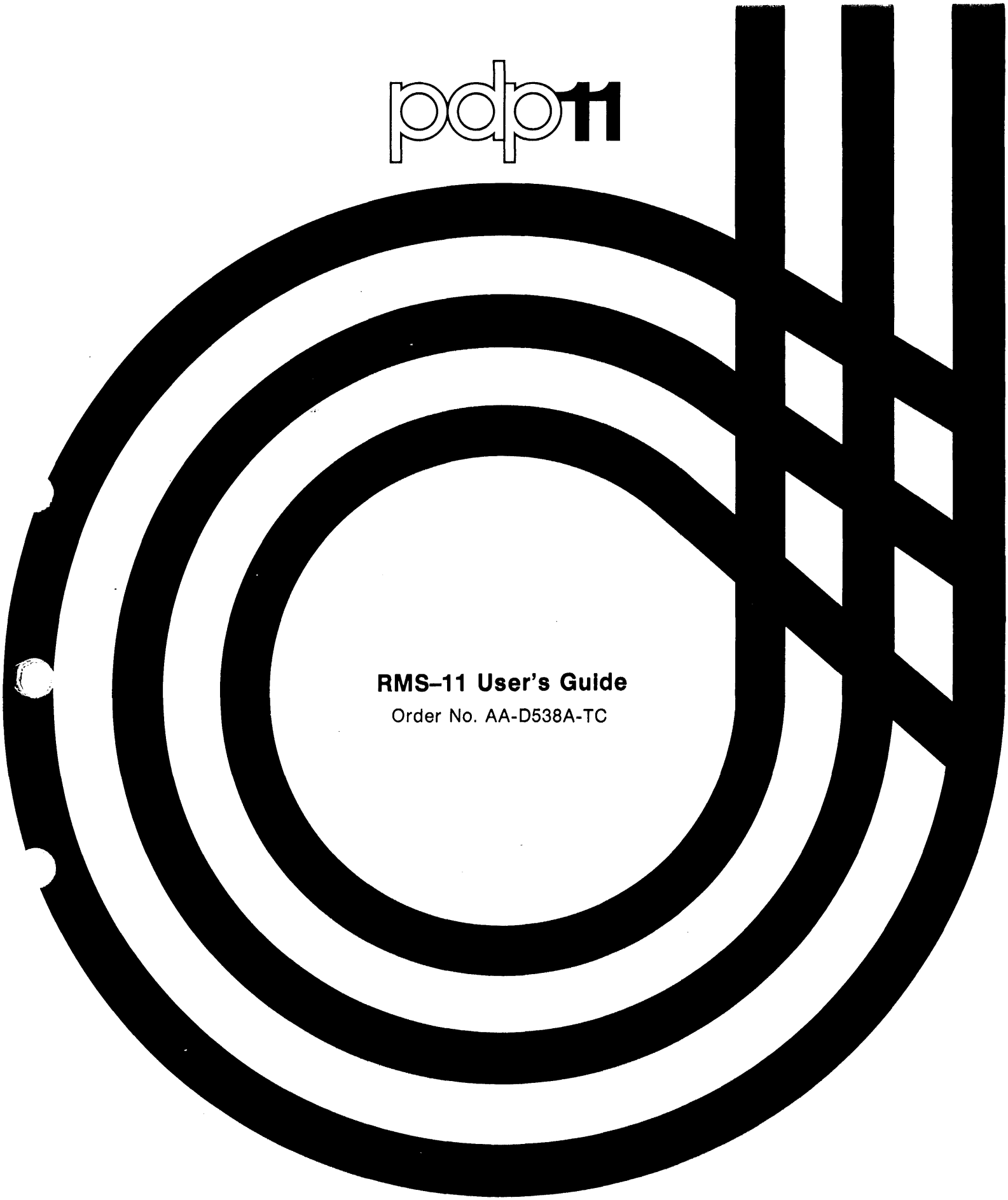


RMS–11 User's Guide

Order No. AA-D538A-TC

digital

March 1979

This manual describes the concepts, structure, and usage of Record Management Services (RMS) software for the PDP-11. Though it describes syntax for all programming languages supported by RMS-11, this Guide is not a definitive source for that information.

RMS–11 User's Guide

Order No. AA-D538A-TC

SOFTWARE: RMS-11 V1.8

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

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

The software described in this document is furnished under a license, and may only be used or copied 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 or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	IAS
DECnet	MASSBUS
DECsystem-10	PDP
DECSYSTEM-20	RSTS
DECtape	RSX
DECUS	UNIBUS
DIBOL	VAX
DIGITAL	VMS
FOCAL	

Contents

Chapter 2 Application Design

Chapter 3 Sequential File Applications

Chapter 4 Relative File Applications

Chapter 5 Indexed File Organization

Chapter 6 Indexed File Design

Chapter 7 Indexed Task Design

Chapter 9 RMS-11 Utilities

Appendix A RMS–11 and the Operating Systems

Appendix B RMS–11 and the Programming Languages

Appendix C RMS–11 Disk-Resident Overlays

Appendix D RMSDFN Command Utility

Appendix E Utility Error Messages

Appendix F Magnetic Tape Handling

Glossary

Index

Figures

Tables

Commercial Engineering Publications Typeset this manual using DIGITAL's TMS-11 Text Management System.

827ALL

Preface

Record Management Services for the PDP-11 (RMS-11) provides powerful data management capabilities. The *RMS-11 User's Guide* tells you about those capabilities and how to use them. This manual is designed for instruction and for reference:

- If you read the manual according to the map in PRE-1 (see "Structure of the Manual"), you can learn practically everything you need to know about RMS-11, except its language implementation.
- If you use the manual as a reference, its structure and extensive index make it easy to consult.

RMS-11 is a set of software routines that transfers data between a running program, which uses data in a logical form called records, and the file processor portion of an operating system, which maintains the physical structure of the data on a storage device.

NOTE

You can use RMS-11 Indexed files only if you have purchased the RMS-11K software product. Your system manager can tell you if your system includes this capability.

Prerequisites

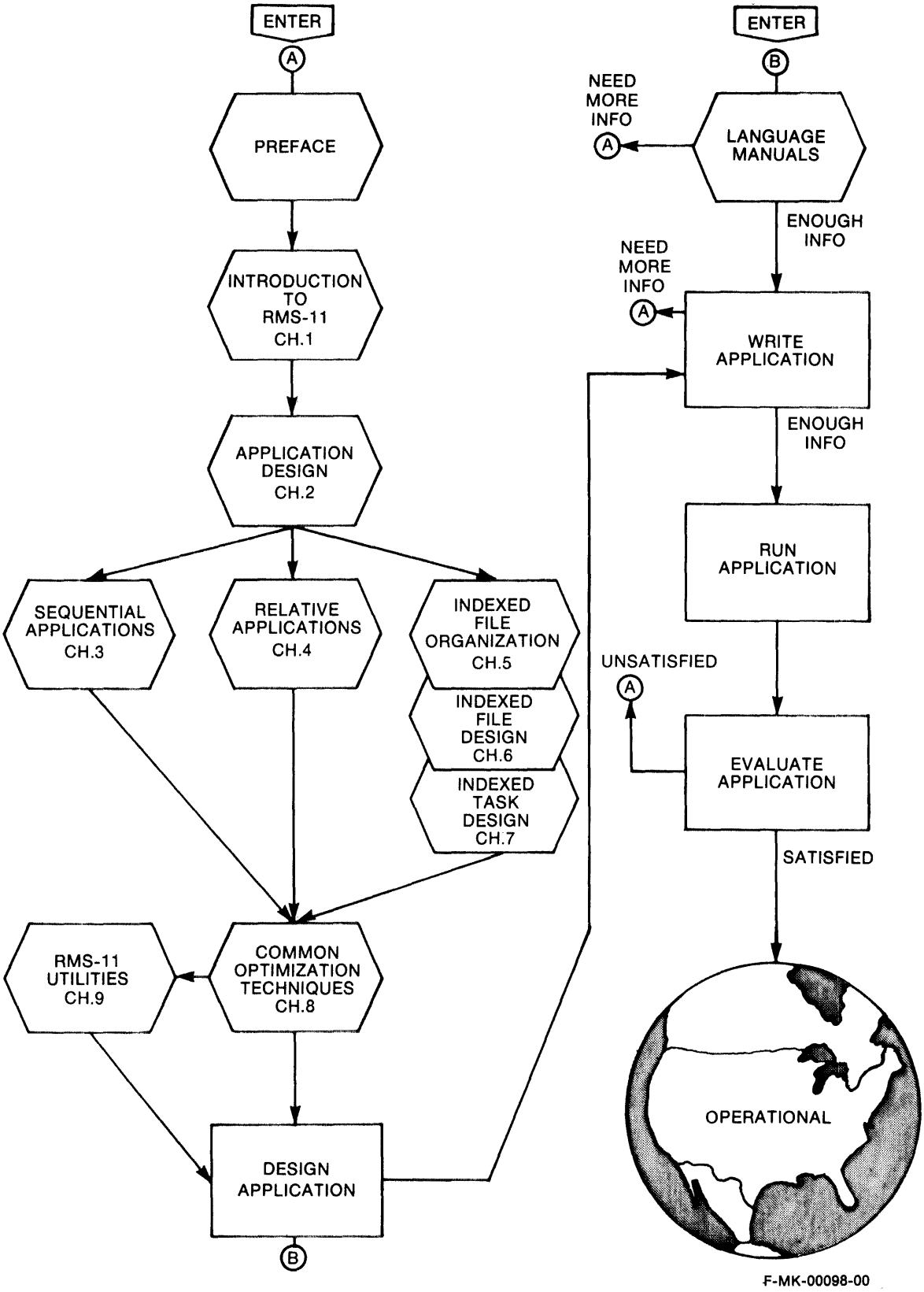
To read this manual, you should be either a MACRO-11 or higher level language programmer who wants to improve the performance of an application you have written or are planning to write. In other words, you should have a working knowledge of a DIGITAL programming language that supports RMS-11 and some motivation to inquire into the workings of RMS-11.

NOTE

This manual describes the functions of RMS-11. Only MACRO-11 programs can use the full set of capabilities. The higher level languages support a subset of those capabilities. You must use your language documentation to determine:

- what RMS-11 facilities you can actually use in your higher level language
- the syntax to use them

Figure PRE-1: Map for the *RMS-11 User's Guide*



Structure of the Manual

Figure PRE-1 shows the paths through this Guide you should follow in the process of writing your applications. The following paragraphs describe the parts of the manual shown in that map:

Chapter 1 addresses the RMS-11 beginner: it discusses basic RMS-11 concepts that are common to all programming languages in which RMS-11 applications can be written.

Chapter 2 introduces RMS-11 design, why it is necessary, and its general premises.

Chapters 3-7 describe the file organizations, their structures, and file and task design considerations.

Chapter 8 covers common techniques that can be used to optimize RMS-11 applications, regardless of the file organization selected.

Chapter 9 describes the RMS-11 utilities. These tasks provide RMS-11 facilities to all users, programmer or operator, higher level language or MACRO-11.

Appendixes provide supplementary information, such as:

Implementation of RMS-11 on the various operating systems
RMS-11 features supported by DIGITAL programming languages
How the higher level languages map RMS-11 error codes
Documentation of the RMSDFN utility
Utility error messages

The *Glossary* reprises all terms defined in this manual.

Associated Documents

The RMS-11 documentation set contains the following manuals:

RMS-11 User's Guide
RMS-11 MACRO-11 Reference Manual
RMS-11 Installation Guide

You must also use operating system and language documentation. See the Documentation Directory for your operating system.

Implementation in Languages

RMS-11 is the record management software for the following PDP-11 programming languages. See Appendix B for more information.

BASIC-PLUS-2
DIBOL
MACRO-11
PDP-11 COBOL
RPG II

Implementation in Operating Systems

RMS-11 is available as an interface between user application programs and data storage devices on the following DIGITAL operating systems. See Appendix A for more information.

IAS
RSTS/E
RSX-11M
VAX/AME

NOTE

The TRAX system uses RMS-11 as part of its central data manager. See TRAX documentation for a discussion of the RMS-11 interface on that operating system.

Documentation Conventions

RMS-11 operates similarly on the supporting operating systems (see Appendix A); RMS-11 is the common record access software for most higher level languages. Therefore, it should be possible to produce a single manual describing that operation. However, the differences among operating systems and languages present a barrier to that unification. The following conventions are designed to enable you to hurdle that fence.

Differences in Operating System Functions

Details that are common to the operating systems are printed in black, and the details that are specific to one operating system or another are printed in color as follows:

- RSTS/E-specific information is printed in red.
- IAS-, RSX-11M-, and VAX-specific information is printed in blue.

Differences in Language Support for RMS-11

Differences in language functionality are noted only in Appendix B. You should also consult the language documentation. However, note that MACRO-11 programs can use all facilities described. Therefore, MACRO-11 error codes are used in the discussions. Appendix B contains a chart showing how the higher level languages map these codes.

Terminology

The operating systems may have different terms for the same process.

Example "Installed" to an RSX-11M user means essentially the same as "CCL" to a RSTS/E user in that the user just has to enter a short command to activate the task.

In this manual, these terms are reconciled under a cover term with a specific definition of what that term means to the different operating systems. The most pervasive cover terms are defined here. Other terms are defined in CONVENTION notes as they are needed.

Cover Term	Definition
filespec	file specification; see your operating system documentation for syntax and Appendix A for RMS-11 restrictions on that syntax
account	the User File Directory (UFD) or its number; also known as User Identification Code (UIC) or the Project, Programmer Number (PPN)
[act nbr]	account number; used in filespecs
file-name	the name of the file
extension	the file-name extension to the right of the dot also called file-type
protection code	the letter codes R, W, E, and/or D and the number codes 0 through 255 that represent the level of protection for a file by the operating system
wild card	the presence of an asterisk (*) in a field of the filespec indicates that any value is acceptable for that field

User Interface

- Even though the IAS/RSX-11M monitor prompt is not shown, it is assumed before any utility or other system task is called.
- Even though the RSTS/E system response is not shown, it is assumed after any utility or other system task is terminated.

Terminal Displays

Any examples of terminal displays show the software messages with underlines; your inputs are not underlined.

Example From "Task Building Against the RMS-11 Resident Library," Section 8.1.2.3:

```
TKB>USER.TSK,USER.MAP=USERA,USERB,USERC
TKB>/
ENTER OPTIONS:
TKB>LIBR=RMSRES:RO
TKB>//
```

Chapter 1
Introduction to RMS–11

Your business, whether commercial, scientific, governmental, or educational, relies on data. That data indicates the state of your business; that data helps you control the future of the business. Therefore, you want fast, effective access to it.

Computer hardware, with its speed (times in nanoseconds) and its mass data storage (millions of characters), provides the means for that kind of access. The problem is converting the data from the way you talk about it to the way the computer system can handle it—and back again.

RMS–11 is the translator between you and your system. A set of software routines used by your programs, RMS–11 processes data for you in units called records. This introduction takes you slowly into RMS–11, beginning with common concepts of data organization. Then the discussion turns to how RMS–11 implements those concepts and fits into its environment, that is, your programs, language, and operating system.

1.1 Concepts of Data Organization

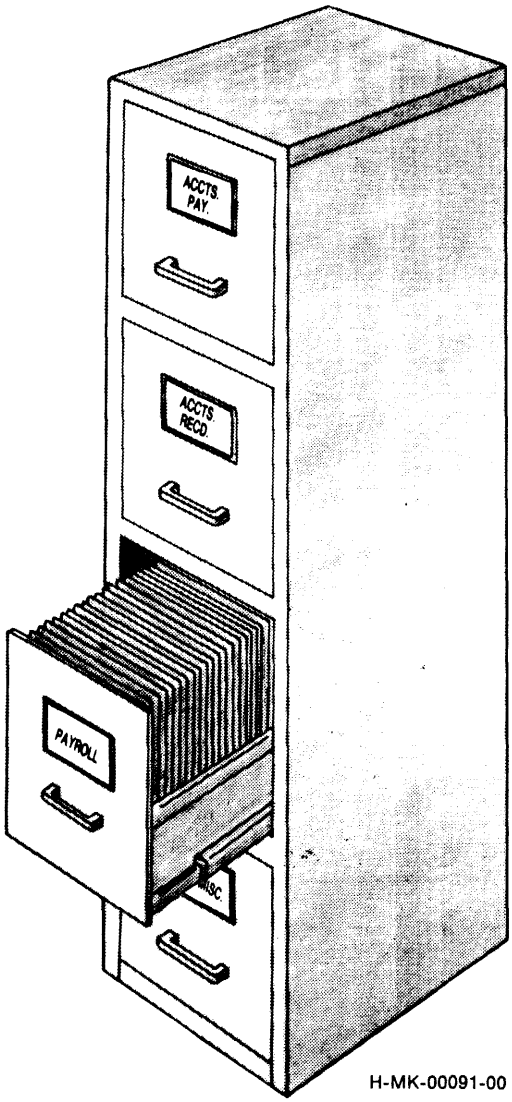
First, let's examine the concepts involved in organizing data, using images from a noncomputer environment you're familiar with.

File When data is stored on paper, people gather it in containers called *files* (see Figure 1–1). A file not only keeps related data in one place, but it also segregates that data from other information. The term *file* applies to the data as well.

Example Personnel information, including names, address, job titles, pay rates, and so on.

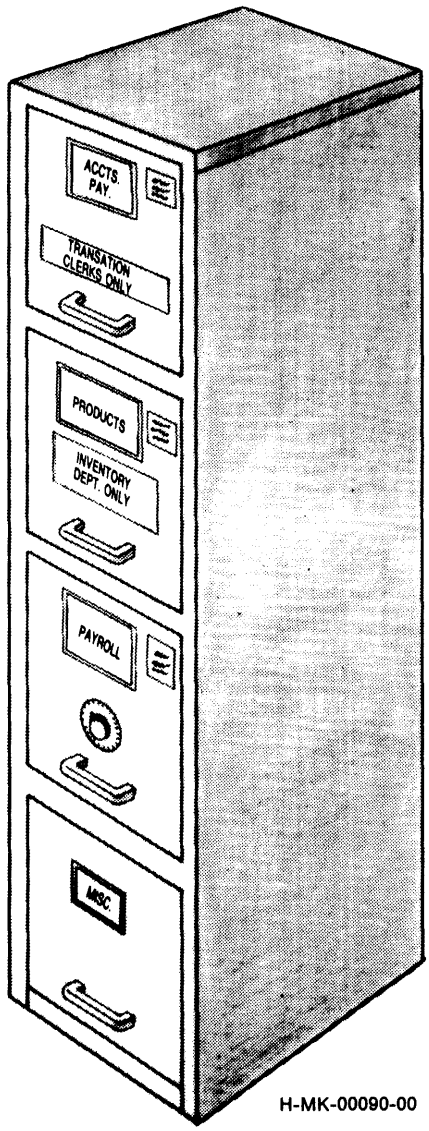
Example Product information, including part numbers, prices, specifications, discount rates, and more.

Figure 1-1: Files



H-MK-00091-00

Figure 1-2: Files with Attributes



H-MK-00090-00

Access An advantage of the segregation provided by files is controlled access. Some files, like payroll and budget files, are available only to a restricted group of people. Other files, like transaction or inventory files, are used by a larger group of different people. And there are some files that everyone uses, such as a telephone book.

You can extend the access concept further and say that within the group of people who can use a file, only certain ones are allowed to put data into it or change the data already in it; others can only read what's in the file.

Example Telephone numbers, especially those of customers, should be changed only by authorized people so that the numbers remain current and accurate.

File Attributes

A file cabinet has features that identify it, that tell what's in it, even how the data is organized. These features, or *attributes*, can be as a simple as "the gray three-drawer" or "the one by the window."

As file requirements become complicated and cabinets multiply, more complete and precise identification is necessary. Then, the files acquire names or numbers, signs announcing who can use the file, cards cross-referencing the data in the drawers, and so on (see Figure 1–2).

Record

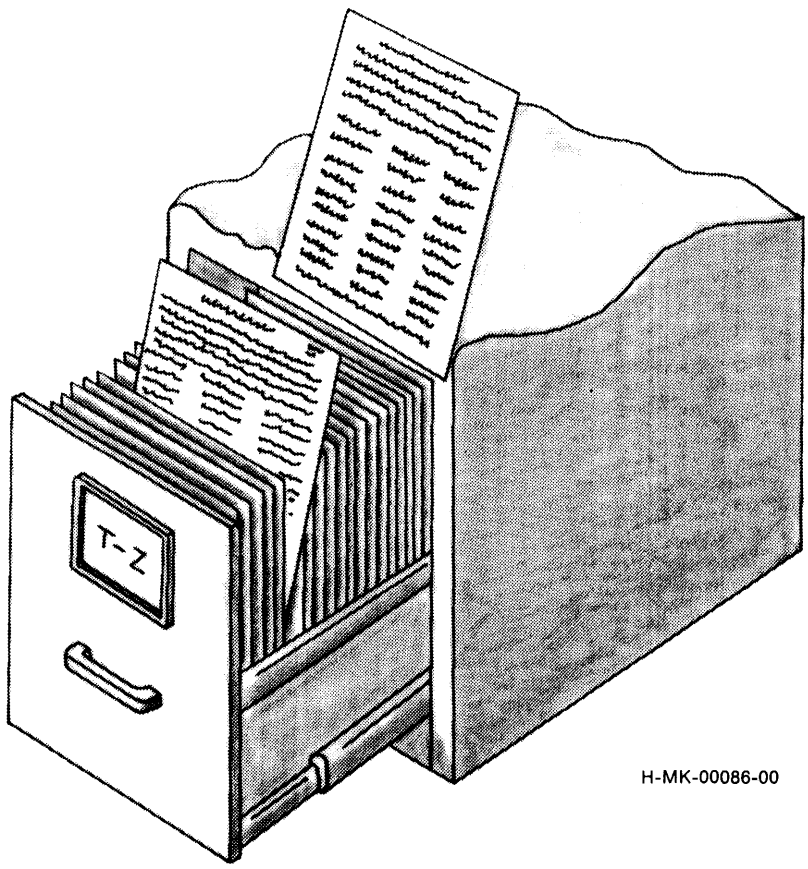
Each file contains discrete groups of items whose form is repeated throughout the data. Each group, or *record*, represents an entity, and a file consists of a series of such entities.

Example In a personnel file, all the information on an employee constitutes a record; therefore, the number of records in the file equals the number of employees.

Example In an inventory file, all the information on a stock item comprises a record.

Within each record are the specific items of data with which you are concerned.

Figure 1–3: Records in a File



Record Format

On paper, a record can be a form (see Figures 1–3 and 1–4), and different records require different forms.

Some forms are always the same length. Their information does not expand with time or use.

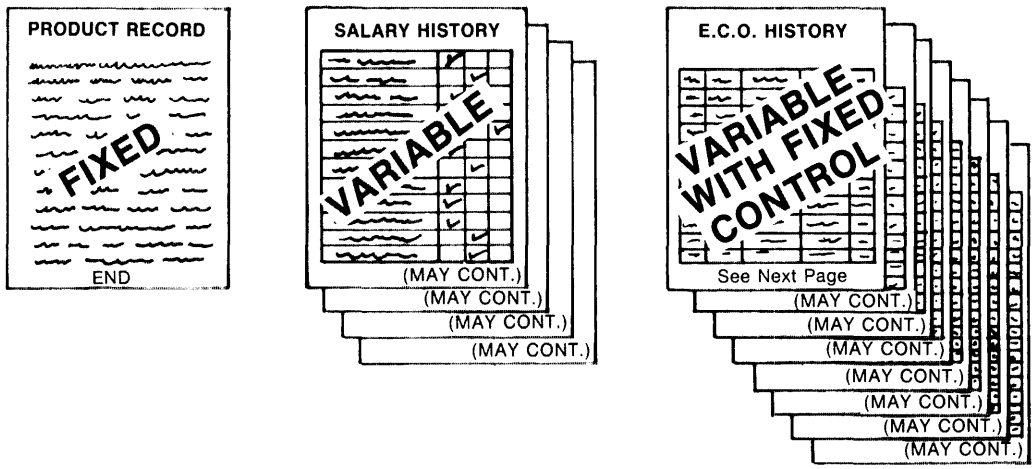
Example A product information form: if the facts about a product change, you fill out a new form; if you add products to the file, you fill out a new form.

Other forms vary over time and use. They have continuation sheets or some other extension.

Example An employee payroll record: an employee with the company ten years has more data on file than a new employee.

Other records use a combination of these two formats.

Figure 1–4: Record Formats



Q-MK-00073-00

Access Modes

Once you have records in a file, you get, or *access*, them in two ways (see Figure 1–5):

NOTE

Record access not only means retrieving a record from a file; it also includes putting the record into the file.

Sequentially
: You pick a point in the file and then access records one at a time. Often you start at the beginning of the file because you want to look at each record in the file—probably for a report you are compiling. At other times, you start partway through the file.

Context

To read each record, you take it out of the file. You mark the position each time with a card or upside-down folder 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 use two fingers to hold open a space. Then you take a sheet from the stack of records and slip it into position between your fingers. You move your finger behind the record you added, holding open a space for another form. Repeating these steps, you insert the entire stack into the file in the sequence they are available to you.

In either type of access, you move through the records consecutively. Each record inserted or retrieved relates to the record accessed right before it.

Randomly

You determine the record you want on some basis other than its order of occurrence in the file. Perhaps you have a list of locations. Then you reach into the file at the exact record's location. Each record selection is independent of the previously accessed record and of the next record processed.

To randomly access records in a file, you use an identifier. You can assign a number to each record and use that to put the record into the file and get it out again. But it is simpler when the identifier is part of the record itself. In fact, it helps if you can locate a record via more than one identifier, or *key*, within the data.

Or you can use the physical location of the record. For example, you have a file stored in a three-drawer cabinet. Inside each drawer you make tiny slots, each capable of holding one record form. Each slot has a number. You can, therefore, identify a record's location, or *address*, within the file, using a drawer number and a slot number within that drawer.

There are times, of course, when you access records in a file sequentially and randomly. Usually, you randomly access the first record in a series and then sequentially access the records in that series.

Example The records in a personnel file contain departmental codes. If the records of all personnel in a department are grouped together, you can produce a report on one department by randomly accessing the first record in the file with the department's code and then reading the consecutive records with that code.

Record Operations

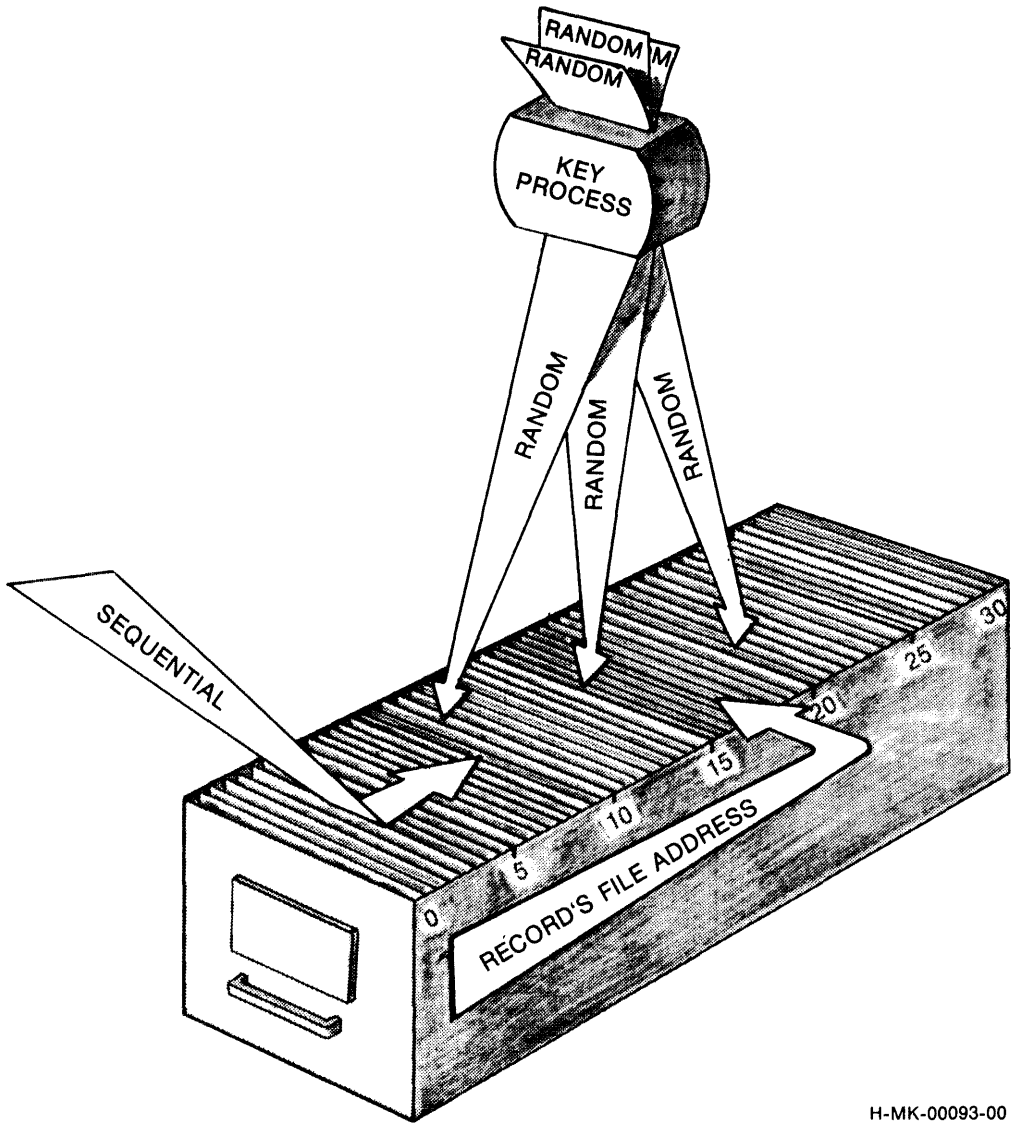
What will you do with a record when you come to its place in the file?

- Verify that the record exists in the place it should.
- Read the record, that is, examine its contents.
- Insert a record in the position you've located.
- Revise the contents of the record, that is, change some data in it.
- Remove the record from the file.

File Organizations

Generally, the person who uses a file establishes a method of organizing the records within it. This method reflects the file's use and dictates the information and time needed to locate a record.

Figure 1-5: Record Access Modes



Sequential For instance, you want to file a series of records:

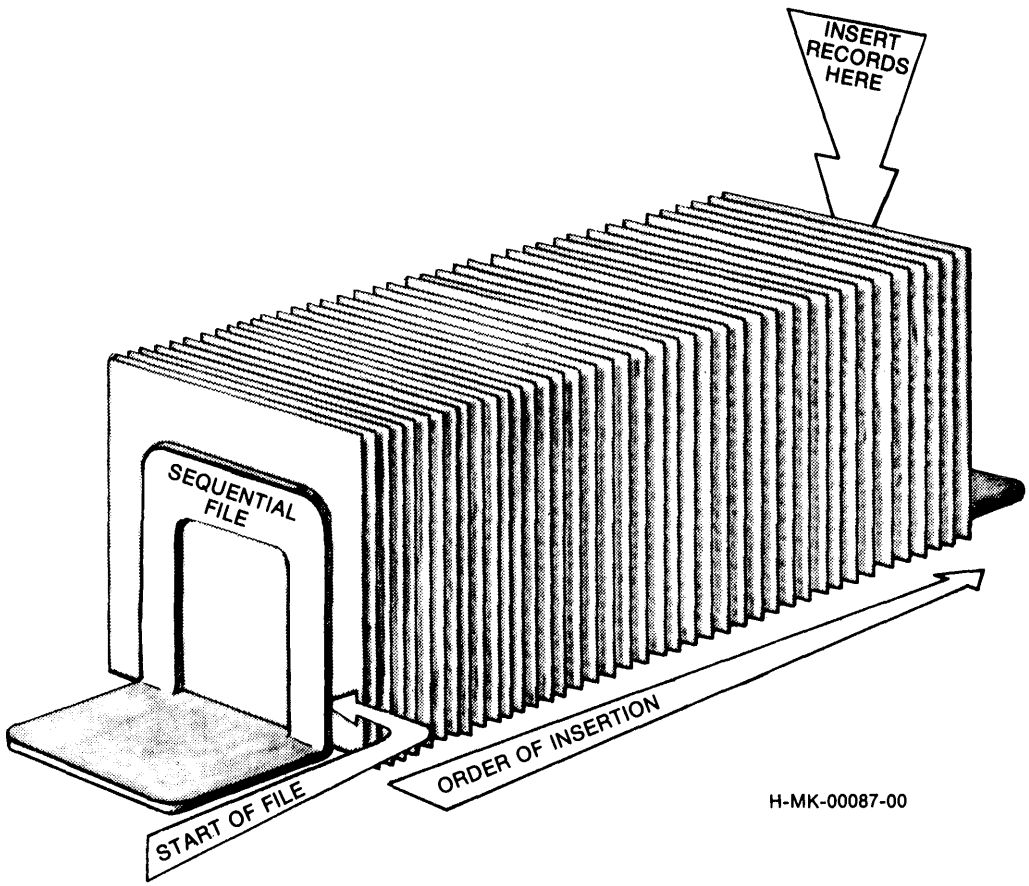
- You have little or no need to access the records randomly.
- You generally use all the records in the file in an unvarying order when you open it.

With these requirements, you can organize the records by stacking them in a file:

- The records assume the physical sequence in which they are put into the file.
- There are no spaces, where records could be inserted later, left in the series. Each record, except the first, has a record before it; each record, except the last, has a record following it.

This is a sequentially organized file (see Figure 1–6). Its overhead and upkeep are minimal. To put a record into the file, you just put it after the last record already there.

Figure 1–6: Sequential File Organization

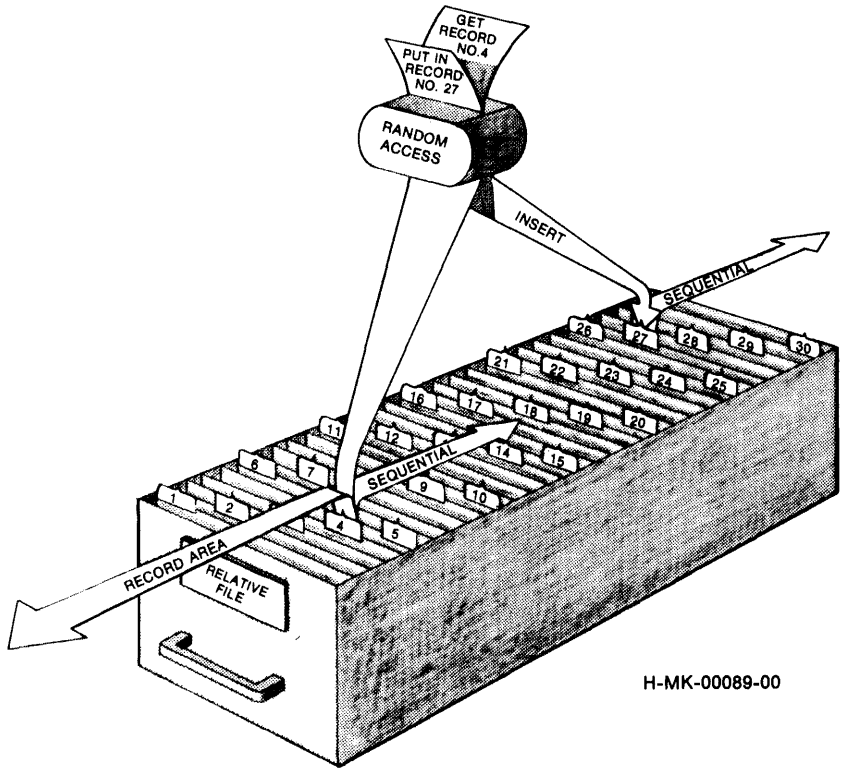


Relative However, if you want more access flexibility, you can change the organization. This time, set up a series of file folders, numbered in sequence from "1" to the last folder in the file. Each folder is the same size; it holds only one record, but it can be empty.

This *Relative* organization (see Figure 1–7) enables you to access specific records more easily. You do not have to look sequentially through the records before the one you want (though you can). You use the numbers on the folders to locate or insert records. You can even code the forms to make use of some internal data items as the basis of these *relative record numbers.*

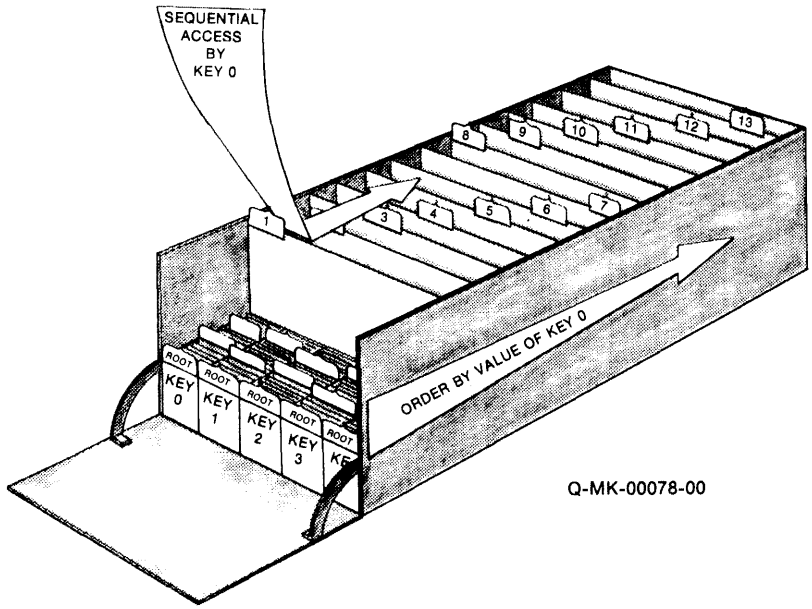
Indexed You have a large file and most of the time, you randomly access its records. The list of relative record numbers is taking more time than it's worth, because you use several kinds of information to look for records. You are ready for an Indexed file, though you may still want to sequentially access the whole file to compile reports.

Figure 1-7: Relative File Organization



When you open the file drawer, you see papers neatly stacked, with numbered tabs sticking up (see Figure 1-8). At the front of the drawer there is a set of small card files; inside them are groups of cards separated by dividers. The cards in each file are an index to the records in the back of the drawer. You look at the record you have to insert and find the data item marked "KEY." Using the information there, you consult an index to determine where you should insert the record.

Figure 1-8: Indexed File Organization



Or you can find a record by looking for a specific value in one of the key files.

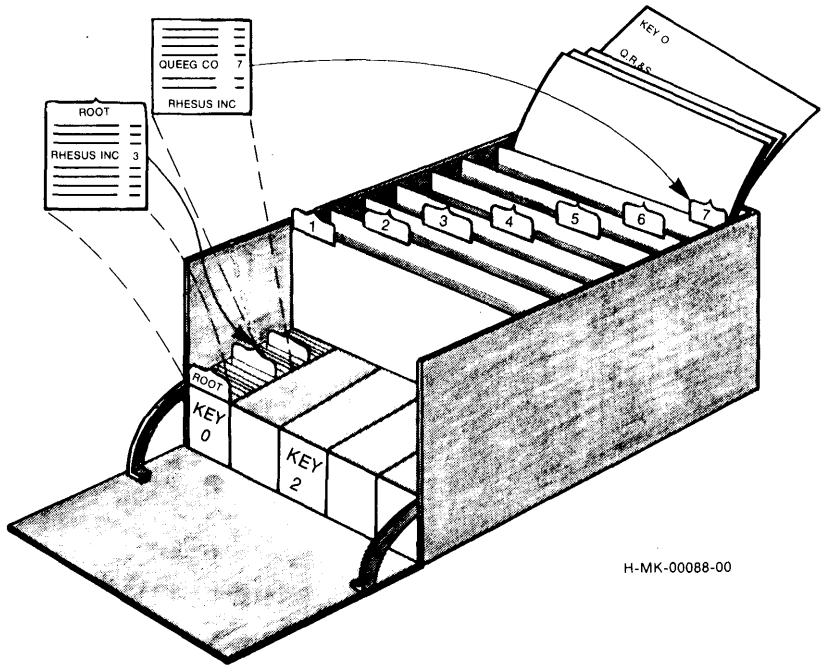
Example You want the record of a transaction with Q,R,&S, so you open the Transaction File drawer. Inside, the records are filed at the back and there are five indexes at the front (see Figure 1–9). You know that "Q,R,&S" is the *Primary Key* of the record you want and you open the index labeled "0".

The first record in the index is the *Root*. It has a list of Primary Key values on it; each value is paired with a number. You look down the alphabetical list until you find a value equal to or greater than "Q,R,&S". You find "Rhesus INC" and the number next to it is "3".

You put the Root back in the index and go to the first divider and the third record behind it. You see that "Rhesus INC" is the last entry on this card, but you scan up the list to find a value closer to the one you're looking for. You find "Queeg Company" and the number "7".

So you reach into the back of the drawer, trailing down the tabs to the seventh one. Behind its divider there is a stack of records. You search sequentially through them until you find the Q,R,&S transaction.

Figure 1–9: Random Access Example



Example Using the Transaction File described in the last example, you want to find a record, but all you know is its transaction number. Fortunately, the *Third Alternate Key* for the file is transaction number. You open the index labeled "2" and look at its Root card. Employing the technique illustrated in the previous example, you move from the Root through the rest of the index. Behind the last divider, you find a record on which the transaction number you are looking for is listed. Next to the number is the code "7/5".

You reach into the back of the drawer, pulling the seventh tab forward. Then you count the records behind it. The fifth one has the right transaction number on it. You notice that the transaction was made with the firm of Q,R,&S.

Utilities Once you establish files and their records, you'd like to do some things with each file as a whole:

Back up
: The data in a file is valuable, or you would not keep it. You should have a duplicate of your records in some other place in case something happens to the original. Therefore you need the ability to back up a file quickly and efficiently.

Restore
: If something does happen to the original records, you must replace them with the back ups quickly and smoothly.

Display
: You need the ability to produce a list of your files, with their names and other attributes.

Convert
: Files do grow beyond your estimate, and to increase their usefulness, you should change their organization, say, from a sequentially organized files to ones with indexes. Or the opposite could happen to a file and you need to make it simpler. Then, there are times when a file set up for one purpose could be used for another application. In that case, you need to copy the organization and contents of the original file into the new one, changing some attributes.

Define
: Then, when you've designed the file, you should also have a procedure for creating the file.

Indexed File Load
: Creating Indexed files can be complicated and time-consuming. You could use a procedure that takes a file of any organization and produces an optimal Indexed file quickly and efficiently.

That's a synopsis of data organization concepts. You should recognize them from your experiences with paper. But now you're using a computer to organize your data: anything you can do at your desk, you ought to be able to do with your computer.

1.2 RMS–11 Implementation of Data Organization

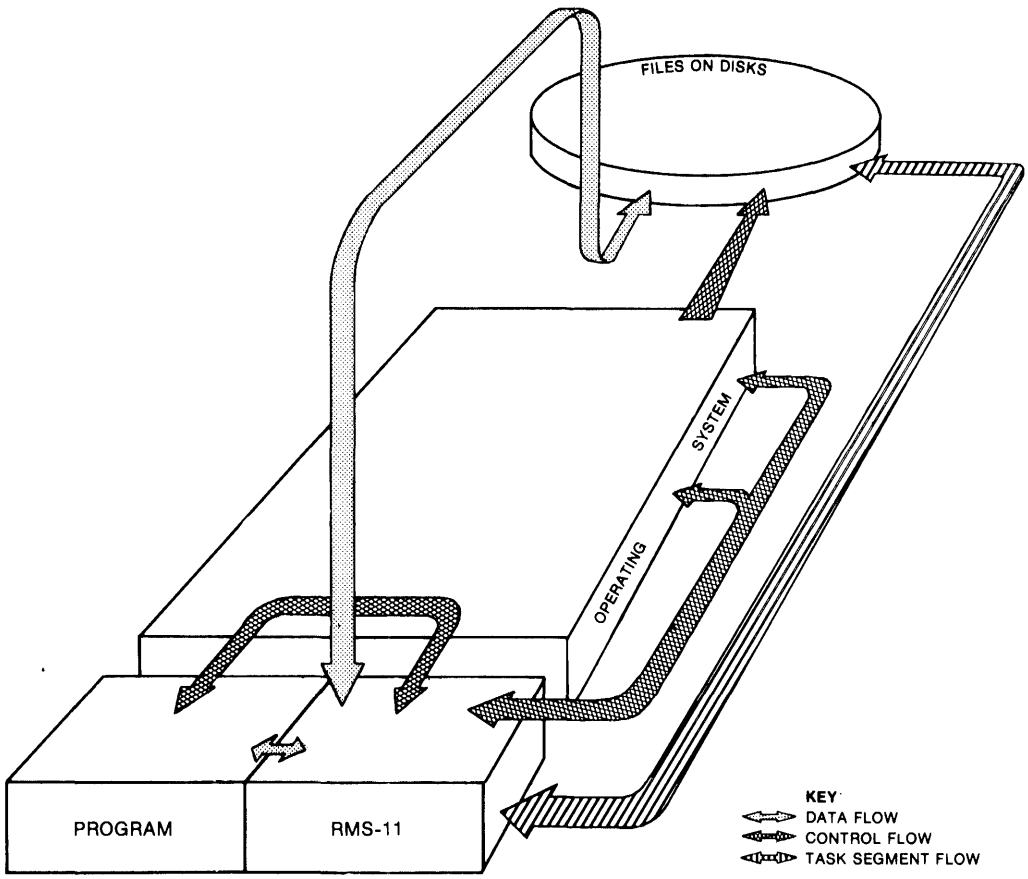
The software routines called RMS–11 organize data on your DIGITAL computer, implementing the concepts we have just discussed. RMS–11 interfaces your data processing programs with the rest of the computer system.

Your computer system consists essentially of layers of software and hardware (see Figure 1–10), each of which has a responsibility in the data management process regulated by RMS–11:

- The hardware devices store the data.

- The operating system controls the hardware to provide the record containers called files.
- RMS-11 controls the internal structure of the computerized files, providing file organizations and record formats, access modes, and operations.
- Your application program drives the process by initiating data processing operations.

Figure 1-10: RMS-11 in Its Environment



The rest of this introduction discusses these system layers.

1.2.1 Hardware Data Structure

The prime storage device on today's computer systems is the disk drive[1]. The disk itself consists of one or more circular pieces of metal (called *platters*) and the drive is the mechanical and electronic equipment to read and write information on the platters.

[1] Although magnetic tapes also provide significant data storage capability, they require sequential access. Therefore, in modern business environments, where fast random access is important, disk drives are used for on-line storage. Magnetic tape, however, does have its uses; details of magtape handling via RMS-11 are covered in Appendix F.

Data is stored on disk platters magnetically, much the same as sound is recorded on tape. The structure of that data is modular, expressed in hierarchical units (see Figure 1–11):

Bit
: A *bit* is the smallest storage location recognized by the hardware. A bit is an area of the disk surface where magnetic orientation can be changed to one of two recognized values, conventionally designated "0" and "1".

Byte
: A *byte* contains eight bits and is frequently used to represent an alphanumeric character with the American Standard Code for Information Interchange (ASCII) codes. Other methods of representing data, particularly numeric data, require two or more bytes at a time.

Sector
: A *sector* consists of 512 bytes on most disks supplied by DIGITAL.

Track
: A *track* consists of the sectors at a single radius on one disk platter. One read/write head of the disk drive can access a track without changing position.

Cylinder
: A *cylinder* consists of the tracks at the same radius on all disk platters. The disk drive's head structure can access a cylinder without changing position.

1.2.2 Software Data Structure

The operating system software makes the computer's hardware available to the user. As it turns out, this user is RMS–11, but it is RMS–11's purpose to complete the logical chain, that is, to make the capabilities of the computer system available to you.

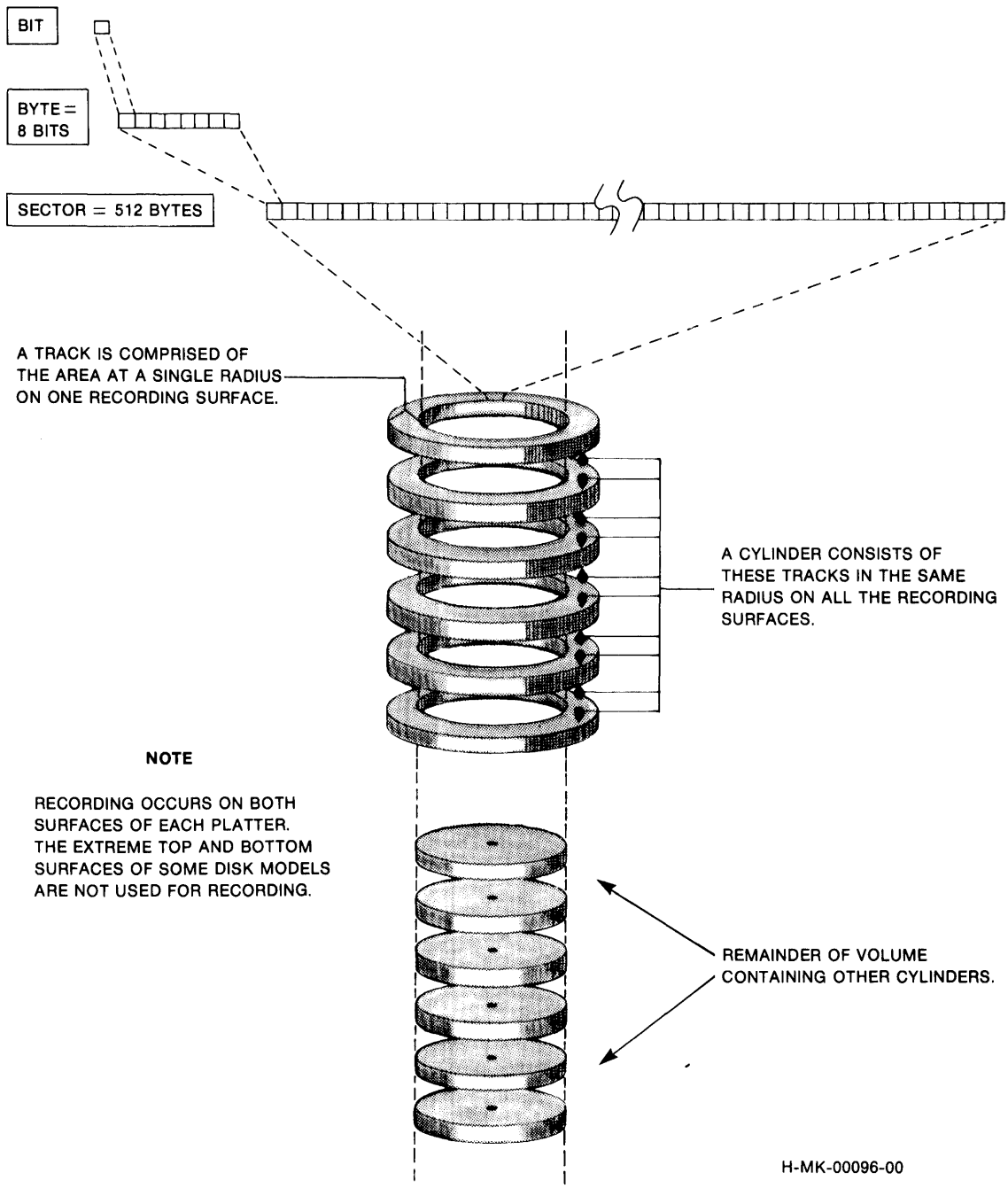
The following operating system components are involved in the data management process:

Device Drivers
: Each *device driver* is software written for a specific type of hardware unit. Drivers instruct their devices during data access operations.

File Processor
: The *file processor* maintains the structure and integrity of data stored on file-structured devices. It provides volume directory and space management functions, as well as translating RMS–11 data requests for the device drivers.

Monitor
: The *monitor* coordinates the other components of the operating system, including the device drivers and the file processor.

Figure 1-11: Physical Storage Structure



CONVENTION

The cover term *monitor* in this manual has the same meaning as the following system-specific terms:

System	Term
IAS	executive
RSTS/E	monitor
RSX-11M	executive

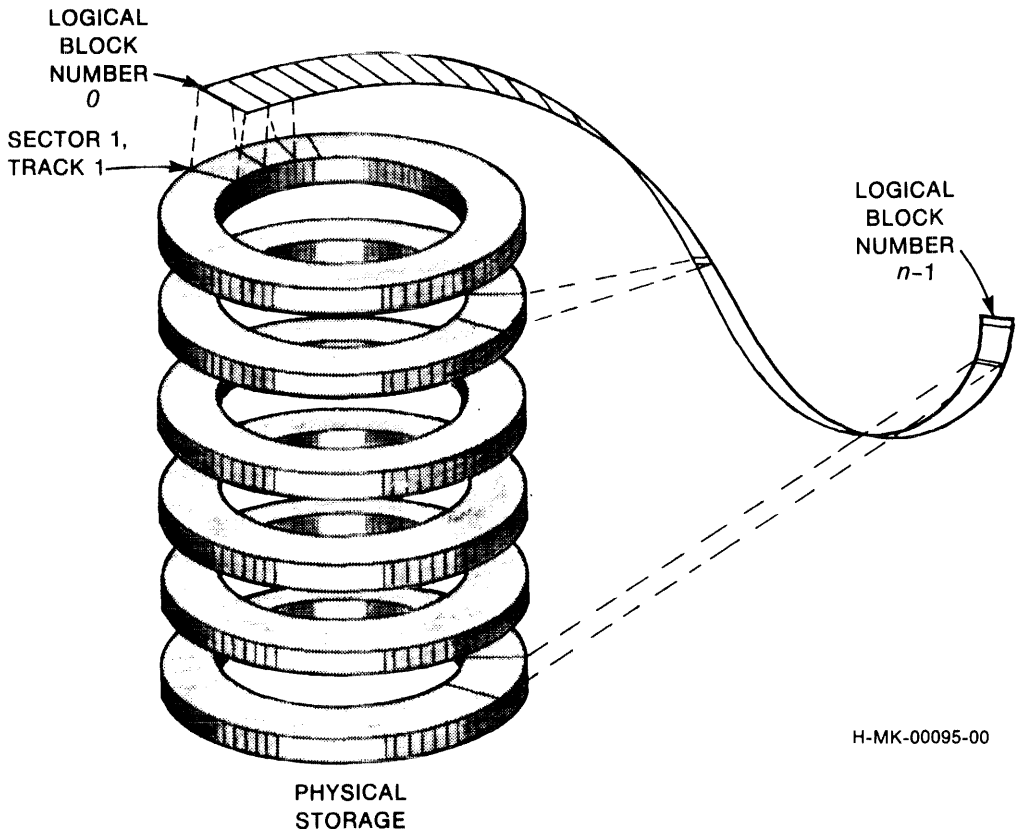
Disks As Logical Devices — To be independent of the disk drivers, the file processor places a logical structure over the data on each disk (see Figure 1-12). In essence, the processor treats a disk as a logically contiguous series of data units called *blocks*. A block contains 512 eight-bit bytes. Logical blocks are numbered from 0 to *n-1*, where *n* is the number of blocks on the disk.

NOTE

Three On-Disk Structures (ODS) are currently supported by the DIGITAL operating systems on which RMS-11 operates:

- RSTS/E has its own disk structure.
- IAS and RSX-11M use a standard named *Files-11 ODS-1*.
- VAX/VMS supports both *Files-11 ODS-1* and *Files-11 ODS-2*.

Figure 1-12: Logical Data Structure



Files As Virtual Devices — The file processor supplies containers for blocks of data. Because these logical structures serve the same purpose as paper files, they are also called *files*.

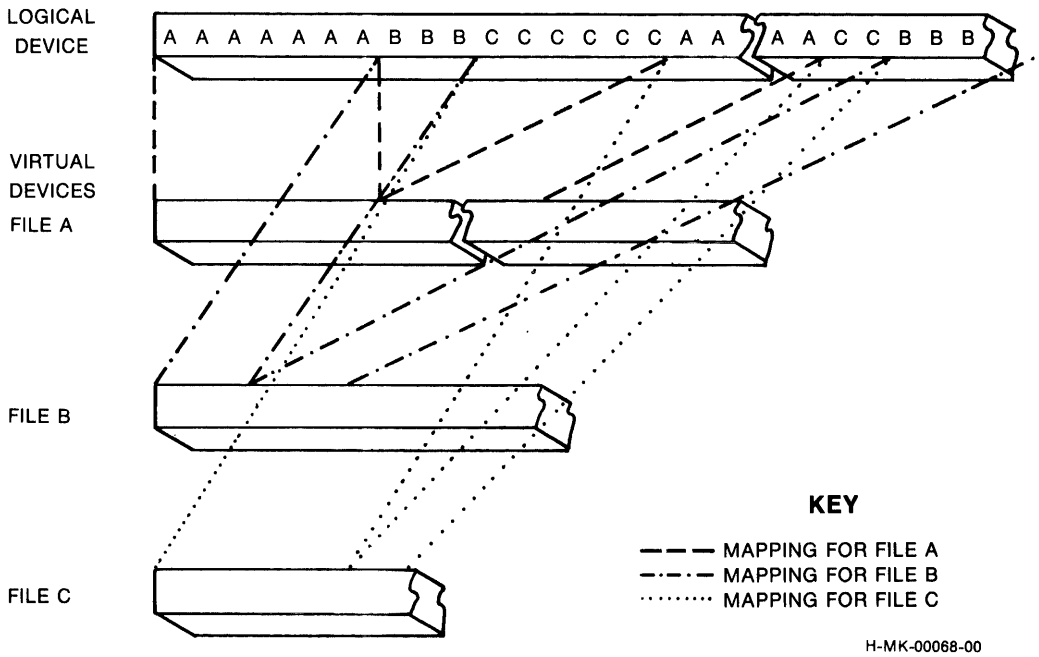
The processor treats each file as a device containing virtually contiguous blocks: it can ignore all blocks on the disk except those in the file being processed. The processor assigns *Virtual Block Numbers (VBNs)* from 1 to *n*, where *n* is the last block in the file.

NOTE

Logical block and *virtual block* describe the same physical unit of storage; only the numbering scheme is different. Virtual blocks have *Logical Block Numbers (LBNs)*, but logical blocks do not have Virtual Block Numbers unless they are allocated to a file.

Virtually contiguous does not necessarily mean logically contiguous. There is more than one file on a disk. As these files take room on the disk, there are fewer contiguous logical blocks. Eventually the file processor creates or extends a file so that portions of it reside in different parts of the disk (see Figure 1–13). The blocks retain their serial Virtual Block Numbers, but they are no longer logically contiguous.

Figure 1–13: Virtual-to-Logical-Block Mapping



Virtual to Logical to Physical Blocks — The device-independent file processor controls the virtual and logical structures applied to data and translates from one to the other. For instance, if a system user requests access to a block within a file, the file processor calculates the logical block on the disk that equates to that Virtual Block Number. In doing so, the file processor takes into account the fact that the virtual blocks may not be logically contiguous.

After calculating the Logical Block Number, the file processor requests that block from the disk driver for the device containing the file. The driver then translates that request into the cylinder/track/sector, or *physical block*, location that the device hardware must read or write.

1.2.3 The RMS-11 Interface

The file handling components of the operating system do not handle records as such. Your business is structured around the use of records within files. RMS-11 manages records, translating your requirements for the operating system and vice versa.

RMS-11 does this by controlling the internal structure of the files supported by the operating system with:

- Record formats
- File organizations
- Record access modes
- Record operations

NOTE

The following discussion is just an introduction. Details on most topics are provided in the file organization-specific chapters later in this manual.

1.2.3.1 Record Formats — To meet the requirements for record formatting discussed previously in this chapter, RMS-11 provides the following record formats:

Fixed
: Every record must have the same length.

Variable
: Each record can have a different length, but it may not exceed a maximum record size you set for the file.

Variable-with-Fixed-Control (VFC)
: Each record has a section that is always the same length and a section that can vary in length. However, no record may exceed the maximum size you set for the file.

RMS-11 also supports the following record formats to be compatible with other PDP-11 file systems (see also Appendix A):

Stream
: Each record is a contiguous series of characters, with no maximum length set.

Undefined
: Essentially no records are defined in the file. Each access operation reads or writes a block.

You control the amount, content, and arrangement of data within records and its entry and interpretation.

1.2.3.2 File Organizations — RMS-11 provides three methods of organizing records within a file:

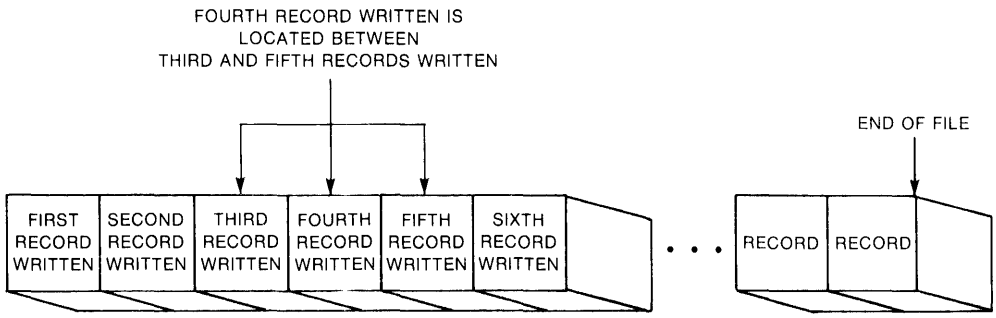
- Sequential
- Relative
- Indexed

Sequential File Organization — Figure 1-14 shows how RMS-11 implements the sequentially organized file described in Section 1.1. The organization is defined as follows:

An RMS-11 *Sequential file* is a series of virtually contiguous records stored in the order they were written.

Therefore, Sequential files can be used on any medium recognized by RMS-11 (tape, disk, or unit record devices).

Figure 1-14: Sequential File Organization



Relative File Organization — Figure 1-15 illustrates how RMS-11 implements the Relative file concepts discussed in Section 1.1. The organization is defined as follows:

An RMS-11 *Relative file* is a series of record storage *cells* with a fixed size.

- 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.
- RMS-11 stores records in the cells and associates them with their cell numbers.

Example Record number 1 is in the first cell.

Example Record number 17 is in the seventeenth cell.

Only one record can be put into a cell, but all cells do not have to contain records.

You can use cell numbers to identify and access the records in the cells. The cell numbers are then known as *relative record numbers*.

Relative files have two capabilities not available with Sequential files:

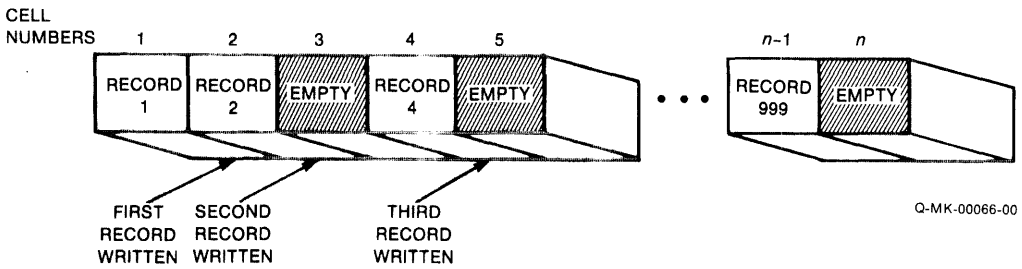
- Random access by record number
- Record deletion

With Relative files, you still have fast sequential access.

NOTE

You can store Relative files only on disks.

Figure 1-15: Relative File Organization

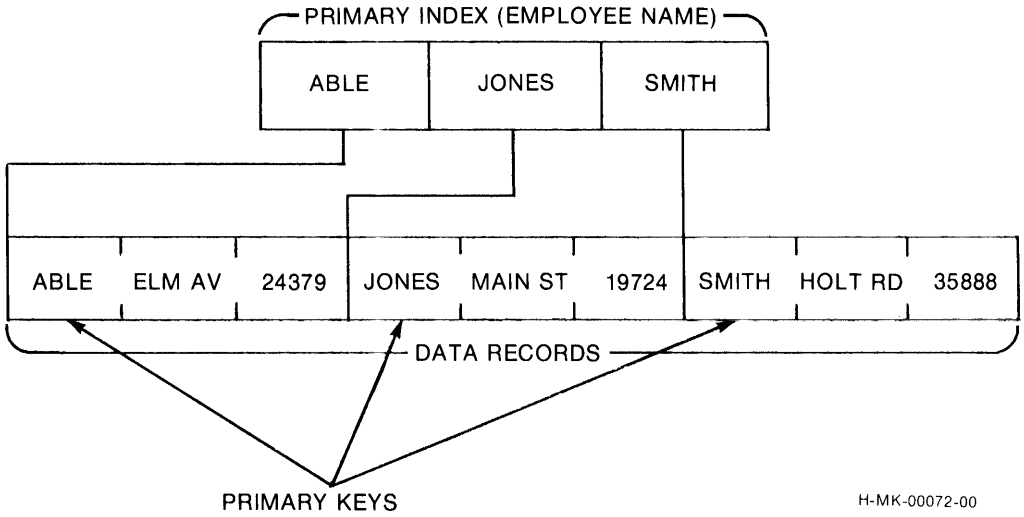


Indexed File Organization — Figure 1-16 shows how RMS-11 implements the Indexed file described in Section 1.1. The organization is defined as follows:

An RMS-11 *Indexed file* contains data records sorted in ascending order by Primary Key value and one or more indexes that point into the data records.

RMS-11 stores each record in an Indexed file according to the value of the data in a part of the record itself. Specifically, when you create an Indexed file, you must identify a section of each record as a *Primary Key*. Thereafter, when you store a record in the file, RMS-11 puts it between a record with a lower or equal Primary Key value and a record with a higher key value (see Figure 1-16). You can also identify sections of the records as *Alternate Keys*, but their values do not affect the placement of the records in the file.

Figure 1-16: Primary Index in RMS-11 Indexed File



Each key provides a logical access path to locate a record within a file. You can specify up to 255 keys for an Indexed file. For each one, RMS-11 constructs an index in the file. This structure embodies an exact and economical search pattern for RMS-11, enabling it to locate any record rapidly.

Because RMS-11 stores records in ascending order, you have fast sequential access to them.

NOTE

You can store Indexed files only on disks.

1.2.3.3 Record Access Modes — RMS-11 provides three *Record Access Modes:*

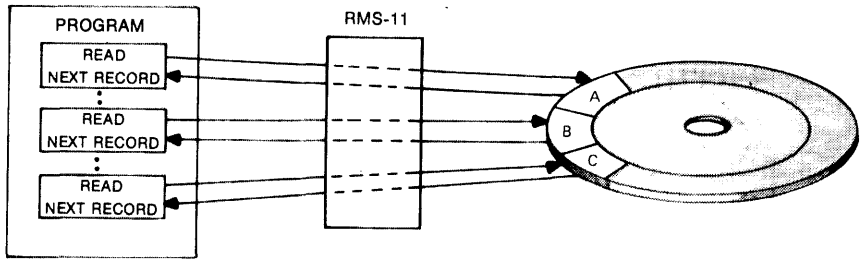
- Sequential Access Mode
- Random Access Mode
- Access by Record's File Address (RFA)

RMS-11 guarantees that every unit of data it retrieves is a record you (or others with access to the file) put into it.

1.2.3.3.1 Sequential Access Mode — Record access starts at some point in the file and continues with consecutive records. The location of the next sequential record is determined by the file organization.

Sequential Access to Sequential Files — Records in a Sequential file are virtually adjacent. To retrieve a specific record using sequential access, you must open the file and look through the records before the one you want (see Figure 1-17). From that point, you can still get any record after the one you just looked at, but if you want to retrieve a record before it, you must go back to the beginning of the file.

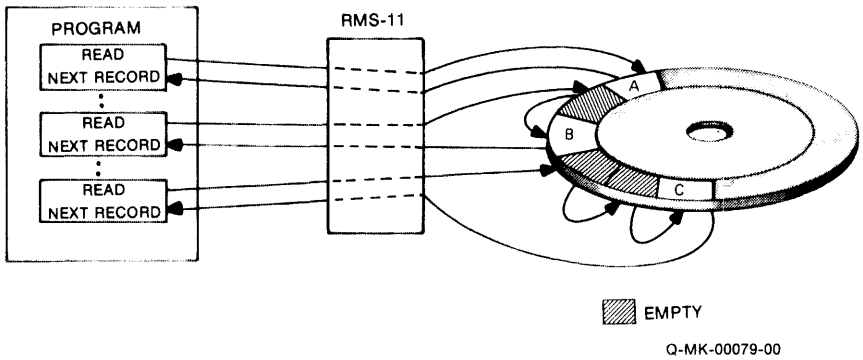
Figure 1-17: Program Sequentially Reading a Sequential File



To insert records, you must go to the end of the file and add them one at a time.

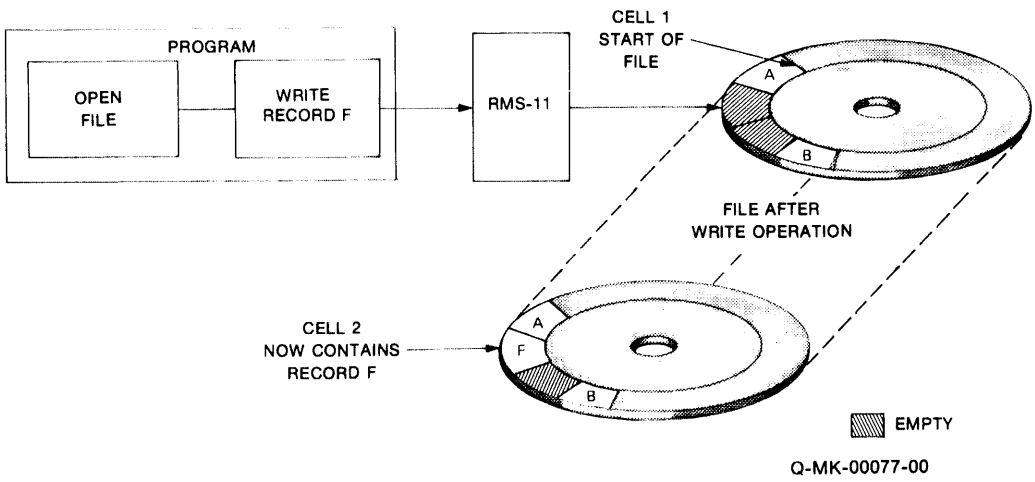
Sequential Access to Relative Files — Records in a Relative file are not necessarily adjacent. Their sequence is established by the relative record number of the cells where they are stored. To retrieve a specific record using sequential access (see Figure 1-18), you must scan the records until you get to the one you want; note that the bypassed records have smaller relative record numbers. RMS-11 skips any cells that are logically empty, returning only valid records.

Figure 1-18: Program Sequentially Reading a Relative File



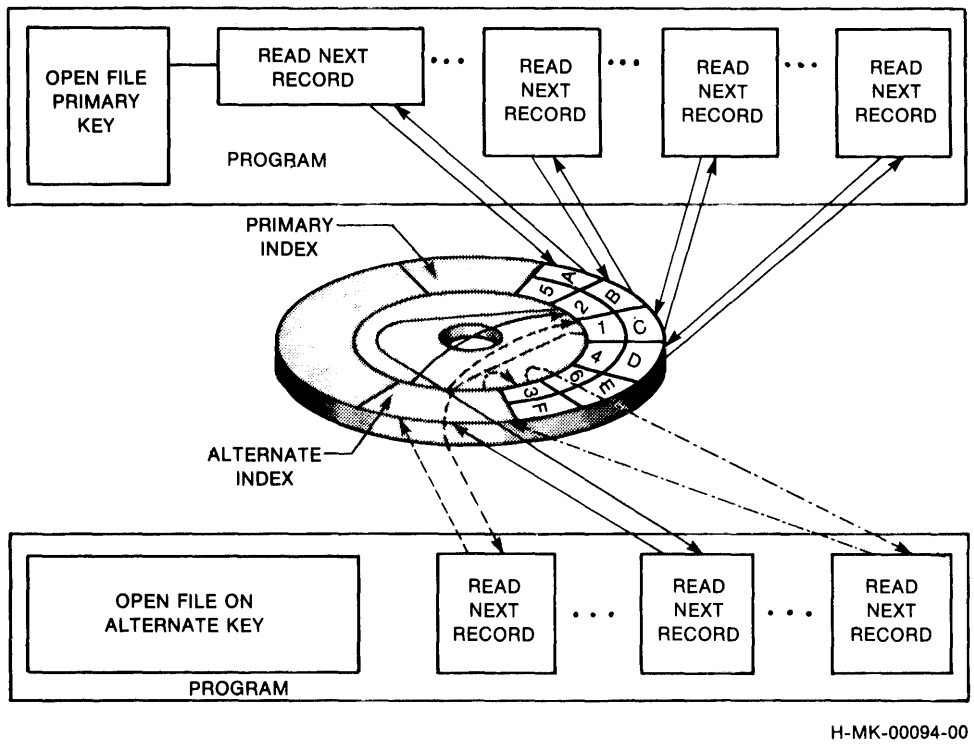
When you sequentially insert a record into a Relative file (see Figure 1-19), RMS-11 puts the record in the next cell (relative number one higher than the current cell accessed)—as long as it's empty. If the next cell contains a record, RMS-11 returns an error.

Figure 1-19: Program Sequentially Writing to a Relative File



Sequential Access to Indexed Files — Records in an Indexed file are logically adjacent, but the sequence depends on the key used for access. To retrieve a specific record using sequential access (see Figure 1-20), you must indicate a key to establish the access sequence and bypass the records with lesser key values. RMS-11 uses the specified index to locate the records.

Figure 1-20: Program Sequentially Reading an Indexed File



H-MK-00094-00

When you sequentially insert records into an Indexed file, they must be ordered in nondescending sequence by Primary Key. RMS-11 inserts the records into the file based on those values, using the same procedure it uses for random insertion (see *Random Access to Indexed Files*, Section 1.2.3.3.2).

1.2.3.3.2 Random Access Mode — You, rather than the organization of the file, establish the order in which records are processed. You must specify a record identifier with each random access. Each record access is independent of the previous record used. Successive operations in the random mode can identify and access records anywhere in the file.

You cannot use Random Access Mode with Sequential files. Both the Relative and Indexed file organizations permit random access to records.

Random Access to Relative Files — You can access a record in a Relative file by specifying (see Figure 1-21):

- a relative record number (RRN)
- whether you want:
  - only the record with the specified RRN: equal match on the RRN

- the first record after the specified RRN: greater than match on the RRN
- the record with the specified RRN or if that cell is empty, the first record after the specified RRN: greater than or equal match on RRN

RMS-11 locates a file cell according to this criteria and checks for a valid record:

- Read a record

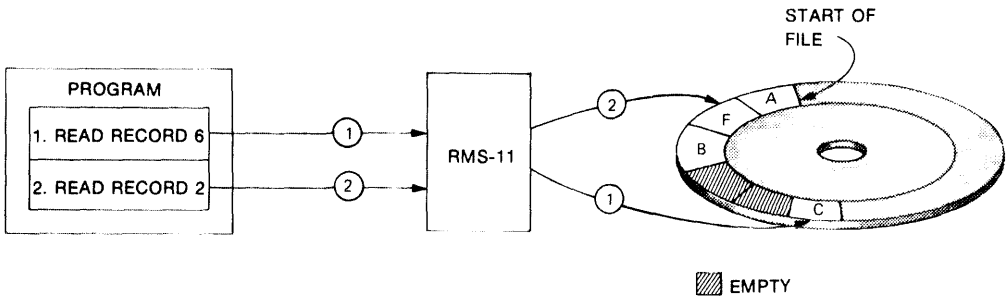
  If there is a valid record in the cell, RMS-11 returns it to you; otherwise, RMS-11 gives you an error code.

- Writing a record

  If there is a valid record in the cell, RMS-11 writes over it only in special circumstances; otherwise, RMS-11 gives you an error code.

  If there is no valid record in the cell, RMS-11 writes the record into the cell; from then on, you randomly read that record with the relative record number.

Figure 1-21: Program Randomly Reading a Relative File



The circled numbers indicate the order of the access operations; in this case, record number 2 is "F".

Random Access to Indexed Files — You can read a record in an Indexed file by identifying it as follows:

- a key number (Primary, First Alternate, and so on)
- a value
- value match (equal to, greater than, or either)
- number of characters for value match

The key number determines the index RMS-11 follows. The number also indicates the section, or *key field*, of the records RMS-11 must compare to the specified value for the length indicated by the match criteria.

Example You have an Indexed personnel file; the Second Alternate Key is the social security number. You can read the first record that contains a social security number starting with "560" by identifying it as follows:

- key number 2
- value in key field must be equal to "560" for first three characters

Figure 1-22 shows the search RMS-11 makes during this read operation.

However, to insert a record randomly, you do not have to indicate a key. RMS-11 uses the Primary Key value in the record to place the record in the file. Then, RMS-11 revises the index(es) if necessary. The example for Indexed File Organization in Section 1.1, shows how RMS-11 determines record location for all types of record access.

1.2.3.3.3 Access by Record's File Address — RMS-11 establishes a unique identifier within a disk file for every record it writes. This *Record's File Address (RFA)* remains valid for that record alone for the life of the file. If the record is deleted, its RFA is not reused; an attempt to read that record returns the information that the record was deleted.

RFA access is the fastest way to find or read a record randomly:

- For Sequential files, it is the only way to access records randomly.
- For Relative files, it bypasses relative record number processing.
- For Indexed files, it eliminates reading the index.

To read a record by RFA, specify the address for that record (see Figure 1-23). You cannot use RFA access to write a record.

1.2.3.3.4 Changing Record Access Modes — You can change Record Access Mode at any time while you are accessing a file. The file organization and storage medium must support the access mode selected. Generally, you use Random Access Mode or Access by RFA to access the first record of a series and then use Sequential Access Mode to access the records in that series.

Example A personnel file has department code as one of its Alternate Keys. You can produce a report on a department by randomly accessing the first record in the file with a specific departmental code and then using Sequential Access Mode to read consecutive records.

Example The Sequential file in Figure 1-23 is written in account-number order, one record per transaction; each account has more than one record. To list the transactions for account C, open the file and sequentially read each record until you find the first one for account C. Then you start the report.

However, if you had saved the RFA for account C's first record when you wrote it, you could access that record with its RFA and then switch to Sequential Access Mode to produce the list.

Figure 1-22: Program Randomly Reading an Indexed File

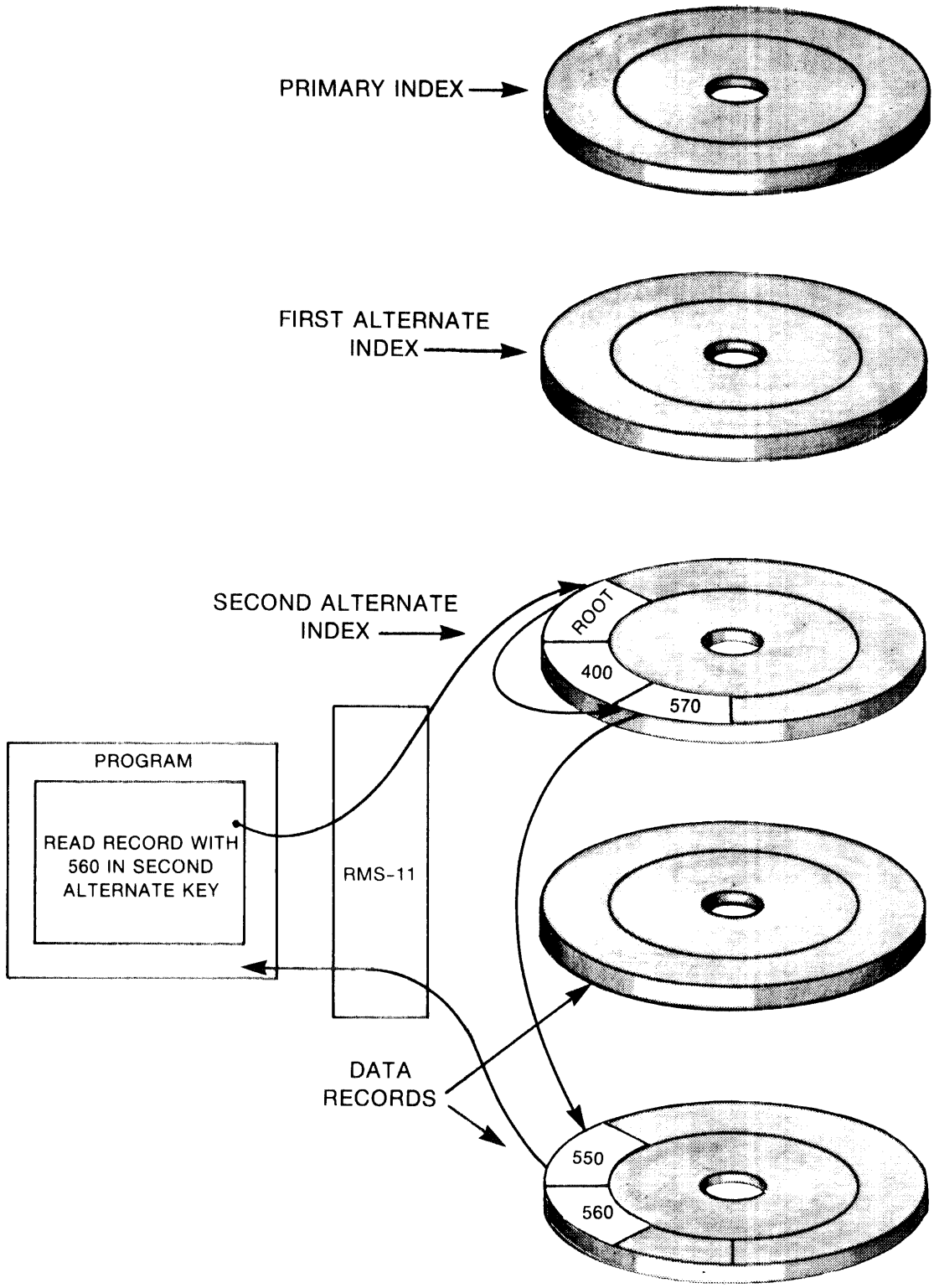
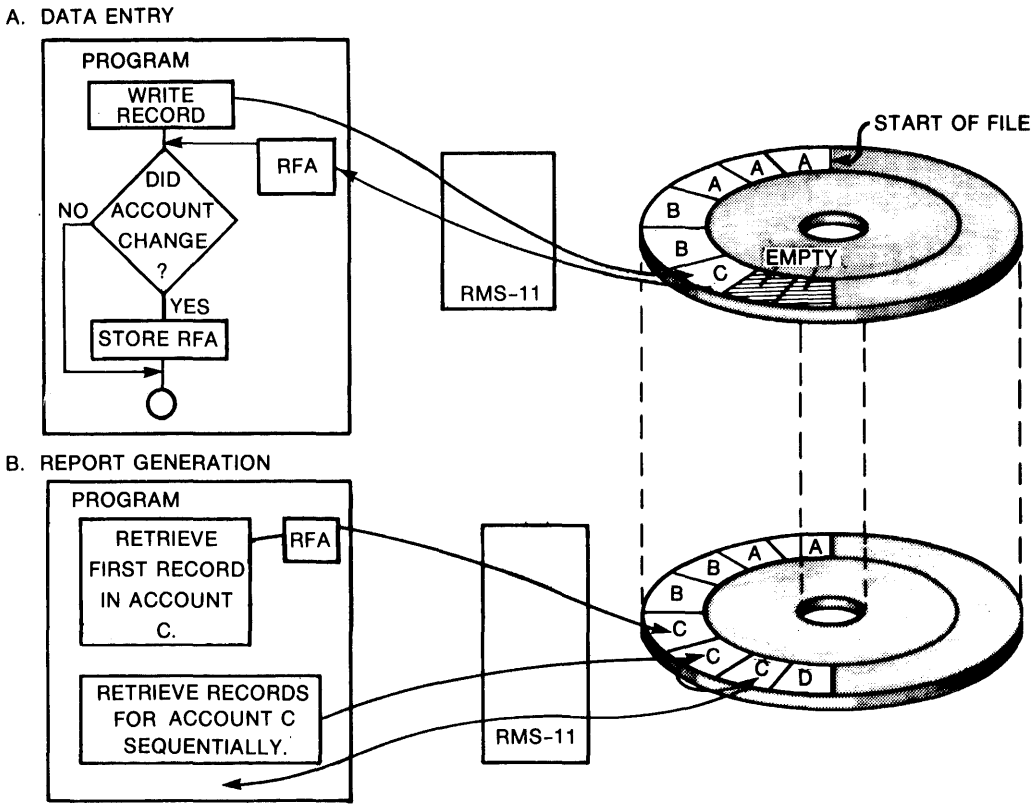


Figure 1-23: Sequential Account File



1.2.3.4 Record Operations — You write programs to process the data units you designate as records. Processing involves record operations that your program initiates and RMS-11 performs. Through RMS-11, your programs can:

- read a record, retrieving data from disk. These operations include:

  Find
  : RMS-11 locates and retrieves the specified record according to the requirements of the Record Access Mode. However, RMS-11 does not make the record available to your program.

  Get
  : RMS-11 locates and retrieves the specified record according to the requirements of the Record Access Mode. Then RMS-11 makes the record available to your program.

- write a record, storing data on disk. These operations include:

  Delete
  : You mark a record in a file, indicating that the record no longer exists. The space used by the record can be reclaimed by future operations according to the requirements of the file organization and the record format.

NOTE

You can delete records at the end of a Sequential file only, by truncating the file (described in "Truncate," Section 3.3.1.8).

Update
: You replace a record in a file with a revised version.

Put
: You store a new record in a file, according to the Record Access Mode.

NOTE

RMS-11 requires that a successful find or get operation precede a delete or update operation. However, some higher level languages hide this prerequisite.

Example Using PDP-11 COBOL, you can use a REWRITE statement without establishing the record being updated with a READ or START statement.

- perform other operations, including:

  Connect
  : You make the records of the file available to your program in preparation for a stream of operations. This operation is hidden in most higher level languages.

  Disconnect
  : You terminate a stream of operations, making the buffers assigned to the stream available for other operations.

  Flush
  : You ensure that all records you have written, updated, or deleted are written to disk before you terminate or change processing.

  Rewind
  : You return to the beginning of the file for sequential access.

"Buffers in Record Operations," Section 1.2.4.2, contains details of the processes RMS-11 uses during these operations.

1.2.3.5 RMS-11 Utilities — DIGITAL provides you with programs, called *utilities*, that use RMS-11 to accomplish standard file-related jobs. Aligned with the requirements discussed in Section 1.1, these utilities are:

RMSBCK
: Used to back up your data, the *RMSBCK* utility copies files in a special format that cannot be mistaken for the original data and therefore cannot be altered during normal operations.

RMSRST
Used to read RMSBCK's special format, the *RMSRST* utility replaces your data files with back-up versions whenever you want.

RMSDSP
The *RMSDSP* utility lists files and their attributes at your request.

RMSCNV
Used for any combination of RMS-11 file organizations and record formats, the *RMSCNV* utility copies one file into another, while preserving the source file.

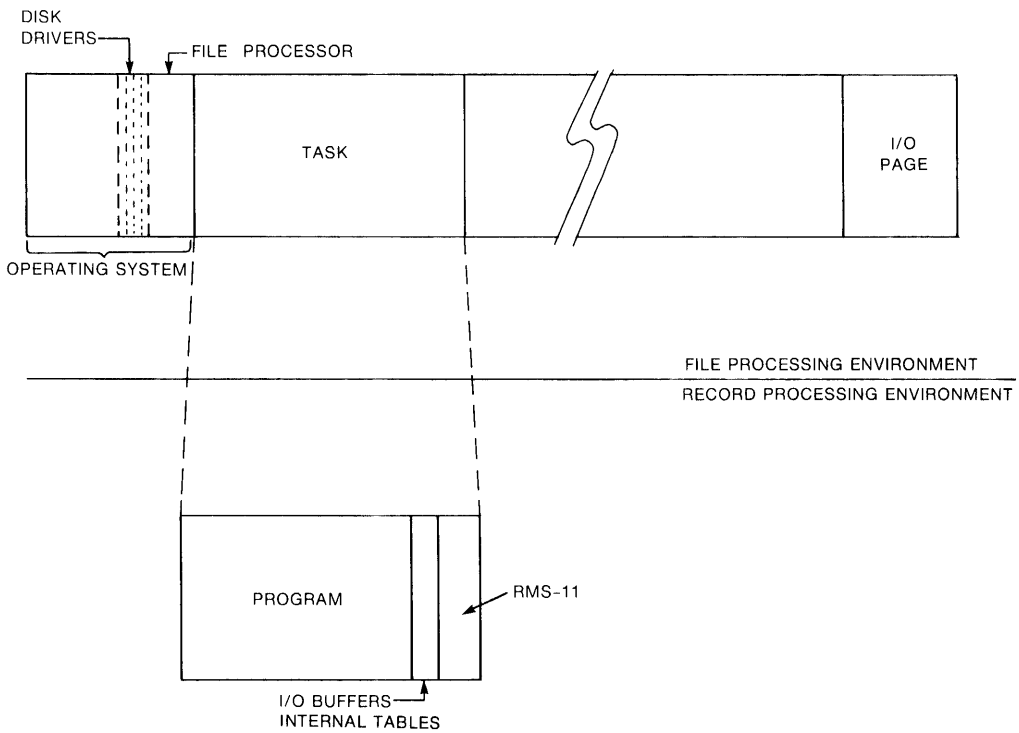
RMSDEF
During an interactive process, the *RMSDEF* utility helps you define attributes for a file and then creates it.

RMSIFL
Bypassing normal RMS-11 methods, the *RMSIFL* utility quickly loads an RMS-11 Indexed file with records from a file you designate, optimizing the structure of all indexes.

1.2.4 Record Processing Environment

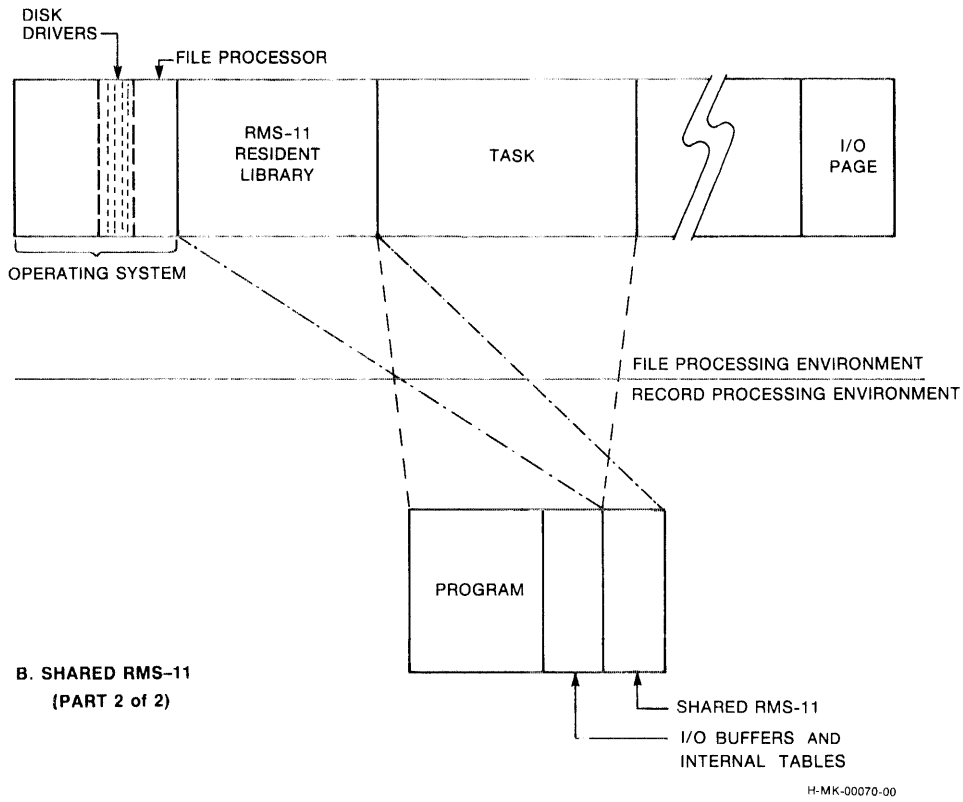
RMS-11 processes records at the command of your program, operating in virtual conjunction with its executable form. In fact, to the operating system, RMS-11 is part of your program (see Figure 1-24).

Figure 1-24: System Memory Layout



(continued on next page)

Figure 1-24: System Memory Layout (Cont.)



1.2.4.1 Using RMS-11 — RMS-11 is a set of file access routines. These routines implement a standard file structure and interface across DIGITAL operating systems and programming languages.

You use these routines by combining them with a program you have written in a language that implements RMS-11 (see Appendix B). You must write this program so that it uses the appropriate RMS-11 functions, obeying the language syntax. Then you convert your program to *object code*, through either a *compiler* or an *assembler*.

Once your program is in object form, combine it with the RMS-11 routines via a utility called the *Task Builder*. This software converts object modules to an executable form called a *task*. In the process, the Task Builder not only combines different object modules, but can 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:

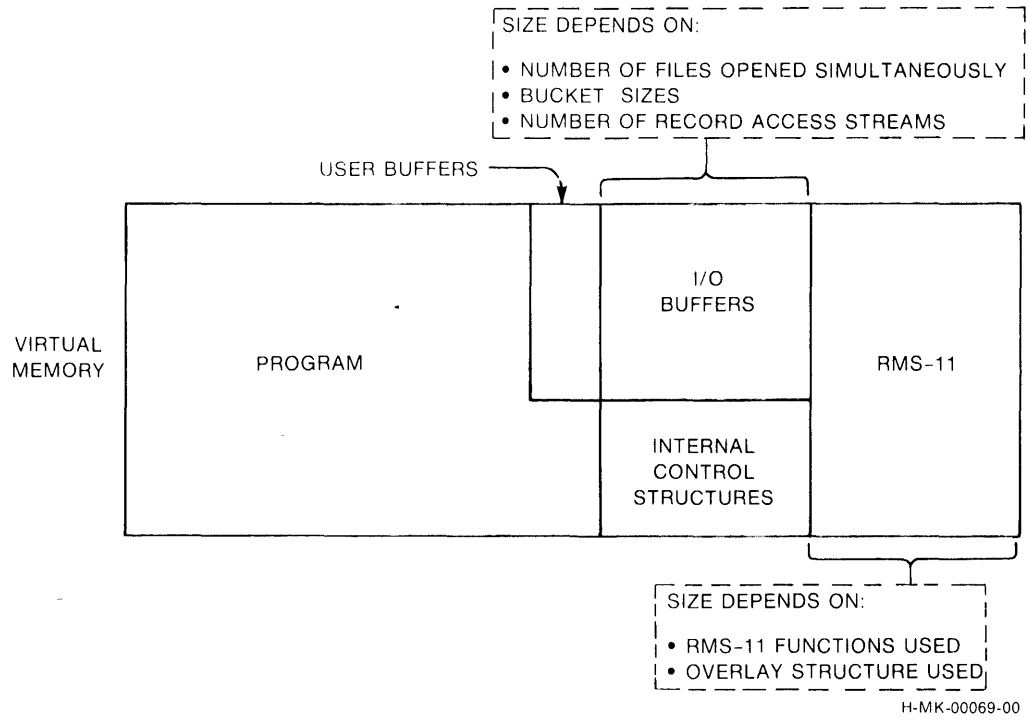
- in the task itself—with *nonoverlaid* routines or a *disk-resident overlay structure*
- in *memory-resident overlays*—a form apart from your task

The primary difference between the techniques is that memory-resident overlays can be shared among programs; the other forms cannot, that is, each program has its own copy of the routines. In addition, memory-resident over-

lays eliminate the I/O operations needed to bring disk-resident overlays from disk; therefore, your tasks run significantly faster.

Either way, your task takes the logical form shown in Figure 1-25. Your program code exists in one part of the task. The RMS-11 routines run in another part. When your program performs an RMS-11 operation, it sets up the parameters and data and then calls the RMS-11 routine. Control jumps to that part of the task, the routine runs to completion, and control returns to your program.

Figure 1-25: RMS-11 Task Structure



Also part of the task are storage buffers that usually come in three forms:

User Buffers
: Your program usually has room to store one record for each open file. This buffer is available to your program and the data in it can be manipulated, read, changed, used for calculations, and so on. RMS-11 can also access this buffer.

I/O Buffers
: For each file your program has open, RMS-11 requires at least one *I/O buffer*. All data meant for or arriving from disk is stored here:

- RMS-11 requests the file processor to move block(s) from disk into this buffer to satisfy its or your program's requirements. Each request from a file always specifies the same number of blocks, termed collectively an *I/O unit*. The size of the I/O unit depends on the file organization and your file design.
- RMS-11 moves records between the I/O buffer and the user buffer. Your program can also access this buffer in restricted circumstances.

Control Structures

RMS-11 control structures communicate with your program and with each other.

1.2.4.2 I/O Buffers in Record Operations — These buffers are used in record operations in the following ways:

Delete

Your program indicates the record to be deleted. RMS-11 has the record's I/O unit in memory because a delete operation must be preceded by a successful get or find operation. RMS-11 then changes the record in the I/O buffer to indicate that the record is deleted and requests the file processor to rewrite the I/O unit on disk. Finally, signalling success, RMS-11 returns control to the program.

NOTE

The space in the deleted record is re-used according to the requirements of the file organization and the record format.

Find

RMS-11 follows the process used for a get operation, except it does not move the record from the I/O buffer to the user buffer.

Flush

Your program tells RMS-11 to write an I/O buffer to disk if the buffer has not already been written.

Get

Your program specifies the record to be read from disk and the user buffer in memory where RMS-11 should put it. RMS-11 attempts to locate the record in the file, using techniques required by the file organization, record access mode, and the program's request. Each technique involves the movement of one or more I/O units of the file into the I/O buffer, where RMS-11 looks for the record. If RMS-11 does not find the record specified, it returns the appropriate error code.

If RMS-11 finds the record, it normally moves the record from the I/O buffer to the user buffer and signalling success, returns control to the program.

Put

Your program specifies the user buffer containing the record. RMS-11 locates the point in the file where the record belongs and has the file processor bring that I/O unit from disk into the I/O buffer. Then RMS-11 moves the record from the user buffer to its place in the I/O buffer. RMS-11 requests the file processor to rewrite the I/O unit (including the new record) to the disk and signalling success, returns control to the program.

Update

Your program specifies the user buffer containing the revised record. RMS-11 has the record's I/O unit in memory because an update operation

must be preceded by a successful get or find operation. RMS-11 then moves the data from the user buffer to the I/O buffer, writing over the old record. Finally, RMS-11 has the file processor write the I/O unit to disk and signalling success, returns control to the program.

1.2.4.3 Record Access Streams — Before your program can access records in a file, it must open that file and connect a *Record Access Stream* to it. This stream is a channel between your program and the file. You use the stream for each record operation.

NOTE

Most higher level languages do not provide Record Access Streams and the connect operation at the user level. They use these facilities to implement their own file access techniques.

Context of a Record Access Stream — Each stream processes record operations for one record at a time. RMS-11 keeps track of the stream's position in a file. This position is called a *context* and consists of the following entities (see Figure 1-26):

Current Record

The *Current Record* is:

- established by a successful find or get operation.
- the target of the following operations:

  Delete

  RMS-11 marks the Current Record as deleted.

  Get Immdediately Preceded by a Find

  Normally, RMS-11 moves the Current Record into the user buffer. Current Record does not change.

  Truncate (Sequential files only)

  RMS-11 logically deletes the Current Record and all records following it in the file by establishing a new end-of-file position at the first byte of the Current Record.

  Update

  RMS-11 replaces the Current Record with the one in the user buffer.

Since only get or find operations set the Current Record, one of these operations must precede an update or delete operation. Other operations leave the stream without a Current Record. RMS-11 rejects any update or delete operation attempted without a Current Record.

Next Record

The *Next Record* is the target of a sequential get, find, or put operation (put operations on Sequential and Relative files only).

- If you specify a get operation, RMS-11 locates the Next Record and puts it in the user buffer.

- If you specify a find operation, RMS-11 locates the Next Record.
- If you specify a put, RMS-11 moves the record in the user buffer into the file at the position of the Next Record.

Next Record is affected by record operations in specific ways explained later in this manual.

Comparable to the marker you leave in a paper file, stream context is important to sequential access only. Record operations affect context in ways designed to facilitate normal processing.

Example When you update records in a paper file, you must locate the record first. You either:

- take the form out of the file so that you can look at it (a get operation)
- verify that the form you've located is the proper one by checking the relative record number or key value (a find operation)

In this process, you are establishing the Current Record.

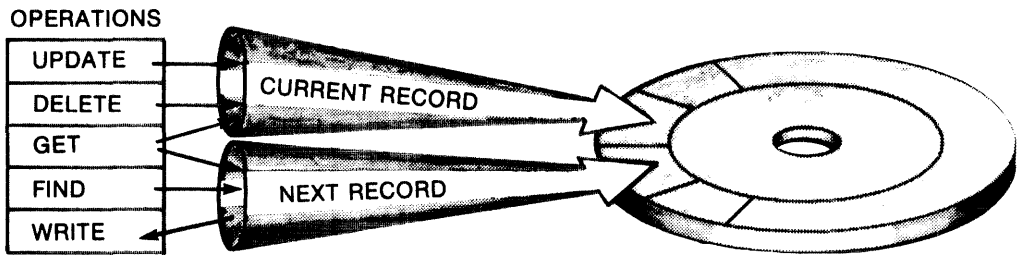
Then, when you update the record, you either change the one you've gotten or replace the one you've found. And you insert the new version where the context (Current Record) indicates.

In addition to setting Current Record with your get or find operation, you establish the position of the Next Record. Then, after you complete the update operation, the context indicates which record you locate next.

NOTE

The specific effects of each record operation on the stream context depend on file organization. Each of the organization-specific chapters describes these effects.

Figure 1-26: Record Operations and Stream Context



Q-MK-00082-00

Multiple Record Access Streams — A stream can handle only one record at a time, but you can connect more than one Record 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 (see Section 1.2.4.4)
- maintain more than one context during the processing of a file

Each stream represents an independent, concurrently active sequence of record operations. Again, most higher level languages hide this capability.

Example A program opens an Indexed file and connects two Record Access Streams. In one stream, the program uses the Primary index to access records in random mode. In the other stream, it sequentially gets records in the order specified by an Alternate index.

1.2.4.4 IAS/RSX-11M 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 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 finished. 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. See your language documentation for asynchronous techniques.

NOTE

If you intend to use asynchronous RMS-11 record operations and/or Asynchronous System Traps (ASTs) in other parts of your program, see the section on your operating system in Appendix A.

1.2.4.5 Record Transfer Modes — Your program can manipulate the data in a record while it resides in the user buffer or while it is still in the I/O buffer. These choices are called *Record Transfer Modes*. The organization-specific chapters discuss these modes more thoroughly.

1.2.5 File Processing Environment

Now that we have discussed the data management process, layer by layer, from hardware to your program, let's examine more details of RMS-11's relationship with the operating system.

RMS-11 manipulates files so that it can process records. The file processing environment involves RMS-11 with complex flows of data, control, and overlay segments (see Figure 1-27). Although its requests initiate activity in the operating system and its devices, RMS-11 is not aware of the file management process.

1.2.5.1 File Processor — Each operating system has a file processor:

- Files-11 Ancillary Control Processor (F11ACP) on IAS/RSX-11M
- File Processor (FIP) on RSTS/E

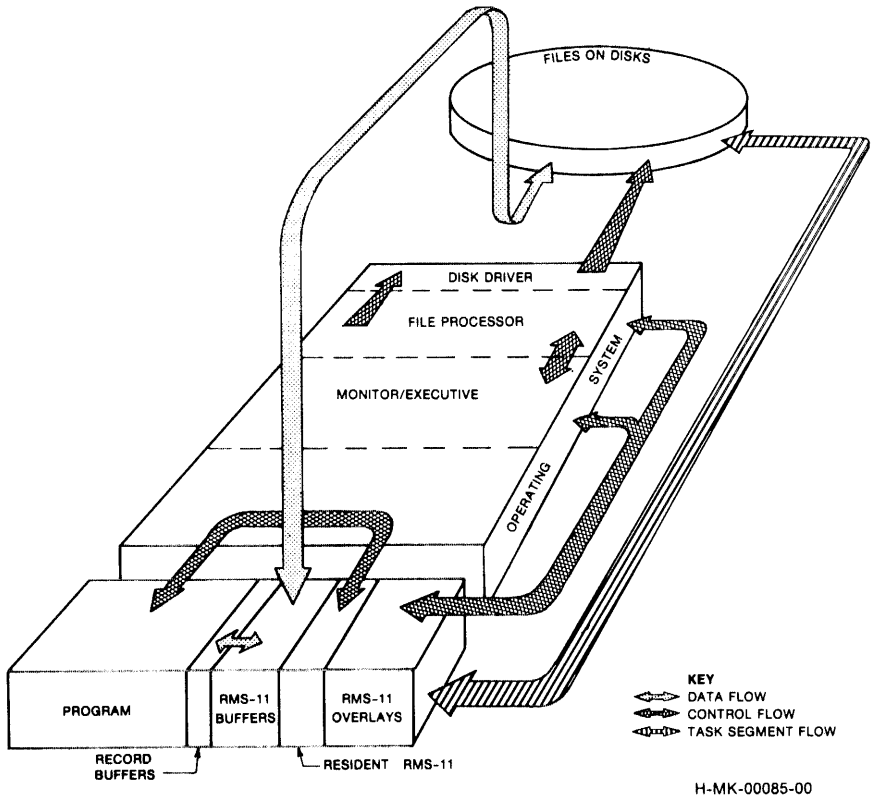
The file processor performs I/O and other operations on files. RMS-11 must make requests in a certain format so that the file processor changes the files properly. Thus, the file processor's operations, while logically invisible to RMS-11, can affect the performance of your program.

However, the file processor is not concerned with the data contents of a file. It only knows Virtual and Logical Block Numbers, directories and other support

information, and the disk drivers involved. Therefore, RMS-11 can manipulate the contents of a file as long as it makes proper requests to the file processor. In this manner, RMS-11 maintains the following file formatting or structures:

Block Spanning — Basically, RMS-11 lays out the records in a Sequential file one right after the other, in the order they are written. However, you must decide whether those records can cross block boundaries. When records span blocks, RMS-11 can pack them with optimal density into the file because a record can be stored in one or more blocks (see Figure 1-28). When block boundaries restrict records, each one must be less than 512 bytes long, and RMS-11 might leave unused bytes at the end of each block.

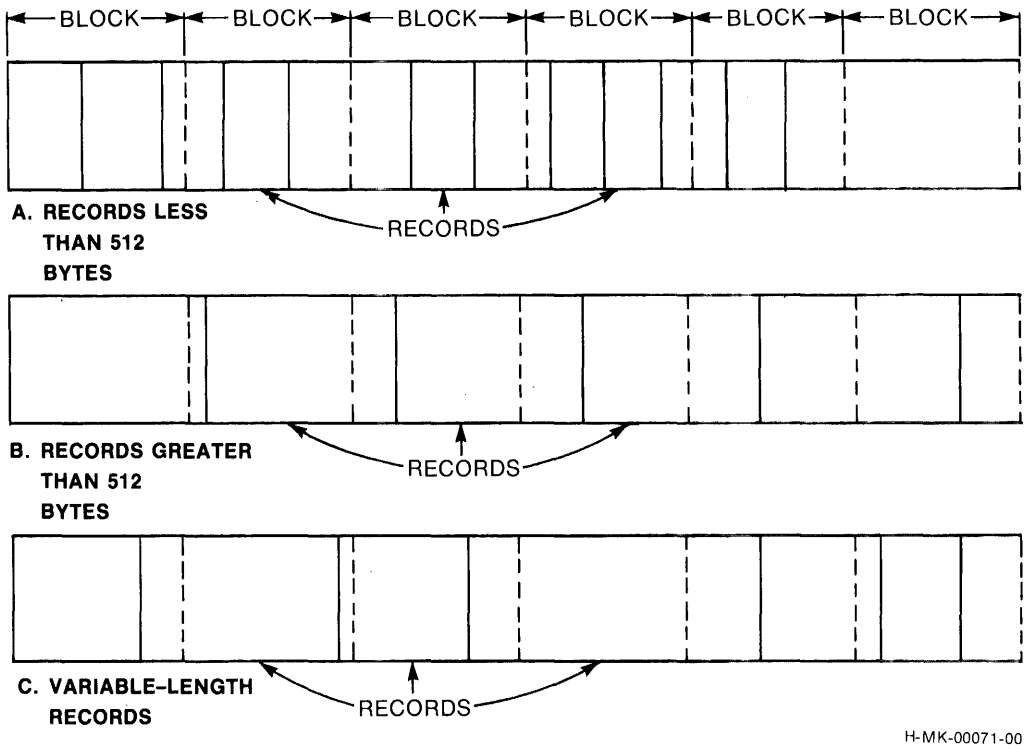
Figure 1-27: RMS-11's Environment



Buckets — The I/O unit for Relative and Indexed files is called a *bucket*. A bucket consists of one or more blocks that RMS-11 treats as a unit. Indexed files, in fact, consist of buckets formatted with control information. Records can cross block boundaries, but they cannot cross bucket boundaries.

When RMS-11 initiates an I/O operation for a file of one of these organizations, it requests the file processor to move a bucket. Since buckets are an RMS-11 concept, the request specifies 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 a file; once created, a bucket contains the same virtual blocks at all times.

Figure 1-28: Records Spanning Blocks



The operating systems limit bucket sizes:

Operating System	Maximum Bucket Size
IAS	32 blocks
RSTS/E	15 blocks
RSX-11M	32 blocks

NOTE

The I/O unit for Sequential files is not the bucket, but 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.

Areas — Maintained and used by RMS-11, *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. You do this to improve performance.

Placement Control — Through the file processor, RMS-11 allows you to place a file, as a whole or by areas, on a disk at specific location(s). You do this to improve performance, taking advantage, for example, of tracks and cylinders.

1.2.5.2 File Sharing — Timely access to critical files often requires more than one program to use those files at the same time. With the help of the file processor, RMS-11 enables programs to share files.

The way programs can share a file depends on the file organization:

- With the exception of magnetic tape files, every RMS-11 file can be shared by any number of programs for read-type operations.
- Only one program at a time can access a Sequential file for write-type operations, while multiple writing programs can share Relative and Indexed files.

File sharing is controlled by the programs and by the order that the programs open a file. Basically, the first program to open a file sets the sharing type; programs attempting to open the file after that generally must specify the same type of sharing. More details on file sharing are provided in Chapter 2.

1.2.5.3 File Operations — Although the file processor does most of the work, RMS-11 provides the following file-level functions:

Creating a File

In addition to the file specification, RMS-11 passes the following information to the file processor when it creates a file:

- An initial allocation of blocks for each area in the file. You specify both the areas and their allocations in your instructions to RMS-11.
- The specific locations on a device where the processor should allocate those blocks. You also supply this information.
- The following file attributes:

  File organization
  Record format
  Forms control
  Record size
  Number of virtual blocks in the file
  End-of-file (Sequential files only)
  Bucket size (Relative and Indexed files only)
  Default extension quantity

  RMS-11 stores the other file attributes, such as key and area descriptions for Indexed files, in the file (see "File Attributes," Section 1.2.5.4).

Opening a File

RMS-11 initiates access to the specified file, reading its attributes.

Extending a File

RMS-11 requests the file processor to add blocks to a file's allocation, in two circumstances:

- A program explicitly directs the extension. RMS-11 passes the request almost directly to the file processor, using the extension quantity supplied by the program.

- A put or update operation cannot be completed because there is not enough room in the file. RMS-11 requests that blocks be added to the file, using either the default extension quantity or if that is zero, a minimum number of blocks depending on the file organization.

Closing a File
: RMS-11 writes all I/O buffers to the file, if they haven't already been transferred, and terminates access to the file.

Erasing a File
: RMS-11 requests the file processor to delete the file from the device directory and release its blocks for re-use. You can erase a file you or other programs are accessing. However, the file processor does not actually erase the file until all accessing programs close it.

1.2.5.4 File Attributes — When you create an RMS-11 file, either through a program or an RMS-11 utility, you must specify the following information:

Medium
: Your selection depends on the file's organization. You can create permanent Sequential files on disk devices or magnetic tape volumes. You can also write transient files on devices such as line printers and terminals.

  However, RMS-11 restricts Relative and Indexed files to disk devices.

File Specification
: The name you assign to a new file enables RMS-11 to find the file later. You follow the file specification conventions for your operating system; see also Appendix A.

Protection
: RMS-11 also allows you to assign a protection code to a file when you create it; again, the format of this specification depends on the operating system.

File organization
: You have a choice of three organizations described in "File Organizations," Section 1.2.3.2: Sequential, Relative, and Indexed.

Record format
: Your choice of the record formats described in "Record Formats," Section 1.2.3.1, is restricted by the file organization (see Table 1-1):

Table 1-1: Record Formats and File Organizations

File Organization	Record Format				
	Fixed	**Variable**	**VFC***	**Stream**	**Undefined**
Sequential	Yes	Yes	Yes	disk only	Yes
Relative	Yes	Yes	Yes	No	No
Indexed	Yes	Yes	No	No	No

* Variable-with-Fixed-Control

Record size
The meaning of the record size information depends on the record format:

For fixed-length records, record size is the same length for every record in the file. RMS–11 rejects any write-type record operation that specifies a record of the wrong size.

For variable-length and stream records, record size is a maximum length. RMS–11 rejects any write-type record operation using a record size greater than the maximum—unless the maximum is zero; then RMS–11 does not check the length of records added to the file. RMS–11 also keeps the length of the longest record actually stored in the file.

For variable-with-fixed-control records, there are two size specifications:

- length of the fixed control area
- maximum length of the variable area

RMS–11 treats these specifications the way it treats the sizes for fixed-length records and variable-length records, respectively. RMS–11 also keeps the length of the longest record actually stored in the file.

Block spanning (for Sequential files)
You decide whether or not records can cross block boundaries.

Bucket size (for Relative and Indexed files)
You establish the number of blocks in each bucket. Bucket size impacts performance.

Maximum record number (for Relative files)
If you set a nonzero *Maximum Record Number (MRN)*, RMS–11 rejects any record operation using a higher relative record number. If you establish an MRN of zero, RMS–11 does not check relative record numbers.

Keys (for Indexed files)
You must decide the following:

- number of keys
- position and size of each key
- data type for each key (including string, two- and four-byte integer and binary, and packed decimal)
- whether records can duplicate key values
- whether Alternate Key values can change during update operations
- null key value for Alternate Keys

Areas (for Indexed files)
You must decide the following:

- the number of areas in the file

- what logical portions of the file go in which areas
- the fill number for each area (space and performance optimization)

Forms control

You can specify two types of forms control for records of any format in a file of any organization:

Carriage Control

When records from the file are written directly to a unit record device, the device driver puts a line feed character in front of the record and a carriage return character after the record before passing it to the device.

Example You use RMSCNV to write the records of an Indexed file to a line printer. If you have specified carriage control for that file, the records are printed on separate lines. If you have not, the records are printed continuously, the only breaks coming at the physical ends of lines.

FORTRAN

When records from the file are written directly to a unit record device, the device driver interprets the first byte of each record as a FORTRAN forms control character.

You are not required to specify either type of forms control.

Default extension quantity

You specify how many blocks you want RMS-11 to add to the file when each allocation has been completely used for data storage. RMS-11 extends the file automatically when it needs space to complete an operation.

CONVENTION

The cover term *file directory* in this manual has the same meaning as the following system-specific terms:

System	Term
IAS	directory entry and file header(s)
RSTS/E	file directory
RSX-11M	directory entry and file header(s)

During the 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 of the file (called the *Prologue*).

NOTE

Attributes also include the file's current size, in blocks. You may specify an initial allocation quantity when you create the file, but this initial size probably changes as you use the file.

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

- The file will not change its characteristics.
- You can design your RMS-11 files off-line. No program accessing the files need specify attributes (except those required by the higher level languages), because RMS-11 uses only a file specification from a program when it opens a file. The files act as virtual devices for the programs.
- You can open an RMS-11 file with only its file specification. After that, RMS-11 enables you to read the file attributes. You can write your own program or use the RMSDSP utility to display those attributes.

1.2.6 Bypassing Record Processing

Finally, your program can bypass RMS-11 record processing and process any RMS-11 file block-by-block in a mode called *Block I/O*. However, RMS-11 requires files that will be written using Block I/O to be created with the following attributes:

- disk or magnetic tape medium
- sequential organization
- undefined record format

However, you can read an RMS-11 file with Block I/O regardless of the organization or record format.

Using Block I/O, your program reads or writes multiple blocks of the file by identifying a starting Virtual Block Number and the number of blocks affected. Your program, of course, must interpret the contents of the blocks once RMS-11 retrieves them.

Chapter 2
Application Design

You're writing an application. You want a program or a set of programs to take in 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 with RMS-11 considerations in mind. These considerations include initial allocation, record format, overlays, key selection, disk usage, and others.

If you don't consider RMS-11, you won't get the best performance possible from your application, **and** you'll probably get less performance than Luck would allow because of the defaults you're accepting without knowing it (see "When To Design" in this chapter).

Example If you do not design your file, you could end up with a file like one user did:

The first time he created the file, he used a higher level language program and took all defaults. Then he loaded records into the file: the process was quite lengthy.

However, he re-examined the file and recreated it, applying a couple of design considerations. With the new file, the record insertion process went ten times faster.

Example If you do not understand the implications of RMS-11 file structure, you could end up like some users accustomed to programming with BASIC-PLUS Record I/O:

They picked up the facts that RMS-11 uses 15 bytes of control data in each bucket and seven bytes of control data for each fixed-length record (more in Chapter 6). Then, because they were used 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 seven-byte pointers (more in Chapter 5). Therefore, when one of those 490-byte records was moved, it left behind seven 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.

Example If you slap together an application with a higher level language, you probably don't worry about RMS-11. In this process, you accept the language's concept of design, if any. The chances are good that the defaults the language uses in its interface with RMS-11 are not suited for your application.

2.1 When To Design

There are two times to design an application:

- 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)
- After you write the application, if you're not happy with its performance. Often, poor performance results from default values. You can often 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:

  Source Language Compilers
  : In many instances, *source language compilers* such as PDP-11 COBOL 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.

    Specifically, PDP-11 COBOL provides the BLOCK CONTAINS clause in the file-description-entry. You can use this clause to set the number of bytes or the number of records in a bucket. However, if you do not include this clause, the PDP-11 COBOL compiler sets the bucket size to the minimum disk blocks required to contain one record.

  RMS-11
  : The intratask interface between the RMS-11 routines and your program has the same structure in all tasks, regardless of their source, PDP-11 COBOL, RPG, MACRO-11, and so on. This interface consists of control blocks (see the *RMS-11 MACRO-11 Reference Manual* 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.

  Operating System
  : RMS-11 acts as a middle man 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 primarily concerned with four things:

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

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

Shared Access
: You want your data to be exactly as accessible to the people using the computer system as necessary, no more, no less.

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 Maximize Speed First

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 (see Figure 2–1) affects I/O time:

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

  bucket size
  number of keys
  number of duplicate key values
  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.

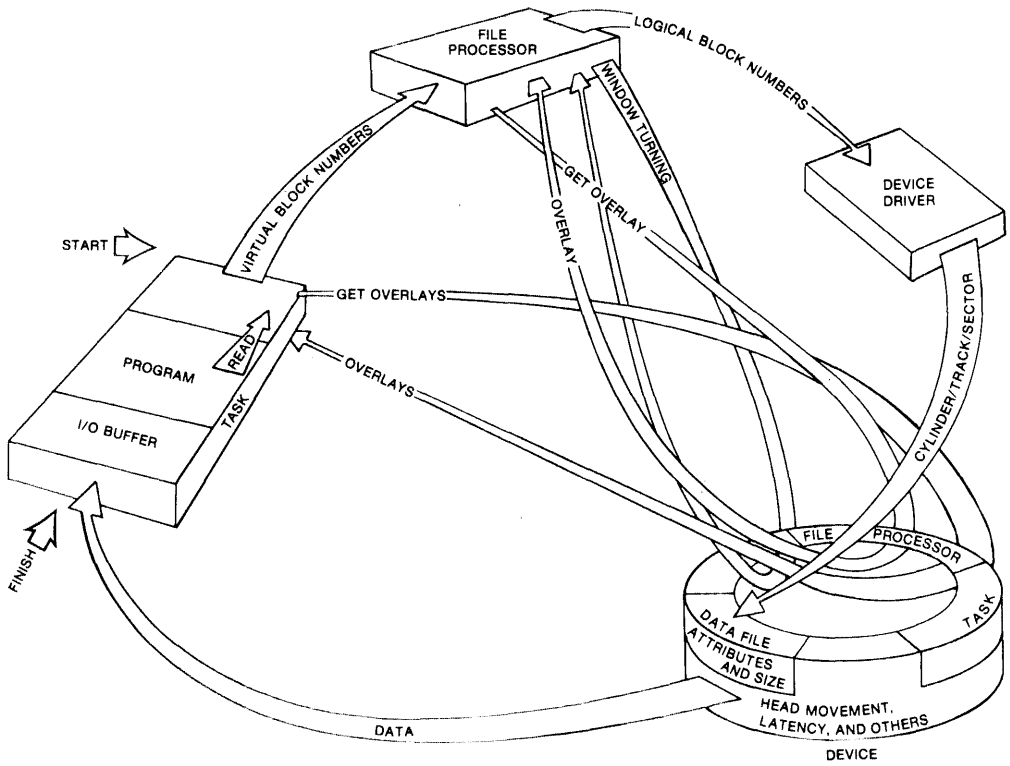
Program
: Your program impacts I/O time by requiring I/O operations for file operations (open, close, and so on), record operations (get, put, and so on), and overlays.

RMS-11
The RMS-11 routines can be overlaid.

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

Device hardware
The storage device is the primary contributor to the length of an I/O operation. The type of device chosen (moving-head, fixed-head, and so on) to contain the task and the data files is crucial to I/O performance.

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



H-MK-00084-00

2.2.2 Reduce Space Requirements

RMS-11 requires space for three reasons:

- to store data in a file
- to store the RMS-11 routines
  - on disk when they're not in use
  - in memory when they're being executed
- to buffer data in memory while the task runs

Data Storage Space — The space RMS-11 requires to store data is proportional to the organization of the file—and 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 record with a word[1] boundary. Also, when the file contains variable-length records, RMS-11 adds a record-length 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 one byte longer than the fixed size of fixed-length records or three 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
- fifteen bytes of formatting information for each bucket
- a seven-byte header for each record
- a record-length 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.

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 "Task Building with RMS-11 Routines," Section 8.1, for more details.

Buffer Sizes — You can vary the size of the I/O buffers RMS-11 uses to store data in memory (see "Using RMS-11," Section 1.2.4.1). Generally, the larger the buffers, the faster the task processes data. See "I/O Techniques," Section 3.3.1.3, 4.3.1.3, or 7.1.3, for the file organization(s) you are interested in.

2.2.3 Provide Shared Access

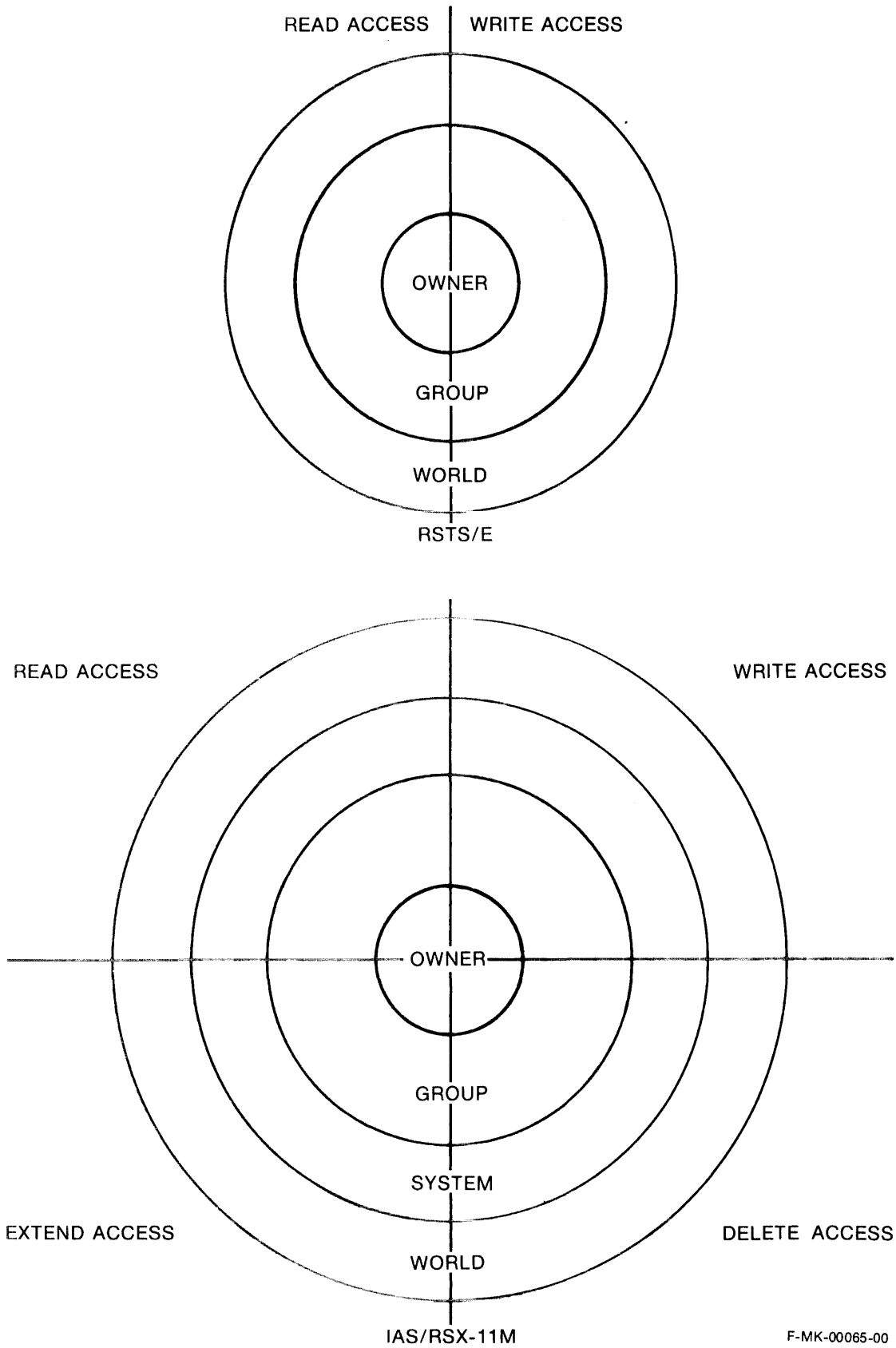
Shared access revolves around the question: who is allowed to read or write to a file? The answer involves two levels of permission to access the file:

- system protection codes
- sharing specifications in accessing programs

2.2.3.1 System 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 Figure 2-2). See your operating system documentation for specific protection conventions.

[1] A word equals two bytes.

Figure 2-2: System Protection Concepts



Before you can share an RMS-11 file, or for that matter, run the task that accesses the file, you must log into your computer system under an account number compatible with the protection code(s) assigned to the file and task.

2.2.3.2 Sharing among Programs — Once the operating system allows a program access to a file, the program's own specifications take effect. Whenever a program opens a file, it must declare:

- the record operations it intends to perform on the file (find, get, put, update, delete, and/or truncate)
- the operations it will allow other programs to perform on the file. These operations are categorized as either[2]:

  Read-type
  Other programs may access the file for get and find operations only.

  Write-type
  Other programs may access the file for put, update, delete, and truncate operations, as well as get and find operations.

Shared Access Criteria — The first program to open a file determines how other (*not-first*) programs can access that file (see also Table 2-1):

- The *access* declaration of the not-first programs must agree with the first program's *allow* declaration.

  Example The first program allows read access. Any program declaring update intentions is denied access to the file.

  Example The first program allows write access. Any other program, regardless of access declaration, is allowed to open the file—if it meets the other requirements.

- The allow declaration of the not-first programs must agree with the first program's access declaration.

  Example The first program has accessed the file for write operations. All not-first programs must allow write access.

CAUTION

On RSTS/E, if the first program has specified access write and allow no-write, other programs with access read, allow no-write declarations can still open the file. However, the reading programs are not protected against the file changes being made by the writing program; the following subsection, "Bucket Locking," discusses this topic.

- The allow declaration in the not-first programs must be the same as the allow declaration in the first program.

[2] Some higher level languages have a broader range of access options; they break down to language-specific checks, then either read-type or write-type sharing when RMS-11 opens the file.

Example If the first program allows write operations, all not-first programs must allow write operations.

NOTE

On IAS/RSX-11M, there is one exception to this criterion: if the first program declares read access, but allows writers, a not-first program with any access declaration and allowing no write access can open the file. However, from that point, programs attempting to open the file must have a no-write allow declaration.

The operating system only allows not-first programs that meet all these criteria to open the file.

CAUTION

The first program opens a file with access read, allow no-write declarations. According to the assigned system protection code:

- If the program has write-access privileges to that file, RSTS/E grants write access to the program. No other program can open the file with write access.

  Other access read, allow no-write programs open the file. Then, the first program closes the file. Write access to the file is then available to other programs.

- If the program has only read-access privileges to that file, RSTS/E does not grant write access to the program; instead, the program opens the file for read access only. Write access to the file is still available.

When the write access is available, a program declaring access write, allow no-write can open the file. Any reading programs also accessing the file are not protected against the changes caused by the writing program.

Table 2-1: Shared Access Criteria

Not-first Program Declarations	First Program Declarations			
	Access Write Allow Write	**Access Write Allow No Write**	**Access Read Allow Write**	**Access Read Allow No Write**
Access Write Allow Write	Opens file	Access denied	Opens file	Access denied
Access Write Allow No Write	Access denied	Access denied opens file	Access denied	Access denied opens file
Access Read Allow Write	Opens file	Access denied	Opens file	Access denied
Access Read Allow No Write	Access denied	Access denied opens file	Access denied	Opens file

NOTE

You cannot ensure that a program has exclusive access to a file.

Bucket Locking — The concern here is data integrity. Anyone who updates a record should be assured that the data written back to the file is good until that record is accessed again.

Conflict occurs when more than one program tries to update a file at the same time. If no control is placed on access, two or more programs could read the same record, one after the other, then update it, one after the other. Only the last update remains in the file.

Therefore, RMS-11 activates bucket locking for a Relative or Indexed file when the first program to open it allows write sharing. From that point, RMS-11 requests the operating system to lock each bucket read from disk until RMS-11 explicitly releases the bucket. Typically, after a get, find, or mass insert put[3] operation, only the bucket containing the data record remains locked. 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:

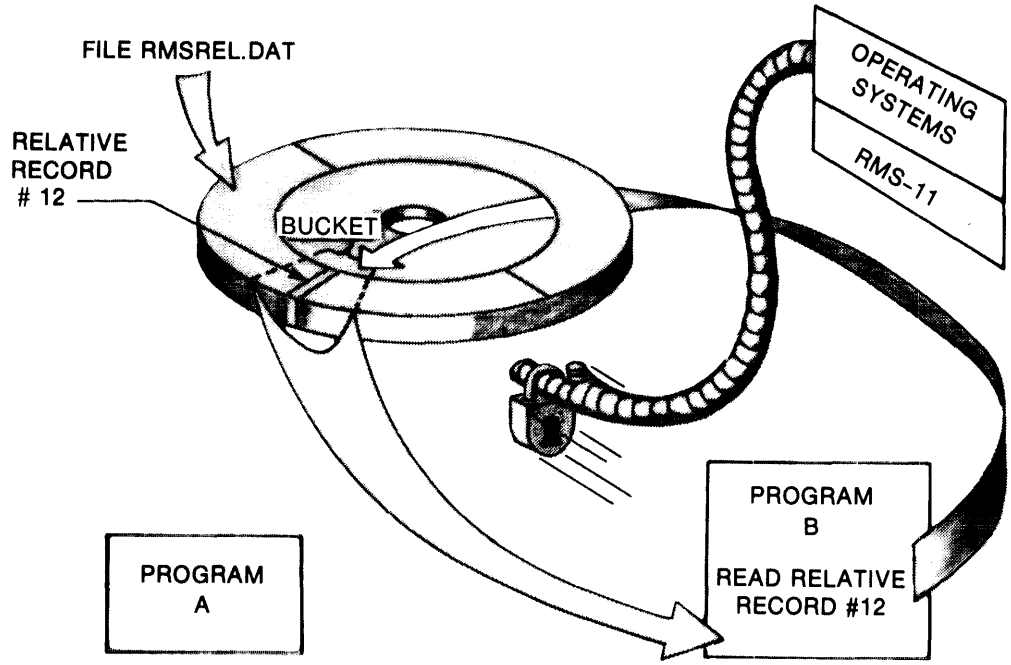
- The get, find, or put operation fails.
- The get or find operation ends successfully—if the program has declared read only access to the file.
- The program initiates another record operation that accesses a different data record bucket.

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

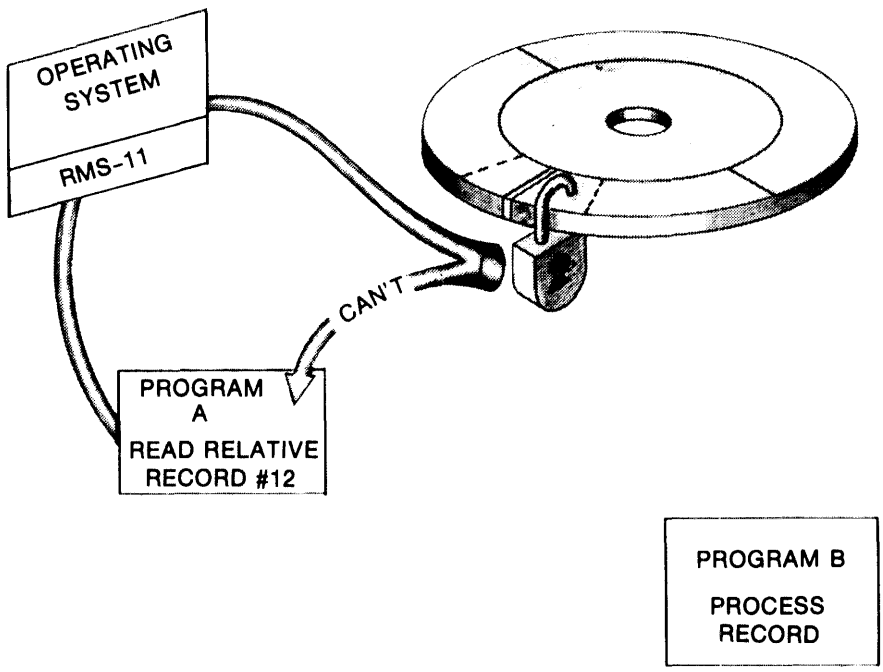
Example Programs A and B are write-sharing a file named RMSREL.DAT (see Figure 2-3). Both try to update relative record number 12. However, program B initiates the prerequisite get operation first, locking the bucket containing the record. The operating system keeps program A from accessing that bucket 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).

[3] A performance-oriented I/O technique used with Indexed files. See Chapter 7.

Figure 2-3: Bucket Locking Example



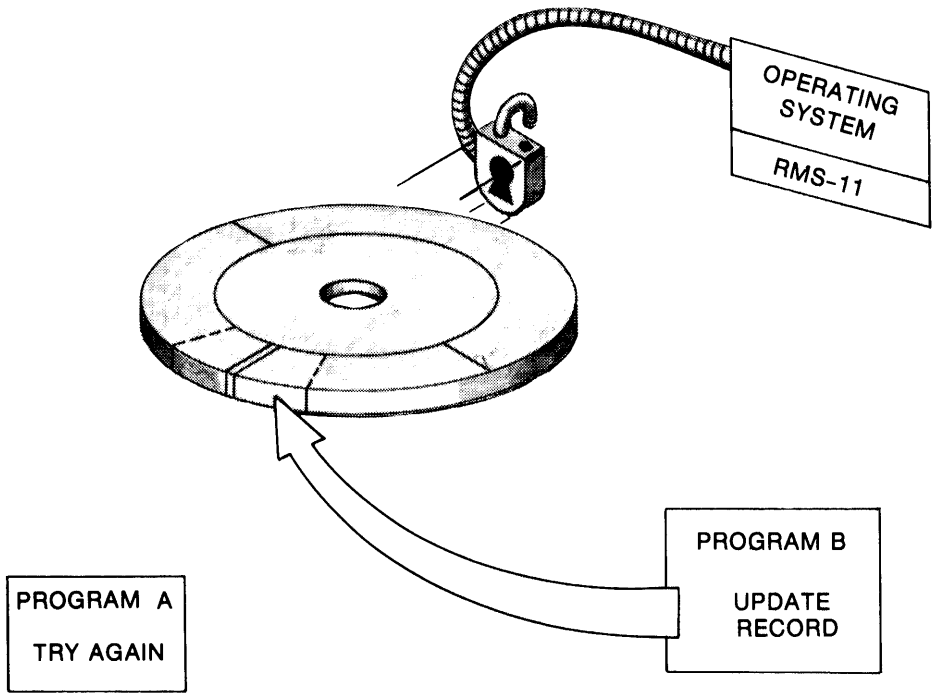
A. PROGRAM B GETS RECORD 12



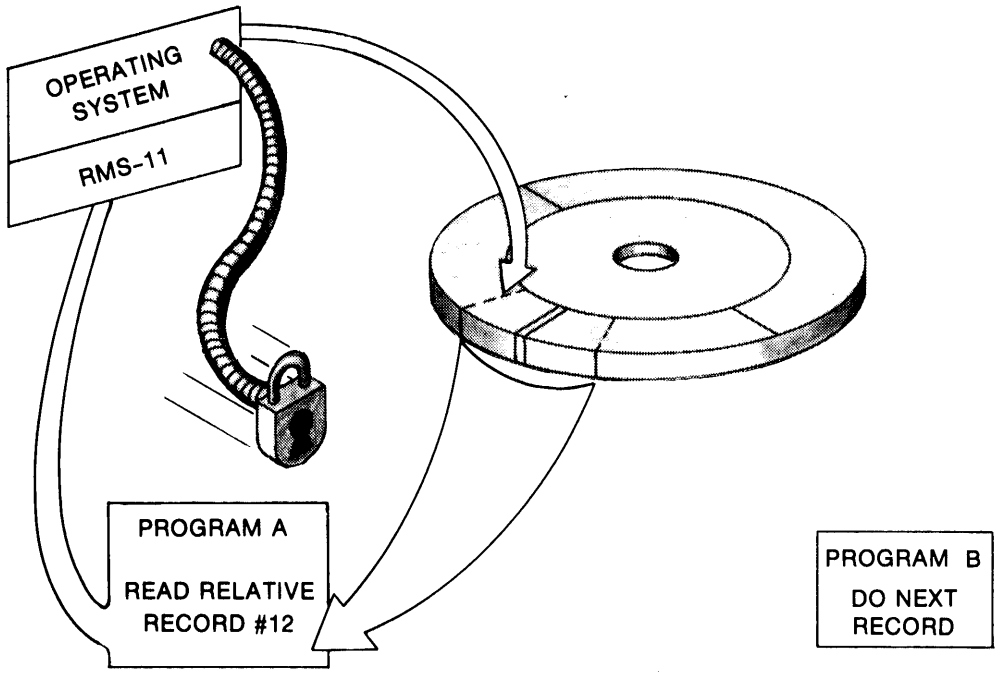
B. PROGRAM A IS DENIED RECORD 12

(continued on next page)

Figure 2-3: Bucket Locking Example (Cont.)



C. PROGRAM B UPDATES RECORD 12, UNLOCKING BUCKET



D. PROGRAM A GETS RECORD 12

F-MK-00097-00

Cost The operating system administers the bucket-locking process. It establishes, for each file, a list of virtual blocks that are locked. The system must scan this list every time RMS-11 performs a read-type operation and then either permit the read or return an error.

In addition to this lock-list overhead, extra instructions are executed to lock and unlock the buckets. The unlock sequence is particularly costly because RMS-11 makes a special monitor call.

File Organization Restrictions — The file organizations restrict file sharing and bucket locking.

Sequential Files

Programs can share Sequential files for read access only. If a program accesses such a file to perform write-type operations, no other programs can open that file. Conversely, if a reading program opens a Sequential file, the operating system prevents writing programs from accessing it. Therefore, programs attempting to open Sequential files must allow no write access.

The primary reason for this restricted access is that Sequential files are not sufficiently formatted to permit simple and economical control of sharing.

Relative and Indexed Files

Sharing of Relative and Indexed files works as described in this section.

2.2.3.3 Sharing among Record Access Streams — In addition to the bucket locking used when programs allow sharing, RMS-11 activates its own version of bucket locking when a program accesses a file for write-type operations. This locking allows multiple Record Access Streams to share the file.

RMS-11 bucket locking works in the same manner as the locking administered by the operating system, except that the locks can be encountered only by different Record Access Streams within the same program.

Cost The overhead for RMS-11 bucket locking is small.

2.2.3.4 Programming Considerations — For the greatest flexibility at run time, you should always assume that any record your program attempts to access 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 Record 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 Record Access Stream associated with the operation.

MACRO-11
Issue a $FREE macro.

BASIC-PLUS-2
Use an UNLOCK or FREE statement.

PDP-11 COBOL
PDP-11 COBOL does not support the free operation.

RPG II
RPG II does not support the free operation.

DIBOL
DIBOL does not support the free operation.

- If your program detects an ER$RLK error (or its higher level language equivalent described in Appendix B), its error processing depends on the number of Record Access Streams active on the file:

  Single Stream
  Set up a loop that waits, 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:

  - continue processing on the other streams, attempting the record operation on the locked-out stream periodically
  - release the buckets locked by all the 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, since the release of a bucket destroys context (see "Record Access Streams," Section 1.2.4.3).

2.2.4 Remember Ease of Design

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

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

  Example From Sequential through Indexed, the RMS-11 file organizations offer more capability, but they are also more complex. Choose the organization that supplies enough capabilities, but no more. For instance, if you want to random access a file by a single key only, you might use a Relative file and some hashing 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 Indexed performance 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, recently a PDP-11 COBOL program needed optimization. The Indexed file being read was made contiguous (discussed in Chapter 6) and the RMS-11 overlay structure was changed (discussed in Chapter 8): execution time dropped from 16 minutes to 8.5. Since this performance was adequate, no more optimizations were considered.

  Example Some optimizations apply to one type of record operation, but not to others. Determine if 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. Table 2-2 shows the capabilities of the RMS-11 file organizations. Table 2-3 describes their advantages and disadvantages.

Once you've selected, go to the appropriate chapter(s):

Sequential	Chapter 3
Relative	Chapter 4
Indexed	Chapter 5

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. You must consider bucket sizes, areas, placement control, index levels, and so on.

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

Finally, you employ the design considerations described in this manual. Write the application programs. 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.

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

2.4 Selecting a File Organization

Table 2-2 lists important features of each file organization to help you decide which one(s) you need. Table 2-3 points out advantages and disadvantages.

But first some information about certain of those features to help you decide:

Record Formats — RMS-11 supports all of the following record formats for Sequential files, but restricts Relative and Indexed file organizations (see Table 2-2):

Fixed

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

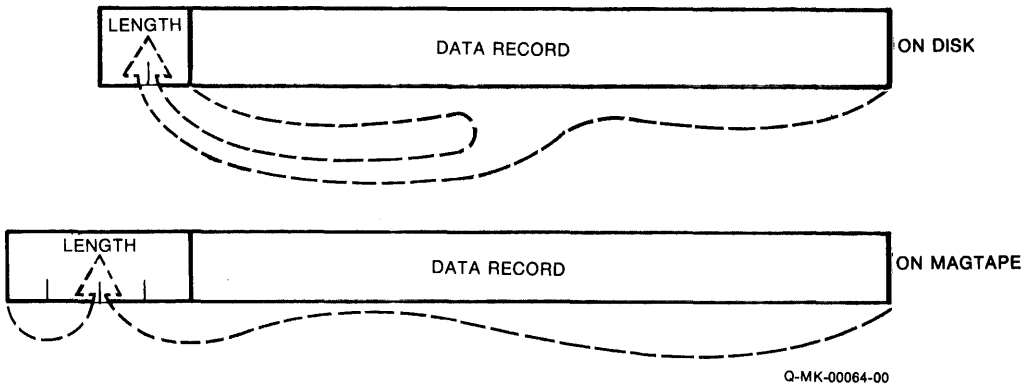
RMS-11 limits fixed block-spanning records to 32,765 bytes, while the minimum valid record is one byte of data on disk, 18 bytes on magnetic tape.

Variable

Records in the file can be any length, up to a maximum stored as a file attribute. For each record, RMS-11 maintains a record-length field specifying the number of data bytes in the record. The size of this field depends on the storage medium for the file (see also Figure 2-4):

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

Figure 2-4: Record-Length Field on Disk and Tape



The variable record format:

- should be used when the data truly varies in length, because the format adds the record-length field to each record's size
- can be used in a new application when future uses may require records to change length

NOTE

Changing a record's length during an update operation is restricted by file organization. See the "Record Operations" sections of the organization-specific chapters.

RMS-11 limits variable-length block-spanning records on disk to 32,763 bytes because of the record-length 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 a two bytes of zeroes representing a null record.

Variable-With-Fixed-Control (VFC)

A VFC record consists of two areas:

- a fixed control area from one to 255 bytes in length; the length is maintained as a file attribute.
- a variable 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 record-length field specifying the number of data bytes in the record including fixed and variable areas. The size of this field depends on the storage medium for the file (see Figure 2-4):

- On disk, the field is a two-byte binary count that does **not** include the length of the field.
- On ANSI magnetic tape, the field is a four-character decimal count that **does** include the length of the field.

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

Stream

A stream record consists of a series of contiguous ASCII characters. RMS-11 detects the end of a stream record only by the presence of one of the following terminators:

Carriage Return-Line Feed(015_8/012_8)
CTRL/Z (032_8)
Escape (033_8)
Form Feed (014_8)
Line Feed (012_8)
Vertical Tab (013_8)

RMS-11 limits stream format to Sequential disk files. Additionally, 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:

Find and Get Operations

RMS-11 scans the stream of ASCII characters, removing leading NUL (000_8) characters and searching for the first occurrence of one of the terminators:

- If it finds a Form Feed, Vertical Tab, Line Feed, or Escape, RMS-11 includes the terminator character with the record.

- If it finds a CTRL/Z:
  - and it has encountered only NUL characters, RMS-11 returns ER$EOF error code.
  - and it has encountered non-NUL characters, RMS-11 includes the terminator character with the record. RMS-11 also notes end-of-file has occurred; ER$EOF will be returned on the next find or get operation.

- 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 terminator characters (Carriage Return and Line Feed) and considers the record complete.
  - If the next character is not a Line Feed, RMS-11 includes the characters 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 ASCII characters. RMS-11 does not move any data into the user buffer during a find operation.

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, Form Feed, or Escape, 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 character sequence to the end of the record.

Undefined

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

I/O Techniques — RMS-11 supports the following techniques so you can adjust the performance 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 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.

Table 2-2: 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	Yes	Yes	No
Stream	Yes	No	No
Undefined	Yes	No	No
Record Overhead	None	One byte per record	Seven bytes per record
Access Modes			
Sequential	Yes	Yes	Yes
Random	No	Yes	Yes
Access by RFA	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 (disk only)	No	No
Update	Yes	Yes	Yes
Put	Yes	Yes	Yes
I/O Unit	One or more blocks	Bucket	Bucket
I/O Techniques			
Deferred Write	Normal mode of operation	Selectable	Selectable
Multi-Block Count	Yes	Use bucket size	Use bucket size
Multiple Record Access Streams	No	Yes	Yes
Multiple Buffers	No	Yes	Yes
Mass Insert	No	No	Yes
File Sharing*	Multiple readers only	Multiple readers and writers	Multiple readers and writers
Other Features	Block-spanning records	Maximum Record Number	

* Important: see system-specific exceptions in "Provide Shared Access," Section 2.2.3.

Multi-Block Count (MBC)

You can open a 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 (MBF)

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 only one task is accessing the file, RMS-11 uses the buffers to save in memory, or cache, buckets from the file, so that they do not have to be read from disk again 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 Insert

Turned on 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 Insert significantly improves performance for single-key Indexed files. However, with each additional key defined for the file, the improvement is smaller.

Table 2-3: File Organization Advantages and Disadvantages

	Sequential	Relative	Indexed
Advantages	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 nonRMS-11 systems • compatible with IAS and RSX-11M FCS files* • compatible with ANSI magnetic tape format	Random access in all languages Allows deletions Allows random get and put operations Optimal if application accesses all records on each run and file must be write-shared Random and sequential access with low overhead	Most flexible random access: • by any one of multiple keys or RFA • key access by generic or approximate value • you access records by record contents Duplicate key values possible Fast sequential access Automatic sort of records by Primary and Alternate keys; available during sequential access Record location is transparent to user.

(Continued on next page.)

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

	Sequential	Relative	Indexed
	• compatible with RSTS/E stream files* Most versatile in storage media; file is portable	Can be write-shared	Can be write-shared Potential range of key values not physically present as in Relative file organization
Disadvantages	To get a record, most higher level languages must access all records before it (no access by RFA). PDP-11 COBOL program cannot access a record already passed without closing and re-opening file (rewind is not available). You can add records only at end of file. You can delete records only at end of file; use truncate record operation. Interactive process is awkward: operator must wait as a program searches for a record. Sharing restricted to multiple readers	Restricted to disk File contains a cell for each cell number between 1 and last record in file; data may not be stored densely. 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. Interactive access can be awkward if you do not access records by relative number. You can insert records only into unused record cells, but you can update existing records. RMS-11 does not allow duplicate relative record numbers.	Highest overhead on disk and in memory Restricted to disk Least simple programming

* 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. See also Appendix A.

Chapter 3
Sequential File Applications

CONVENTION

The cover term *file directory* in this manual has the same meaning as the following system-specific terms:

System	Term
IAS	directory entry and file header(s)
RSTS/E	User File Directory entries
RSX-11M	directory entry and file header(s)

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 1:

- If records cross block boundaries, 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 (see Appendix A), RMS-11 flags space not used at the end of each block as shown in Table 3-1. When you allow records to span blocks, the only unused space starts after the last record in the file.

Table 3-1: End-of-File Indicators

Medium	Record Format	EOF Indicator
Disk	All but Stream	-1 in word following last valid byte
Disk	Stream ASCII	nulls (000_8) to end of block
Magnetic tape	All	circumflex (^) to end of block
Unit record	All	CTRL/Z (032_8)

However, for disk Sequential files, RMS-11 uses the end-of-file attribute, stored in the file directory, to tell where the valid data in a file ends. It does **not** rely on the indicators shown in Table 3-1. This attribute includes a Virtual Block Number 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.

Example The end-of-file indicators shown in Table 3-1 are like the words "THE END" printed on the last page of a book. On the other hand, the end-of-file attribute that RMS-11 uses is like listing the last page or "THE END" in the table of contents.

RMS-11 reads the end-of-file 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 — 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. RMS-11 can access the records in the same order or randomly via Record's File Address (disk files only).

3.1 Record Definition

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 sizes for records in a Sequential file. These are the sizes of your data; they are adjusted for RMS-11 restrictions and overhead.

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

Format	Maximum Data Size: With Block-Spanning	Maximum Data Size: Without Block-Spanning	Data Size Calculation
Fixed	32766	512	Your data + MOD(*yd*/2)[1]
Variable	32765	510	Your data + 2 + MOD(*yd*/2)[1]
VFC	32765[5]	509	Fixed + variable + 2 + MOD(*yd*/2)[1]
Stream	None	511[2]	Data + terminator(s)

[1] MOD(*yd*/2) is the remainder after the length of your data (*yd*) is divided by 2:

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

For VFC, *yd* = fixed + variable.

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

3.2 File Design

With Sequential files, design includes:

Record format selection
Record Formats in Section 2.4 completely discusses your choices of record formats.

Medium selection
Sequential files can be accessed on disk and magnetic tape. When you select the medium for your file, 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
RSTS/E disk structure is not compatible with IAS, RSX-11M, or VAX, and vice-versa. If you need to use the file across these systems, you should consider using a magtape file.

Allocation
Allocation involves two different 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 when RMS-11 extends it automatically

The concept of contiguity involves both these quantities. Contiguity significantly impacts performance, but its use differs by operating system (discussion in Chapter 8).

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 amount 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 reside in the file at one time

RSZ is the size of the record in bytes:

- For a variable record format (VAR or VFC), use the average record size, including two bytes for the record-length field.
- For fixed-length records, use the actual record size.

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

This allocation can be done by the RMSDEF utility or by your application program as follows:

MACRO-11
Use the initialization macro or $STORE to set the ALQ field in the FAB (or XAB if you are using placement control) of the file to the calculated number of blocks before issuing $CREATE.

BASIC-PLUS-2
Use the FILESIZE clause in the OPEN statement that creates the file.

PDP-11 COBOL
Use the /AL:*n* switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
Use the RPGASN utility to override the default value set by the compiler with a switch.

DIBOL
Use RMSDEF; DIBOL does not support allocation quantities during the creation of Sequential files.

3.2.2 Default Extension Quantity

You should establish a reasonable Default Extension Quantity (DEQ) whether the file is totally allocated at creation time or not. A reasonable DEQ minimizes the number of file extensions. The time required for each file extension is significant; involved are:

- A call to the system file processor
- Possible I/O operations to bring file 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 to determine the number of blocks.

If you do not specify a DEQ, it defaults to zero whether you create the file with RMSDEF or a higher level language. RMS-11 responds to a DEQ of zero by requesting five blocks from the file processor each time it automatically extends the file.

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

- If DEQ is zero, RMS-11 extends the file by five blocks each time it runs out of space. Therefore, in this example, RMS-11 extends the file twenty 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 four times.
- If DEQ is 100, RMS-11 extends the file only once.

The DEQ for the file can be set by the RMSDEF utility or by your application program as follows:

MACRO-11
: Use the initialization macro or $STORE to set the DEQ field in the FAB (or XAB if you are using placement control) of the file to the calculated number of blocks before issuing $CREATE. You can also set a run-time DEQ.

BASIC-PLUS-2
: Use RMSDEF; BASIC-PLUS-2 does not support DEQ specifications.

PDP-11 COBOL
: Use the /EX:*n* switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
: Use the RPGASN utility to override the default value set by the compiler with a switch.

DIBOL
Use RMSDEF; DIBOL does not support DEQ specifications during the creation of Sequential files.

3.2.3 Contiguity

Finally, you should consider contiguity for a Sequential file to minimize the time spent in each I/O operation. If the blocks in a file are not contiguous, they can be on different parts of the disk. The device must therefore move its heads to access the file contents. However, physical contiguity ensures that the file is stored on one track, or at worst, adjacent tracks. Since the disk can read a track without moving the heads, file contiguity reduces head movement. This statement assumes that no other software is accessing the disk at the same time.

Contiguity also enhances virtual-to-logical-block mapping (discussed in Section 8.3).

To ensure that the blocks in the file are physically contiguous, allocate the whole file when you create it (see "Initial Allocation," Section 3.2.1).

MACRO-11
Use the initialization macro or $SET to set the FOP field in the FAB (or XAB if you are using placement control) of the file to include FB$CTG before issuing $CREATE.

BASIC-PLUS-2
Use the CONTIGUOUS clause in the OPEN statement that creates the file.

PDP-11 COBOL
Use the /CO switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
Use RMSDEF; RPG II does not create contiguous files.

DIBOL
Use RMSDEF; DIBOL does not create contiguous files.

3.3 Task Design

The record and file processing capabilities described in Chapter 1 are available for Sequential files. This section discusses the operations and their implementation and restrictions with Sequential files.

3.3.1 Record Operations

RMS-11 performs a record operation at the request of a program. The available operations include:

Connect
Disconnect
Find
Flush
Get
Put
Rewind
Truncate
Update

In all record operations, except truncate, RMS-11 establishes Current Record (if any) and Next Record (if applicable). If any record operation fails, RMS-11 normally sets Current Record to NONE and does not change Next Record. "Record Access Stream," Section 1.2.4.3, introduces the concepts of Current Record and Next Record.

3.3.1.1 Connect — A connect operation affects the Current Context for the Record Access Stream as follows:

Current Record
: There is no Current Record. Any operation requiring 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.

Your program specifies that it will append records to the file as follows:

MACRO-11
: Use the initialization macro or $SET to set the value RB$EOF in the ROP field of the RAB before issuing $CONNECT.

BASIC-PLUS-2
: Use ACCESS APPEND in the OPEN statement that creates the file.

PDP-11 COBOL
: Use the keyword EXTEND in the OPEN statement.

RPG II
: RPG II does not support this feature.

DIBOL
: DIBOL does not support this feature.

3.3.1.2 Disconnect — A disconnect operation destroys the Current Context for the Record Access Stream. You cannot resume this context by reconnecting the stream.

3.3.1.3 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 access mode:
   - Location is indicated by Next Record pointer in Sequential Access Mode.
   - Location is determined by specified RFA in Access by RFA.
2. reads the block containing the record, or the first part if it 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 it was required by a previous operation.
3. returns the RFA to the program, but does not transfer the record to the program's user buffer

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$EOF, meaning that no record was located because there are no more records in the file.
- In Access by RFA, the error code is ER$RFA, meaning that no record was located at the RFA specified.

A find operation affects the Current Context for the Record Access Stream as follows:

- find in Sequential Access Mode:

  Current Record
  Set to value of the record found, that is, the Next Record before operation started.

  Example You've connected a stream to a Sequential file without specifying append. 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
  Set to the record virtually following the Current Record.

  Example From the previous example, Next Record is the second record in the file.

- find in Access by RFA:

  Current Record
  Set to the record found, that is, the reocrd identified by the RFA.

Next Record
Unchanged.

Example In the preceding examples, you've done a sequential find 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 do another sequential find, Current Record is set to the second record in the file, **not** the record following the one found by RFA.

You use find instead of a get operation because:

- it is quicker 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.
- it does not change Next Record in access by RFA. This convention allows you to branch off sequential processing for updating, deleting, or truncating, and yet 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 with RFA. You could then initiate successive get operations, where the first one gets the record found by RFA.
- To establish a Current Record for an update or truncate operation.

3.3.1.4 Flush — See "Records Operations," Section 1.2.4, for a summary of the flush operation.

A flush operation does **not** affect the Current Context for the Record Access Stream.

3.3.1.5 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:
     - Next Record pointer, if the get operation was **not** immediately preceded by a successful find operation.
     - Current Record pointer set by an immediately preceding successful find operation.
   - Location is determined by specified RFA in Access by RFA.
2. reads the block containing the record, or the first part if it spans blocks, from disk into the task's I/O buffer, if it 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 record operation your program executes.

3. 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. 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.

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$EOF, meaning no record was located because there are no more records in the file.
- In Access by RFA, the error code is ER$RFA, meaning no record was located at the RFA specified.

A get operation affects the Current Context for the Record Access Stream as follows:

- get in Sequential Access Mode **not** immediately preceded by a successful find operation:

  Current Record
  Set to value of the record read, that is, the Next Record before operation started.

  Example You've connected a stream to a Sequential file without specifying append. There is no Current Record, but the Next Record is the first record in the file. If you execute a sequential get operation, the Current Record is set to the first record in the file.

  Next Record
  Set to record virtually following Current Record.

  Example You've connected a stream to a Sequential file without specifying append. There is no Current Record, but the Next Record is the first record in the file. If you execute a sequential get operation, the Next Record is set to the second record in the file.

- get in Sequential Access Mode immediately preceded by a successful find operation:

  Current Record
  Unchanged (from Current Record set by find operation).

  Next Record
  Set to record virtually following Current Record (possibly changing Next Record set by find operation).

- get in Access by RFA:

  Current Record
  Set to record specified by RFA, that is, the record read.

  Next Record
  Set to record virtually following Current Record. This convention differs from find by RFA which does not change Next Record.

3.3.1.6 Put — To perform a put operation on a Sequential file, RMS-11:

1. determines if Sequential Access Mode is specified; if it is not, RMS-11 returns the error code ER$IOP.
2. determines if Next Record is end-of-file; if it is not, RMS-11 returns the error code ER$NEF.

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

   - specifying append when the program connects the Record Access Stream to the file (see "Connect," Section 3.3.1.1).
   - initiating sequential find and/or get operations until RMS-11 returns an ER$EOF error code.

3. reads the last block in the file into the I/O buffer, if it is not already in memory
4. moves the record from the user buffer in the program to the task's I/O buffer
5. writes the I/O buffer to disk only if the buffer is full. If there is not room for the block(s) in the file, RMS-11 extends the file (see "Default Extension Quantity," Section 3.2.2) and then writes the buffer.

A put operation affects the Current Context for the Record Access Stream as follows:

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.

3.3.1.7 Rewind — A Rewind operation sets the context of the Record Access Stream to the beginning of the Sequential file. In doing so, it affects the Current Context for the stream as follows:

Current Record
: None. Any operation requiring a Current Record fails at this point.

Next Record
: Set to first record in file.

3.3.1.8 Truncate — A truncate operation declares an end-of-file at the position of the Current Record. In doing so, the operation deletes the Current Record and all records in the Sequential file following that record.

Truncate 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 Current Context for the Record 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 to the file using put operations.

NOTE

The truncate operation does not reduce the size of a Sequential file.

3.3.1.9 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 the prerequisite get or find operation. However, RMS-11 does not immediately write the buffer to the file. RMS-11 requests the file 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 then 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$RSZ.
- You cannot update Stream format records. If you attempt it, RMS-11 returns the error code ER$IOP.

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

An update operation affects the Current Context for the Record Access Stream as follows:

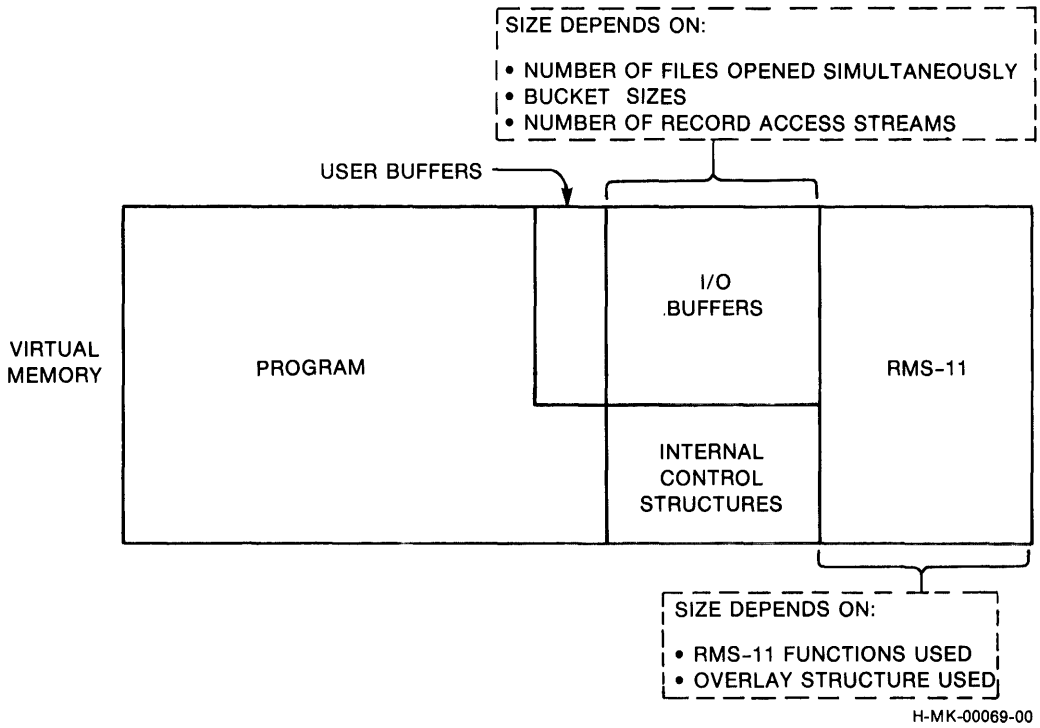
Current Record
None. Any operation requiring a Current Record fails at this point.

Next Record
Unchanged.

3.3.2 Record Transfer Modes

You can manipulate records either in the I/O buffer or in your program's user buffer (see Figure 3-1). 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: RMS-11 Task Structure



3.3.2.1 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 on all file organizations.

3.3.2.2 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 the I/O buffer 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 get and put records in Locate Mode. See your language documentation to determine if the language supports Locate Mode and if it does, what the programming techniques are.

3.3.3 I/O Techniques

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

3.3.3.1 IAS/RSX-11M 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 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 language documentation for asynchronous techniques.

NOTE

If you intend to use asynchronous RMS-11 record operations and/or Asynchronous System Traps (ASTs) in other parts of your program, see the section on your operating system in Appendix A.

3.3.3.2 Deferred Write — The normal mode of operation for Sequential files is similar to operations using Deferred Write with the other file organizations. Using this technique does not change or improve performance.

3.3.3.3 Multiple Buffers — The multiple buffer capability is not available to Sequential files.

3.3.3.4 Multiple Record Access Streams — RMS-11 allows each program to use only one stream on a Sequential file because Sequential files are not sufficiently formatted to permit simple and economical sharing.

3.3.3.5 Multi-Block Count (MBC) — 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 as 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 MBC, the I/O buffer in the task grows by 512 bytes.

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

Example You are using fifty-byte records. During sequential processing, if MBC is one, RMS-11 requests a disk I/O operation for every tenth record operation your program executes, whether the operations are gets or puts. If you set MBC to five, for instance, RMS-11 causes a physical I/O operation for every fifty record operations.

Since MBC is a run-time parameter, the quantity is set by the application program:

MACRO-11
: Use the initialization macro or $STORE to set the MBC field in the RAB of the Record Access Stream being used to the desired number of blocks before the stream is set up ($CONNECT).

BASIC-PLUS-2
: When you specify the organization as SEQUENTIAL in the OPEN statement for the file, the compiler equates the BUCKETSIZE to the Multi-Block Count for the Sequential file. If you try to allocate more buffer space than is available to your task at runtime, the task terminates with an error (?Maximum memory exceeded).

PDP-11 COBOL
: When you specify the organization as SEQUENTIAL in the OPEN statement for the file, the compiler equates *n* in the RESERVE *n* AREAS clause to the Multi-Block Count for the Sequential file.

RPG II
: RPG II does not support this feature.

DIBOL
: When you specify a value after the PROC statement, the compiler uses that value as the Multi-Block Count for all Sequential files opened by the program.

3.3.4 File Operations

You can perform the following file operations on Sequential files. File operations do not involve records and can only perform synchronously.

3.3.4.1 Close — A close operation disconnects the Record Access Stream before RMS-11 releases access to the file. You can also specify magnetic tape volume operations during a close operation. See Appendix F.

3.3.4.2 Create — In addition to the file specification, RMS-11 passes the following information to the file processor when it creates a file:

- An initial allocation of blocks for the file.
- The location on a specific device where the processor should allocate those blocks.
- The following file attributes:

  File organization
  Record format
  Forms control
  Record size
  Number of virtual blocks in the file
  End-of-file
  Default extension quantity

3.3.4.3 Open — You can specify the file you want to open in two different ways:

By filespec
: The first time you open a file, you must use the file specification.

By File ID
: When you create or open a file by filespec, RMS-11 returns an identifying notation to your program. You can store this File ID, either in memory or in a file, and use it to open the file from that point on.

On IAS/RSX-11M systems, open by File ID is significantly faster than open by filespec, because the process bypasses directory reads and other overhead. However, on RSTS/E, open by File ID is no faster than open by filespec.

You can also specify magnetic tape volume operations during an open operation. See Appendix D. See also "File Operations," Section 1.2.5.3, for an introduction to the open file operation.

3.3.4.4 Erase — You cannot erase a magnetic tape or unit record file; RMS-11 returns the error code ER$IOP. "File Operations," Section 1.2.5.3, introduces the concept of erasing files.

3.3.4.5 Extend — You cannot extend a magnetic tape file. "File Operations," Section 1.2.5.3, introduces the concept of extending files.

Chapter 4
Relative File Applications

CONVENTION

The cover term *file directory* in this manual has the same meaning as the following system-specific terms:

System	Term
IAS	directory entry and file header(s)
RSTS/E	file directory
RSX-11M	directory entry and file header(s)

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 Virtual Block 1 — unless bucket size is two, four, or eight blocks. In that case, RMS-11 makes the Prologue equal to one bucket in size; this step can improve performance by aligning buckets with file clusters. 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.

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, though they can cross block boundaries in multi-block 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, via both relative record number and RFA.

4.1 Record Definition

RMS-11 calculates the size of a Relative record cell as follows:

1	byte for RMS-11 overhead
rfo	bytes for record format overhead (0 for fixed; 2 for variable)
+ *ds*	bytes in the data itself
CL	bytes in each record cell in the file

The data size used for variable records 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 already adjusted for RMS-11 restrictions and overhead.

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

Format	Maximum	Record Cell Size Calculation
Fixed	16,383 or 7,679	Data size + 1
Variable	16,381 or 7,677	Maximum record size + 3
VFC	16,381 or 7,677	Fixed + variable + 3

4.2 File Design

Though not as critical as for Indexed files, design is still important for an application using a Relative file. Design considerations include:

1. Bucket size
2. File allocation
3. Maximum Record Number

4.2.1 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 it performs. Sequential

access, especially, benefits when there are multiple records per bucket. There is, of course, a trade-off: the larger a bucket, the larger the task, but the faster it reads data sequentially:

- Each block added to the bucket size increases the task size by 512 bytes.
- 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 isn't locked after each get operation and a writing program may have changed the bucket since it was last read.

Bucket size can be set by RMSDEF or by your application program:

MACRO-11
: Use the initialization macro or $STORE to set the BKS field in the FAB (or XAB if you are using placement control) of the file to the chosen number of blocks before issuing $CREATE. Note that the default value is one block per bucket.

BASIC-PLUS-2
: Use the BUCKETSIZE clause in the OPEN statement that creates the file. See "Program Syntax," section 6.5.3, for cautions in the use of the BUCKETSIZE clause.

PDP-11 COBOL
: In the file-description-entry (FD), use the BLOCK CONTAINS clause.

RPG II
: Use the RPGASN utility to override the default value set by the compiler.

DIBOL
: Use RMSDEF; DIBOL does not create Relative files.

4.2.2 Allocation

File allocation involves two different 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

The concept of contiguity involves both these quantities. Contiguity has a significant impact on performance, but its use differs by operating system.

4.2.2.1 Initial Allocation — Total allocation of a file when you create it is the most efficient technique regardless of file organization, but with Relative files, pre-allocation becomes most critical. Each allocation, whether at creation 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 amount as follows:

ALQ = PLG + (NRF/NRBKT)*BKS

where:

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

NRF is equal to 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:

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

where:

RSZ is the size of the record:

- data size for fixed-length records
- maximum record length for variable-length records
- size of fixed control area + maximum variable area size for VFC

RFO is the record format overhead:

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

RMS-11 rounds ALQ up to the next bucket size if you don't.

This allocation can be done by the RMSDEF utility or by your application program as follows:

- during file creation

  MACRO-11
  Use the initialization macro or $STORE to set the ALQ field in the FAB (or XAB if you are using placement control) of the file to the calculated number of blocks before issuing $CREATE.

  BASIC-PLUS-2
  Use the FILESIZE clause in the OPEN statement that creates the file.

  PDP-11 COBOL
  Use the /AL:*n* switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
Use the RPGASN utility to override the default value set by the compiler.

DIBOL
Use RMSDEF; DIBOL does not create Relative files.

- by putting a record with the Maximum Record Number (MRN) in the file first. Before RMS-11 can write this record, it must allocate all record cells from 1 to MRN and initialize the new blocks. When the put operation is finished, the Relative file is completely allocated.

4.2.2.2 Default Extension Quantity — However, if the file cannot be totally allocated at creation, then 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.

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 "Initial Allocation," Section 4.2.2.1.

The default extension quantity should be equal to a multiple of the bucket size.

If you do not specify a DEQ, it defaults to zero whether you create the file with RMSDEF or a higher level language. RMS-11 responds to a DEQ of zero by requesting four times bucket size in blocks from the file processor each time it automatically extends the file.

The DEQ for the file can be set by the RMSDEF utility or by your application program as follows:

MACRO-11
Use the initialization macro or $STORE to set the DEQ field in the FAB (or XAB if you are using placement control) of the file to the calculated number of blocks before issuing $CREATE. You can also set a run-time DEQ.

BASIC-PLUS-2
Use RMSDEF; BASIC-PLUS-2 does not support DEQs.

PDP-11 COBOL
Use the /EX:*n* switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
Use the RPGASN utility to override the default value set by the compiler.

DIBOL
Use RMSDEF; DIBOL does not create Relative files.

4.2.2.3 Contiguity — Finally, you should consider contiguity for a Relative file to minimize the time spent in each I/O operation. If the blocks in a file are not

contiguous, they are, by definition, in different parts of the disk. The device must therefore move its heads to access the file contents. However, physical contiguity ensures that the file is stored on a single track, or at most, adjacent tracks. Since the disk can read an entire track without moving the heads, file contiguity reduces head movement. This statement assumes that no other software is accessing the disk at the same time.

Contiguity also enhances virtual-to-logical-block mapping (discussed in Section 8.3).

Therefore, if possible, you should allocate a Relative file contiguously, and the only way to ensure that all blocks in the file are physically contiguous is to allocate the whole file when you create it.

MACRO-11
: Use the initialization macro or $SET to set the FOP field in the FAB of the file to include FB$CTG before issuing $CREATE.

BASIC-PLUS-2
: Specify CONTIGUOUS in the OPEN statement that creates the file.

PDP-11 COBOL
: Use the /CO switch on the file specification in the ASSIGN clause or the VALUE of ID.

RPG II
: Use RMSDEF; RPG II does not create contiguous files.

DIBOL
: Use RMSDEF; DIBOL does not create Relative files.

4.2.3 Maximum Record Number

The Maximum Record Number (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 if 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 Maximum Record Number were written into the file.

You can calculate the file size (FSZ) in buckets from the largest relative record number (LRN) in the file (greatest value equals MRN):

$$FSZ = \frac{LRN}{(BKS*512)/(RSZ+RFO)}$$

where:

BKS is the bucket size in blocks

RSZ is the size of the record:

- size of your data for fixed-length records
- maximum record length for variable-length records
- size of fixed control area + maximum variable area length for VFC

RFO is the format overhead:

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

MRN can be set by RMSDEF or by your application program:

MACRO-11
Use the initialization macro or $STORE to set the MRN field in the FAB of the file to the desired number of records before issuing $CREATE. Note: if you want no limit checks, do not include F$MRN in the FAB's initialization block or set the field to zero at run time prior to initiating the $CREATE macro.

BASIC-PLUS-2
Use RMSDEF; BASIC-PLUS-2 does not support MRN specifications.

PDP-11 COBOL
Use RMSDEF; PDP-11 COBOL does not support MRN specifications.

RPG II
Use RMSDEF; RPG II does not support MRN specifications.

DIBOL
Use RMSDEF; DIBOL does not create Relative files.

4.3 Task Design

The record and file processing capabilities described in Chapter 1 are available for Relative files. This section discusses the operations and their implementation and restrictions with Relative files.

4.3.1 Record Operations

RMS-11 performs a record operation at the request of a program. The available operations include:

Connect
Delete
Disconnect
Find
Flush
Get
Put
Rewind
Update

In all record operations, RMS-11 establishes Current Record (if any) and Next Record (if applicable). If any record operation fails, RMS-11 normally sets Current Record to NONE and does not change Next Record. See "Record Access Stream," Section 1.3.1.5, for an introduction to the concepts of Current Record and Next Record.

4.3.1.1 Connect — A connect operation affects the Current Context for the Record Access Stream as follows:

Current Record
: There is no Current Record. Any operation requiring Current Record fails at this point.

Next Record
: The Next Record is the first record cell in the file.

4.3.1.2 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. The prerequisite get or find operation brought the cell's bucket into the I/O buffer.

Then, RMS-11 writes the bucket over its original location on the disk, unless you have specified Deferred Write (discussed in "I/O Techniques," Section 4.3.3).

A delete operation requires a valid Current Record. Therefore, a delete 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 Context for the Record Access Stream as follows:

Current Record
: None. Any operation requiring a Current Record fails at this point.

Next Record.
: Unchanged.

4.3.1.3 Disconnect — A disconnect operation destroys the Current Context for the Record Access Stream. You cannot resume this context by reconnecting the stream.

4.3.1.4 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:
   - Location is indicated by the Next Record pointer in Sequential Access Mode.
   - Location is determined by the specified relative record number and match criterion in Random Access Mode.
   - Location is determined by the specified RFA in Access by RFA.

2. reads the bucket containing the indicated cell from disk into the task's I/O buffer, if it is not already in memory. The bucket may be in memory if it was required by a previous operation.
3. checks the contents of the cell:
   - If the cell contains a valid record, RMS-11 returns the RFA 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.
     - In Random Access Mode, RMS-11 reacts according to the specified match criterion:

       On equal match, RMS-11 returns the error code ER$RNF

       On greater-than or greater-than-or-equal match, RMS-11 adds one to the relative record number and repeats steps 1 through 3.
     - In Access by RFA, RMS-11 returns the appropriate error code:

       ER$RFA
       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 Current Context for the Record Access Stream as follows:

- find in Sequential Access Mode:

  Current Record
  Set to the relative record number of the record found, that is, the Next Record before operation started.

  Example You've 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 find operation, the Current Record is set to the first record in the file.

  Next Record
  Set to relative record number one higher than relative record number for Current Record.

  Example From the previous example, Next Record is the second record cell in the file.

- find in Random Access Mode or Access by RFA:

  Current Record
  Set to record found, that is, the record identified by the relative record number or RFA.

Next Record
Unchanged.

Example In the previous examples, you've done a sequential find 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, if you do another sequential find, Current Record will be set to the second record cell in the file, **not** the cell following the one found by RFA.

You use find instead of a get operation because:

- it is quicker 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.
- it does not change Next Record in Random Access Mode or Access by RFA. This allows you to branch off sequential processing for purpose of updating or deleting and yet 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 with RFA. You could then initiate successive get operations, where the first one gets the record found by RFA.
- To establish a Current Record for a delete or update operation.
- To determine the existence of a record in Random Access Mode.

4.3.1.5 Flush — See "Records Operations," Section 1.2.4, for a summary description of the flush operation.

A flush operation does **not** affect the Current Context for the Record Access Stream.

4.3.1.6 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:
     - the Next Record pointer, if get was **not** immediately preceded by a successful find operation
     - the Current Record pointer set by an immediately preceding find operation
   - Location is determined by the specified relative record number in Random Access Mode.
   - Location is determined by specified RFA in Access by RFA.

2. reads the bucket containing the indicated cell from disk into the task's I/O buffer, if it is not already in memory.

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

   NOTE

   If you have opened a Relative file with read-only access and allow write declarations, each get operation causes an I/O operation.

3. checks the contents of the cell:

   - If the cell contains a valid record, RMS-11 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.
   - 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.
     - In Random Access Mode, RMS-11 reacts according to the specified match criterion:

       On equal match, RMS-11 returns the error code ER$RNF.

       On greater-than or greater-than-or-equal match, RMS-11 adds one to the relative record number and repeats steps 1 through 3.

     - In Access by RFA, RMS-11 returns the appropriate error:

       ER$RFA
       No valid record has ever existed at the specified location.

       ER$DEL
       The overhead byte in the cell indicates that the record in it was deleted.

A get operation affects the Current Context for the Record Access Stream as follows:

- get in Sequential Access Mode **not** immediately preceded by a find operation:

  Current Record
  Set to the relative record number of the record read. See "Find in Sequential Access Mode" for example.

Next Record
Set to relative record number one higher than relative record number for Current Record. See "Find in Sequential Access Mode" for example.

- get in Sequential Access Mode immediately preceded by a successful find operation:

Current Record
Unchanged (from Current Record set by find operation),

Next Record
Set to relative record number one higher than relative record number for Current Record (possibly changing Next Record set by find operation),

- get in Random Access Mode or Access by RFA:

Current Record
Set to the relative record number of the record read.

Next Record
Set to relative record number one higher than relative record number for Current Record. This differs from find by RFA which does not change Next Record.

4.3.1.7 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 destination.
   - In Random Access Mode, the specified relative record number indicates destination.
2. determines if 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 it 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 contains a valid record, returns error code ER$REX; otherwise, goes to next step.
5. moves the record from the user buffer in the program to the task's I/O buffer.
6. writes the I/O buffer to disk, unless you have specified Deferred Write (see "I/O Techniques," Section 4.3.3).

A put operation affects the Current Context for the Record Access Stream as follows:

- put in Sequential Access Mode:

  Current Record
  None. Any operation requiring a Current Record fails at this point.

  Next Record
  Cell with relative record number one higher than relative record number of former Next Record (the cell where this put operation inserted a record).

- put in Random Access Mode:

  Current Record
  None. Any operation requiring a Current Record fails at this point.

  Next Record
  Unchanged.

4.3.1.8 Rewind — A Rewind operation sets the context of the Record Access Stream to the beginning of the Relative file. In doing so, it affects the Current Context for the stream as follows:

Current Record
None. Any operation requiring a Current Record fails at this point.

Next Record
Set to first record cell in file.

4.3.1.9 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 the prerequisite get or find operation. Then, RMS-11 writes the bucket over its original location on the disk, unless you have specified Deferred Write (see "I/O Techniques," Section 4.3.3).

An update operation requires a valid Current Record. Therefore an update 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 Context for the Record Access Stream as follows:

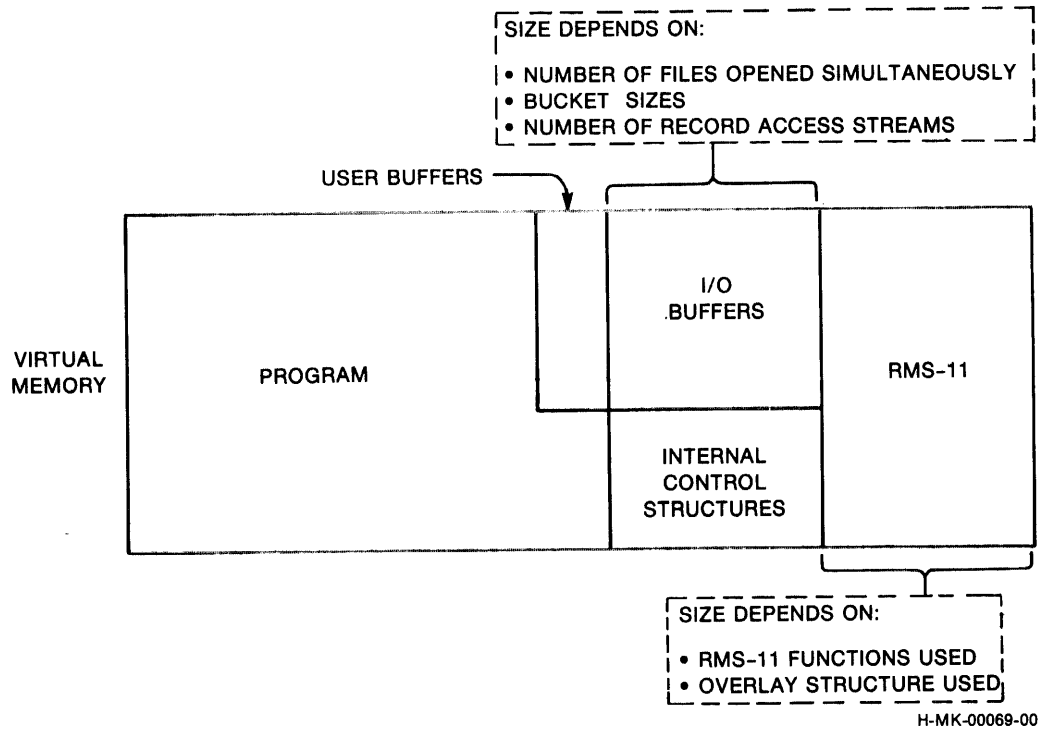
Current Record
None. Any operation requiring a Current Record will fail at this point.

Next Record
Unchanged.

4.3.2 Record Transfer Modes

You can manipulate records either in the I/O buffer or in your program's user buffer (see Figure 4-1). 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: RMS-11 Task Structure



4.3.2.1 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 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 on all file organizations.

4.3.2.2 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 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 get records in Locate Mode. See your language documentation to determine if the language supports Locate Mode and if it does, what the exact programming techniques are.

4.3.3 I/O Techniques

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

4.3.3.1 IAS/RSX-11M 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 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 language documentation for asynchronous techniques.

NOTE

If you intend to use asynchronous RMS-11 record operations and/or Asynchronous System Traps (ASTs) in other parts of your program, see the section on your operating system in Appendix A.

4.3.3.2 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 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.

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 three blocks. During sequential write-type operations, Deferred Write causes I/O operations per bucket to drop from five to one.

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

Only your application program can specify Deferred Write:

MACRO-11
Use the initialization macro or $SET to set the value FB$DFW in the FOP field of the FAB of the Relative file.

BASIC-PLUS-2
BASIC-PLUS-2 does not support Deferred Write.

PDP-11 COBOL
PDP-11 COBOL does not support Deferred Write.

RPG II
RPG II does not support Deferred Write.

DIBOL
DIBOL does not support Deferred Write.

4.3.3.3 Multiple Buffers — When you open a Relative file, normally RMS-11 allocates one 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 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.

4.3.3.4 Multiple Record Access Streams — RMS-11 allows each program to use from one to 255 streams on a Relative file.

4.3.4 File Operations

You can perform the following file operations on Relative files. File operations do not involve records and can only perform synchronously.

4.3.4.1 Close — A close operation disconnects all Record Access Streams connected to a file before it releases access.

4.3.4.2 Create — In addition to the file specification, RMS-11 passes the following information to the file processor when it creates a file:

- An initial allocation of blocks for the file.
- The locations on a specific device where the processor should allocate those blocks.
- The following file attributes:

  File organization
  Record format
  Forms control
  Record size
  Number of virtual blocks in the file
  Bucket size
  Default extension quantity

  The other file attributes, such as the Virtual Block Number of the first block in the first bucket and the last initialized block, RMS-11 stores in the Prologue of the file.

4.3.4.3 Open — You can specify the file you want to open in two different ways:

By filespec
: The first time you open a file, you must use the file specification.

By File ID
: When you create or open a file by filespec, RMS-11 returns an identifying notation to your program. You can store this File ID, either in memory or in a file, and use it to open the file from that point on.

On IAS/RSX-11M systems, open by File ID is significantly faster than open by filespec, because the process bypasses directory reads and other overhead. However, on RSTS/E, open by File ID is no faster than open by filespec.

4.3.4.4 Erase — "File Operations," section 1.2.5.3, introduces the concept of erasing files.

4.3.4.5 Extend — "File Operations," section 1.2.5.3, introduces the concept of extending files.

Chapter 5
Indexed File Organization

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

Contents-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's File Address*) established by the put operation. The record may be deleted, but its unique identifier is never reused.

Preserve state of processing despite 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.

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

- File corruption is avoided or minimized even if the 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 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 "Initial Allocation," Section 6.6.1.

Also, if both of the following are true:

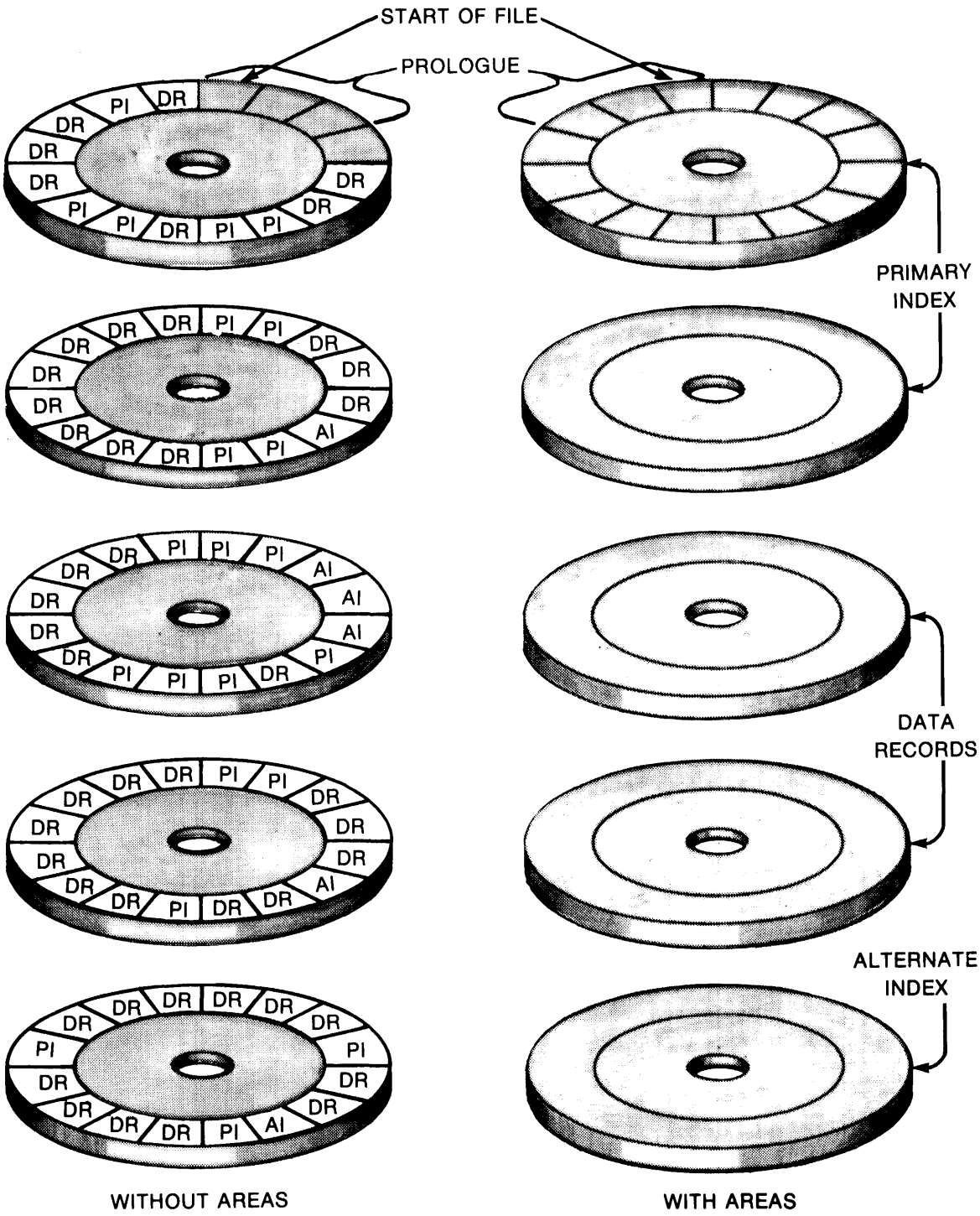
- you have defined the same bucket size for all areas of the indexed file
- that bucket size is two, four, or eight blocks

RMS-11 extends the Prologue to an integral multiple of bucket size. This step can improve performance by aligning buckets with file clusters. If the bucket size does not meet these criteria, RMS-11 does not make the Prologue larger than necessary. "Bucket Size," Section 6.5, contains more discussion.

The location of the index and data blocks is up to you (see Figure 5-1):

- 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: Indexed File with and without Areas



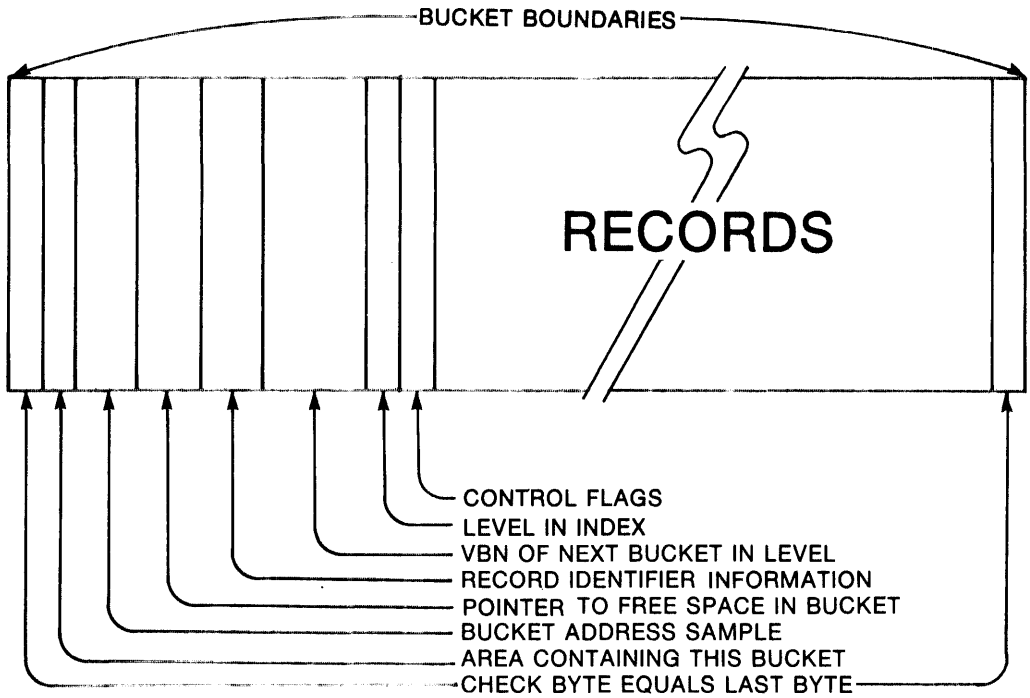
RMS-11 formats buckets in an Indexed file as it requires them for record storage. The RMS-11 control bytes are set to their initial values (see Figure 5–2):

- Fourteen 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: Formatted Bucket

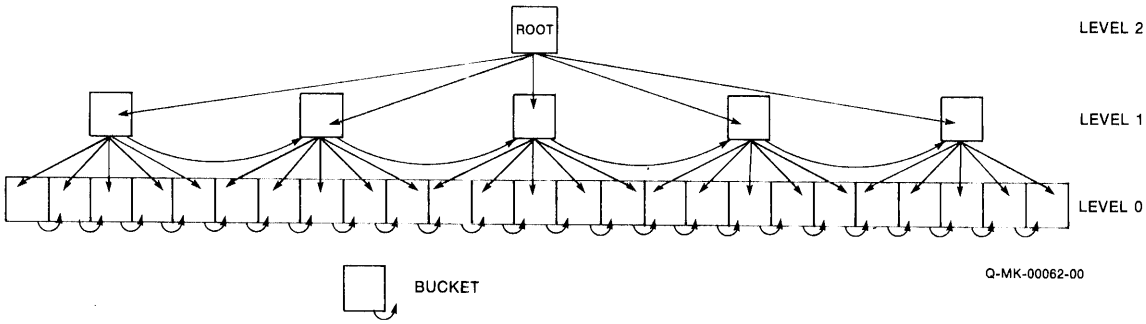


H-MK-00063-00

5.2 Conceptual 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* and can be illustrated as a pyramid (see Figure 5–3).

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 increasing Primary Key value along the bucket chain. The records having the lowest Primary Key value reside in the first bucket of the level and so on until the records with the highest Primary Key values comprise 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.

With these properties, each bucket has a *High-Key value*, located in the last record of the bucket. This concept is the core of RMS-11 Index file structure.

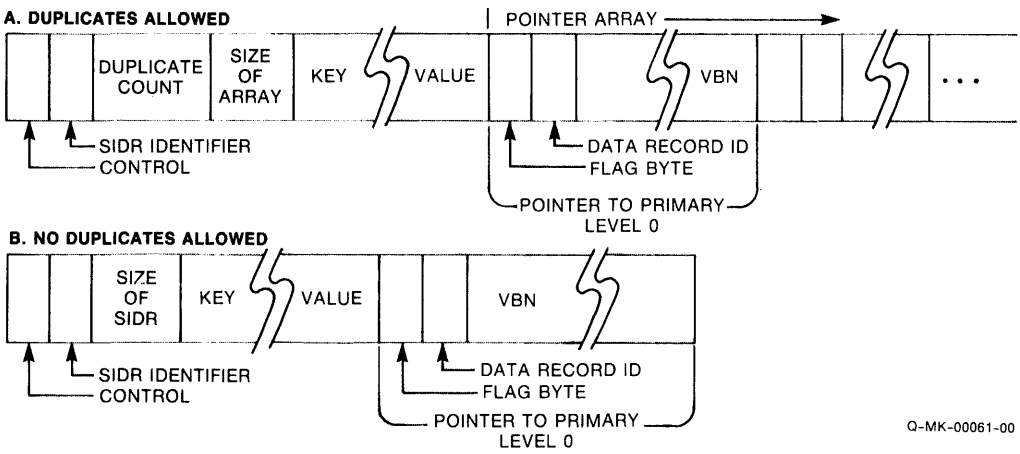
NOTE

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 the Alternate Indexes — Levels 0, the data levels, of Alternate indexes contain *Secondary Index Data Records (SIDRs).* A SIDR consists of (see Figure 5-4):

- 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.
- 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: Format for Secondary Index Data Record



5.2.2 Indexes

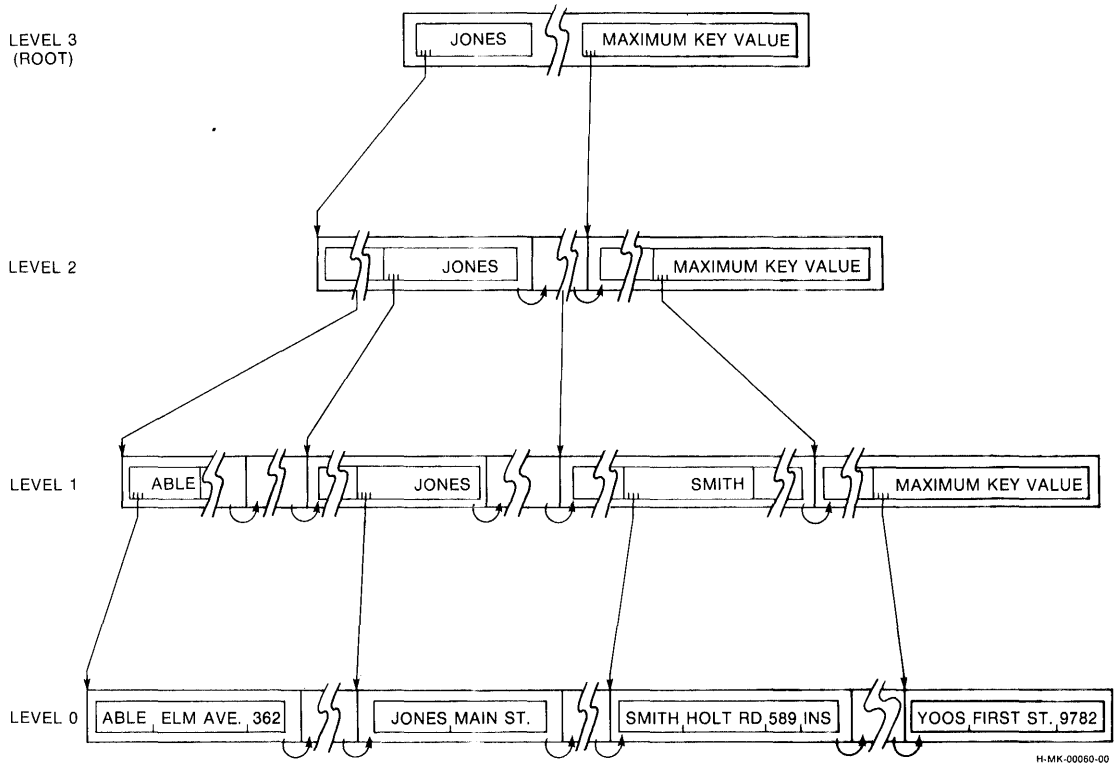
Levels 1 and above in an index are called the *index levels*; they consist of *index buckets.* Index buckets contain *index records* that guide RMS-11 through the levels to the data records in Primary Level 0. An index record consists of:

- the High-Key value of a bucket in the next lower level in the index. Since 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.

- 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 a sample Primary index.

Figure 5–5: 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 ten 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

5.2.3 Random Access Using the RMS-11 Indexed File Structure

The following steps show how RMS-11 uses this Indexed file structure to execute a random access operation. These steps comprise 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

   The Root can be cached (see "I/O Techniques," Section 7.3), 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 during this example.

The specified Primary Key value is "YOOS."

RMS-11 determines the Virtual Block Numbers (VBNs) of the Root bucket from the memory-resident Index Descriptors and requests the file 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 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, containing the maximum key value. This index record points to a Level 1 bucket.

Upon RMS-11's request, the file 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 processor reads the indicated Level 0 bucket at RMS-11's request.

5.2.4 Why this Structure?

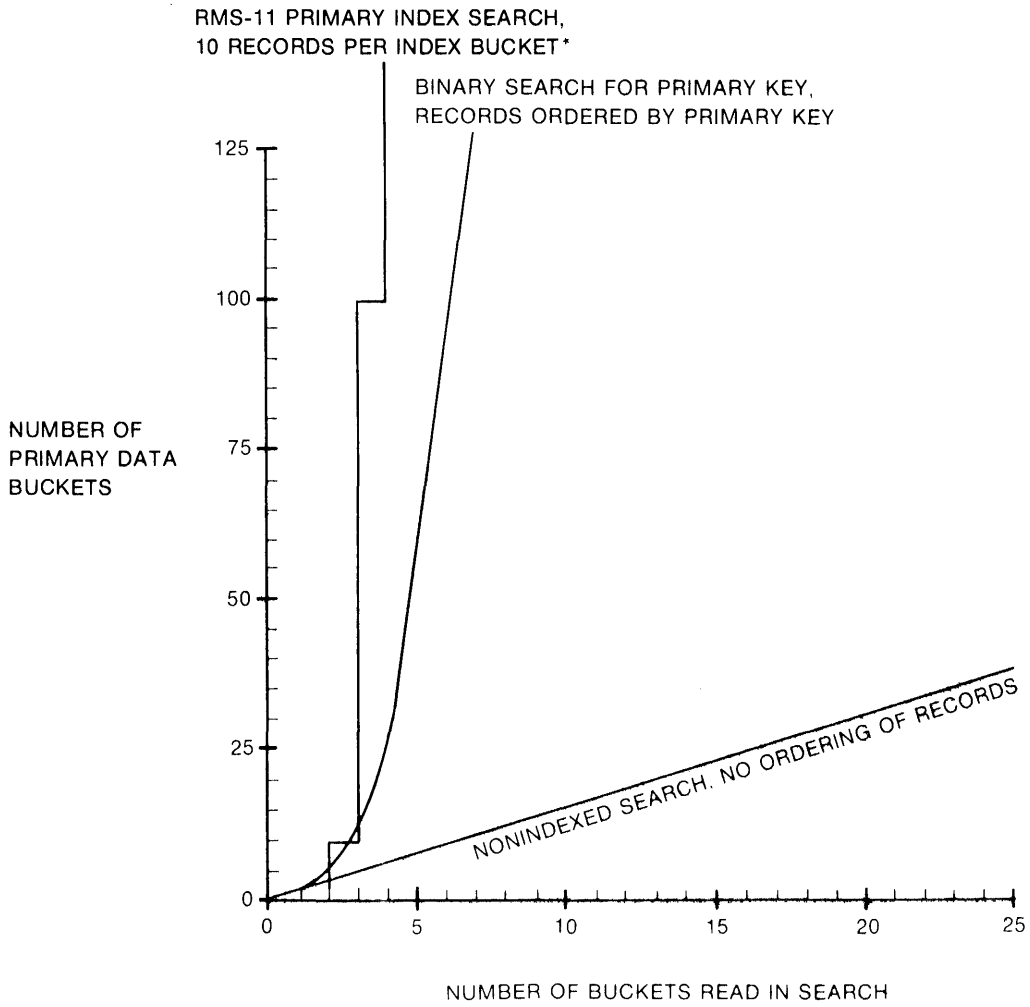
Mechanical data storage devices make I/O operations the slowest part of file processing. Ideally, a file is read to memory when it is opened and maintained there, without additional I/O operations, until the file is closed. Some very small files allow this approach and are handled most efficiently by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

However, most files you require for indexed access are very much larger than the memory available for data buffering. Such files are 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 sequentially scan through the buckets of a file to find a specific record. Such a search, on the average, accesses half the buckets in the file (see Figure 5-6).

Figure 5-6: Search Time Curves



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 quicker 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.

5.3.1 Putting 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 minimum 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. Establishes a position before that record, or after the last existing record in the bucket if:

   - 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. "Key Characteristics," Section 6.2.5, discusses reusing space from deleted records.

5. Determines if 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:

   a. Follows the Alternate index to the proper Level 0 bucket.

   b. 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 SIDR for the record before the SIDR for the higher value.
      - If a match is found, determines if duplicates are allowed for the Alternate Key:

        - If duplicates are allowed, 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, returns a successful completion code to the program.

        - If duplicates are not allowed, 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 during 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 processor to read the bucket into the I/O buffer associated with the stream. When that I/O operation ends, 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 it 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 in a procedure called *bucket splitting.*

This insertion process is identical with the simplest case (Section 5.3.1.1) to step 5 where RMS-11 determines if 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, formats them into a bucket and updates the Area Descriptor to reflect the new bucket. Otherwise, RMS-11 requests the operating system to allocate enough blocks, then proceeds as described.

2. Splits the target bucket at the point where the record should be inserted. Moves the records in the *high portion* of the bucket into the new bucket; these records have Primary Key values higher than that 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 seven 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 it 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 it will not fit, inserts it into the new bucket. If it will not fit there, creates another bucket (see step 1) and puts 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, 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, update 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:

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

Of course, 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 random get or find operation, RMS-11 performs the following steps:

1. Determines from the instruction initiating the operation the following criteria:

   - key of reference, that indicates the index to search and the key field within the data record to examine
   - value to find
   - value match (equal to, greater than, or either)
   - number of characters for value match

2. Follows the index to the proper Level 0 bucket.

3. Reads the Level 0 bucket. Sequentially scans for the first record with a value in the specified key field that matches the specified value according to the value match criteria. This search can continue into other buckets.

   - If no such record is found, returns an error code.
   - If such a record is found:

     a. Determines which index has been read:

        - If this is the Primary index, goes to the next step.
        - If this is an Alternate index, the record located is a SIDR. Follows the SIDR pointer to the Primary Level 0 data record.

     b. For a get operation only, moves the record to the user buffer associated with the Record Access Stream performing the operation.

     c. Sets the current context for the stream performing this operation. The effect of each record operation on context is described in "Record Operations," Section 7.1.

     d. Returns successful completion code.

Example Refer to Figure 5–5 during 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 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 three 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 to be preceded by a get or find operation, although some higher 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 Record Access Stream performing the operation. Compares those key values with the values in the Current Record:

   - If the Primary Key value changed, returns an error code.
   - If an Alternate Key value changed, checks if you allowed changes for that key:
     - If not, returns an error code.
     - If so, continues processing.

2. For each Alternate Key where the key value changed, deletes the pre-update value from the Alternate index:

   a. Reads the data bucket containing the Current Record, if it is not in memory.

   b. Saves the pre-update Alternate Key value from the Current Record.

   c. Follows the index to the Level 0 bucket that should contain the SIDR for the pre-update key value.

   d. Reads the Level 0 bucket and sequentially scans for the pre-update key value:

      - If a value higher than the one specified is found, goes to the next Alternate index.

        Example RMS-11 is scanning 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, 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, PDP-11 COBOL enables you to specify changes, but no duplicates for keys. In this situation, the PDP-11 COBOL Object Time System (OTS) uses a find operation to check whether an update will cause a duplicate; if it does, the OTS returns an error to your program.

3. Reads the data bucket containing the Current Record, if it is not in memory. 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, inserts the post-update value in the Alternate index:

   a. Reads the data bucket containing the Current Record, if it is not in memory.

   b. Follows the index to the Level 0 bucket that should contain the post-update key value.

   c. Reads the Level 0 bucket and sequentially scans for the post-update key value:

      - If a value higher than the one specified is found, inserts a SIDR for the new record before the SIDR for the higher value.

      - If a match is found, 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, 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, although some higher level languages hide this prerequisite.

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

1. If there are Alternate Keys, updates those indexes, using the same sequence of steps for each:

   a. Reads the data bucket containing the Current Record, if it is not in memory.

   b. Follows the index to the Level 0 bucket that should contain the SIDR for the key value in the deleted record.

   c. Reads the Level 0 bucket and sequentially scans for the specified key value:

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

      - If a match is found, determines if you allow duplicates:

        - If so, follows the duplicate pointer array in the SIDR to the entry for the record being deleted and flags it as deleted.

        - If not, deletes the SIDR.

2. Reads the data bucket containing the Current Record, if it is not in memory.

3. Changes the flag byte in the header of the Current Record to indicate that it is deleted.

4. Writes the bucket to the file.
5. If the record has moved, reads the Level 0 bucket containing the RRV. Changes the flag byte in the RRV to indicate that it 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 "Putting a Record," 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 may use the Sequential Access Mode to perform the following record operations:

- Find
- Get
- Put

During sequential get and find operations, RMS-11 does not usually read an index to locate the specified record. Instead, RMS-11 uses the Next Record pointer for the stream performing the operation to identify the proper data bucket. Then, RMS-11 requests the file processor to move that bucket into the I/O buffer, if it 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 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 random 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:

- 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 (discussed in Chapter 8).

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

Record Operation	Primary Key	Each Alternate Key
Getting or Finding a Record Randomly		
Record in original location	$n+1$	$n+2$
RRV in original location (record moved)	$n+1$	$n+2$
Getting or Finding a Record Sequentially	0–2[1]	1–4[2]
Putting a Record		
Simplest case	$n+2$	$n+2$
Split in data level	$2n+6$[3]	$2n+6$[3]
Bucket split up entire index	$(n**2+11n+20)/2$[4]	$(n**2+11n+20)/2$[4]
Updating a Record[5]		
Alternate Key value did not change	1	0
Alternate Key value changed	1	$2(n+2)$[6]
Deleting a Record		
Record in original location	1	$n+2$[7]
RRV in original location (record moved)	3	$n+2$[7]

[1]Breaks down to:

0 or 1 I/O to position to Current Record
0 or 1 I/O to locate Next Record

[2]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

[3]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

[4]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 3)
5 I/Os to create the new Root

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

[6]$n+2$ if either the old or new key value doesn't 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.

[7]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 the Alternate indexes.
- 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 to the methods used to put the data into the file. This range includes:

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

6.1 Record Definition

You can use only fixed- and variable-length records in RMS–11 Indexed files. RMS–11 calculates their lengths as follows:

7	bytes for RMS–11 record header
rfo	bytes for record format overhead (0 for fixed; 2 for variable)
+ *ds*	bytes in the data itself
RL	bytes for each valid data record

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

```
512   bytes in a block
-15   bytes 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 coordination 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.

Of course, if your application requires 490–byte records, use them, recognizing the preceding limitation and perhaps, choosing a different bucket size. However, if you can avoid that or comparable lengths for larger buckets, do so.

NOTE

Records in an Indexed file cannot span buckets and bucket sizes are limited by the operating systems. Therefore, the maximum record size, including overhead, is 16,369 or 7,665 bytes.

6.2 Key Selection

A file's keys cause a significant portion of an Indexed file and of the I/O operations needed to access the file. During key selection, you should consider the following:

- Number of keys
- Key size
- Key data type
- 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 file
- 254 Alternate Keys

Cost 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 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 "Initial Allocation," Section 6.6.1).

Whether the cost of each Alternate Key is bearable depends on your application. If its primary purpose is to put, 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 Size

Keys for Indexed files have length restrictions according to their data types, as shown in Table 6-1; see also Section 6.2.3.

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

Cost 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. However, there is only one index record for each bucket in the next level down.

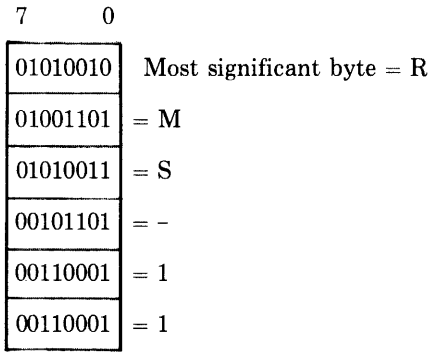
6.2.3 Key Data Type

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

String

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:



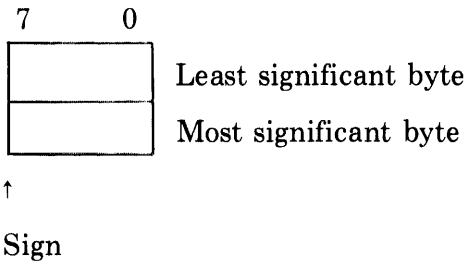
The first (lowest-addressed) byte of the key is the most significant byte of a string key for collating purposes. RMS-11 compares Primary Keys byte-by-byte first-to-last when it determines where the record should be placed in the file.

The maximum key value is all bits in each byte set to 1 (377_8).

Cost The number of bytes specified as size. 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.

Two-Byte Signed Integer

Each key requires two bytes; RMS-11 interprets the data in the following format:



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*.

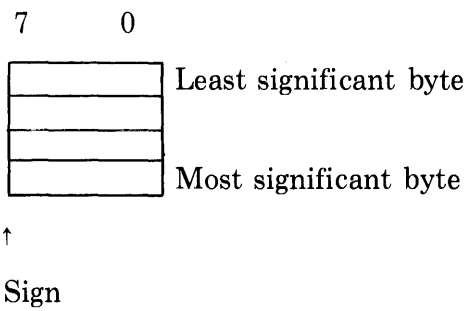
A two-byte signed integer can represent the decimal values -32,768 through +32,767.

Maximum key value is +32,767.

Cost Two bytes per representation.

Four-Byte Signed Integer

Each key requires four bytes; RMS-11 interprets the data in the following format:



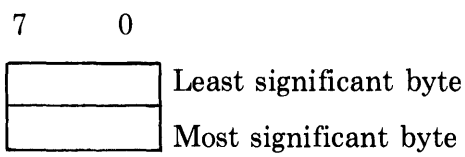
A four-byte signed integer can represent the decimal values -2,147,483,648 through +2,147,483,647.

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

Cost Four bytes per representation.

Two-Byte Unsigned Binary

Each key requires two bytes; RMS-11 interprets the data in the following format:



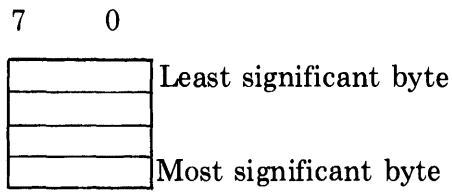
A two-byte unsigned binary can represent the decimal values 0 through +65,535.

Maximum key value is 65,535.

Cost Two bytes per representation.

Four-Byte Unsigned Binary

Each key requires four bytes; RMS-11 interprets the data in the following format:



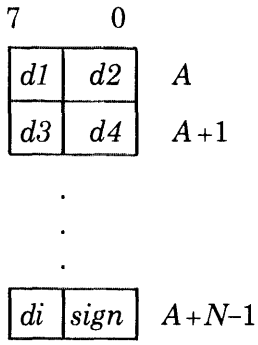
A four-byte unsigned binary can represent the decimal values 0 through +4,294,967,295.

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

Cost Four bytes per representation.

Packed Decimal

RMS-11 recognizes two decimal digits of the key in each byte except the last. The key format takes the following form:



where:

d1, *d2*, through *di* are decimal digits, and

d1 is the most significant digit
di is the least significant digit

sign has a value 10 through 15 where

+ is represented by a 10, 12, 14, or 15
- 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 1 through 31, where

$i = 2N - 1$

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

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[1].

You benefit from careful placement of keys within the record:

[1] PDP-11 COBOL, in keeping with the ANSI standard, does not permit more than one key to start at the same position. The standard calls it *leftmost correspondence*.

Deleting a Record

When you allow duplicates in the Primary Key of variable 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 record deletion 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 less overhead remains 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.

Putting a Record

You can optimize put operations for variable-length records only, 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.

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 the RMSDEF utility do not have to align with the data fields you define within the records.

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

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(0)	Alternate(1+)
Duplicates	Yes or No	Yes or No
Changes	No	Yes or No
Null Key	No	Yes or No

The combination of changes and duplicates is also restricted by key number:

	Combination			
Key Type	CHG + DUP	CHG + NODUP	NOCHG + DUP	NOCHG + NODUP
Primary(0)	Error	Error	Allowed	Allowed
Alternate(1+)	Allowed	Error	Allowed	Allowed

NOTE

PDP-11 COBOL allows the CHG+NODUP combination for Alternate Keys. To enable this option, the PDP-11 COBOL OTS uses a hidden find operation to check on duplicates each time an Alternate Key value changes on an update operation (REWRITE in PDP-11 COBOL).

Duplicates — If duplicates are allowed in a key, more than one record can have the same value in that key field.

Cost File Space

Duplicates have little affect 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 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. However, this benefit becomes apparent only in large files with long string keys.

Putting a Record

RMS-11 stores records with duplicate key values for first-in, first-out access. Putting (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 two-byte indicator. However, if you allow duplicate values for the Primary Key, RMS-11 keeps enough of the record to contain the entire Primary Key:

- If the format is fixed, the entire record remains in the file.
- 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. RMS-11 never checks an array to see if all records have been deleted.

Updating a Record

If you allow duplicate values on the Primary Key, the length of a variable record cannot be changed during an update operation. Also, 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 definitely not good key fields.

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 an 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. But if the data does not change, RMS-11 does not touch the Alternate index.

Null Key — You can specify a null key value 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 (0) is the null key character for the numeric key data types (integers, binaries, and packed decimal). The null key character for string keys can be any octal value (000_8 through 377_8) including an ASCII character.

Cost The use of null key value can reduce the disk space 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 physical volume

Areas allow you to gather logical elements of the file into groups of continuous ranges of Virtual Block Numbers (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 does **not improve** performance.

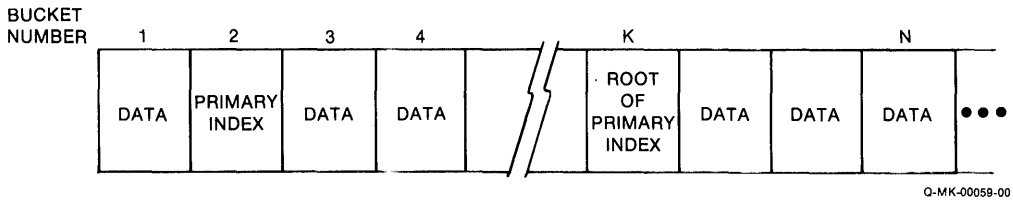
Areas can be set up for:

- Primary index Level 0 (the data records)
- Primary index Level 1 (the lowest index level)
- Primary index Levels 2+ (the rest of the index)
- Alternate index Level 0 (SIDRs)

- Alternate index Level 1 (the lowest index level)
- Alternate index Levels 2+ (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+, 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 6–1: Single-Area Indexed File



Example To randomly access a specific record in the single-area Indexed 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 multi-area 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 no or little 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 6–2: Example of Single-Area Indexed File

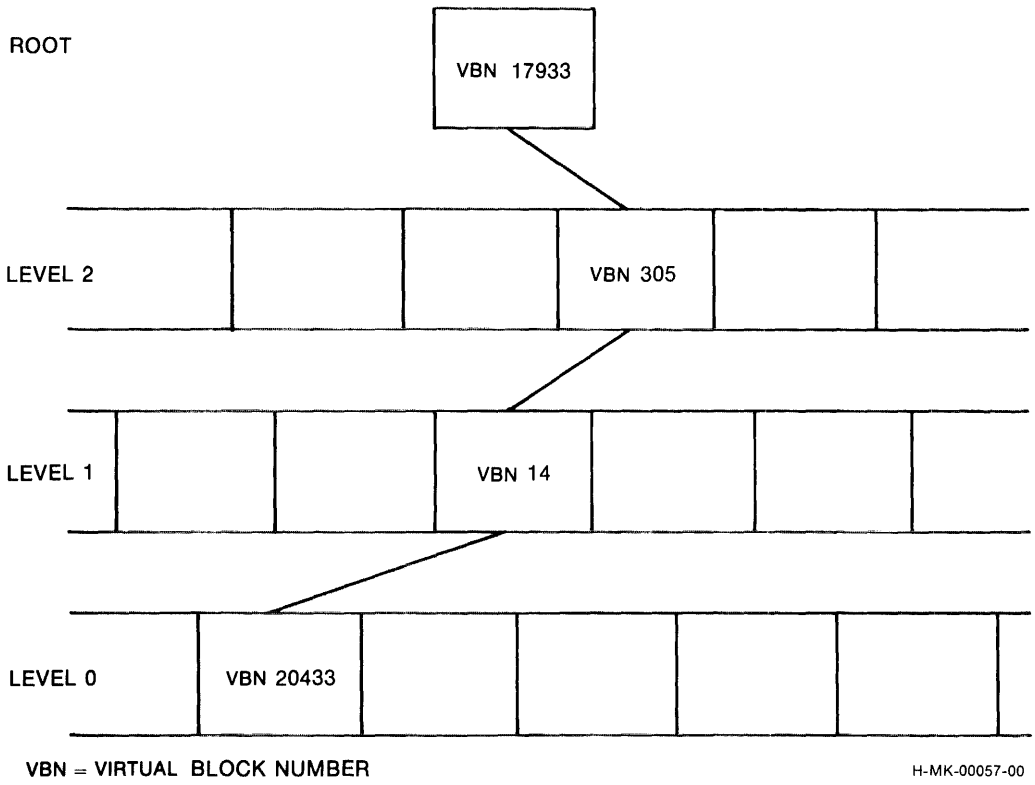
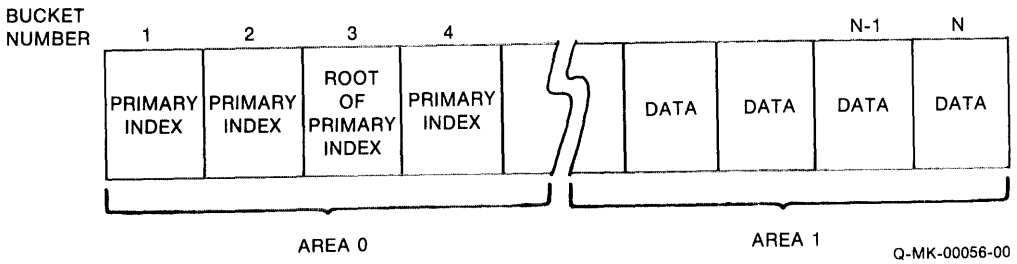


Figure 6–3: Two-Area Indexed File



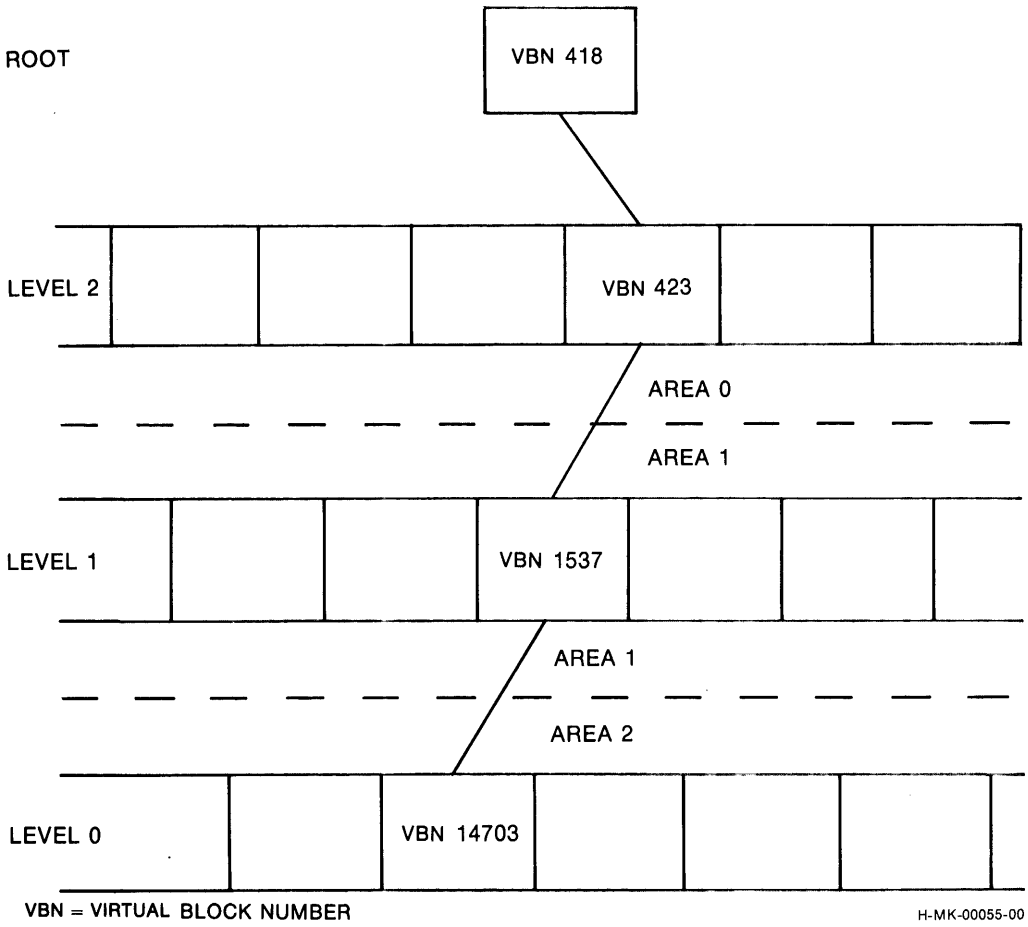
To refine your file even more, place the lowest level of each index (Level 1) in an area separate from the rest of the index (Levels 2+).

Example To randomly access a specific record in the multi-area Indexed 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 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 multiple areas, initially RMS–11 creates the file as Area 0. After the file processor signals successful creation, RMS–11 explicitly extends the file for each extra area specified. As a result, files with more than one area cannot be created as contiguous files, though each area can be contiguous within itself. However, this lack of contiguity may be in name only. Operating systems usually try to allocate each extension of a file as near the end of that file as possible. Therefore, if there are enough unused blocks after the end of Area 0, all extra areas are allocated contiguously: the file comprises a consecutive series of logical blocks.

Contiguity is very important to performance. See further discussion in "Initial Allocation," Section 6.6.1, and in "Virtual-to-Logical-Block Mapping," 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 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 the 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.

   Example The following decimal numbers apply to an RP06:

   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 and 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

NOTE

On RSTS/E, you specify Device Cluster Number (DCN) for placement control. Therefore, you must make one more calculation in this procedure: divide the starting LBN by the device clustersize to get the DCN containing that LBN. If you get a nonzero remainder, use a DCN one higher than your result. See the *RSTS/E User's Guide* for a table of device clustersizes.

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 may be the whole file, if you are not using multiple areas) at the location you calculated.

Then, for Areas 1+, use VBN specifications to place them near the end of the file as it exists at that time:

- Area 1 near the end of Area 0
- Area 2 near the end of Area 1
- and so on.

The VBNs specified are the running sum of the allocation quantities for the areas.

Example You've divided an Indexed file as follows:

Primary data level in Area 0 (also includes the Prologue)
Primary index levels in Area 1
Alternate Key 1 data levels in Area 2
Alternate Key 1 index levels in Area 3

Using the following allocation quantities for those areas, you can calculate the last VBNs for Areas 0 through 2, so you can start the next area near them:

Area	Allocation (in blocks)	Start near VBN
0	15,467	N/A
1	210	15467
2	1,342	15677
3	89	17019

RSTS/E automatically places Areas 1+ as near the end of Area 0 as possible.

If you are using a higher level language, you specify placement control via the RMSDEF utility. If you are programming in MACRO-11, you 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 it 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. Your operating system limits bucket size as follows:

Operating System	Maximum Bucket Size
IAS	32 blocks
RSTS/E	15 blocks
RSX-11M	32 blocks

- The larger the bucket, the more records fit in it, 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 opens 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 "Task Building with RMS-11 Routines," Section 8.1).

Therefore, you should set bucket size to some lower value that still allows good performance; a reasonable goal is an index depth of three (Root at Level 3), though very large files can require four levels of index.

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 seven-byte header) when you calculate Primary data bucket size:

  - Records that move from one bucket to another leave a seven-byte pointer.

  - Deleted records leave from two 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 "Population Techniques," 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.

- In choosing a bucket size, you should consider the file clustersize. Since file clustersize governs file extension quantities and the way the RSTS/E file processor handles disk read operations, sometimes you can improve performance by aligning buckets with clusters. RMS–11 makes this step more reasonable by increasing the size of the file Prologue to an integral multiple of bucket size when:
  - you define the same bucket size for all areas
  - that bucket size is two, four, or eight blocks

6.5.1 Bucket Size for Primary Index

You can calculate bucket sizes with the following steps:

1. Calculate the following quantities for different bucket sizes (1, 2, 3, and so on):

$$NIRBK = \frac{(512*BKS)-15}{PKL + BPL} \qquad \text{(Equation 1a)}$$

and

$$NDRBK = \frac{((512*BKS)-15) - AO}{RSZ + RFO} \qquad \text{(Equation 1b)}$$

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 = 3 for pointers to the first 65,535 blocks in file

BPL = 4 for pointers to the blocks in file numbered between 65,536 and 2^{24}–1

BPL = 5 for pointers to the blocks in file 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 = 7 bytes for fixed-length records
RFO = 9 bytes for variable-length records

AO is activity overhead:

AO = (7*NRI) + (DO*ND)

where:

NRI is the number of data records you estimate will be inserted randomly over the life of the file

DO is the deletion overhead, the number of bytes not reusable in a bucket after RMS-11 deletes a record.

When you allow no duplicates for the Primary Key:

DO = 2

When you allow duplicate values for the Primary Key:

- For fixed-length records, DO = RSZ
- For variable-length records,

  DO = 2 + position of last byte of Primary Key + 1

ND is the number of data records you estimate will be deleted over the life of the file.

If you cannot predict values for the components of AO, use a value at least 5% of bucket size.

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

NRF/NDRBK is the number of Primary index data buckets

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 "Population Techniques," Section 6.7, in this chapter.
- Bucket size is a step function of index depth. Therefore intermediate bucket sizes generally waste address space.

Example Given a file in 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.

Example A file of 50,000 200–byte fixed-length records with a 15–byte Primary Key. We use the equations in steps 1 and 2 to fill in the following grid:

1. Calculate values for Number of Index Records per BucKet (NIRBK) using Equation 1a and bucket sizes 1 through 6. Drop the remainder; use only the integer part of the result.
2. Calculate values for Number of Data Records per BucKet (NDRBK) using Equation 1b and bucket sizes 1 through 6. Drop the remainder; use only the integer part of the result.
3. 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.
4. Calculate NIRBK squared (NIRBK**2) for the values of NIRBK corresponding to bucket sizes 1 through 6. Round the result up to the nearest integer.

5. Calculate NIRBK cubed (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

Now we determine the combinations of bucket sizes where Equation 2 is true:

6. 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, we have a valid bucket size combination.

   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.

7. 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, we have a valid bucket size combination.

   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 these comparisons, Equation 2 was true in the following cases:

	NIRBK**3			NIRBK**2	
DBKS	IBKS	IOB (bytes)	DBKS	IBKS	IOB (bytes)
1	2	2048	1	6	6144
2	1	2048	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 buffer 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:

1. The records-per-bucket equations for Alternate indexes are:

$$NIRBK = \frac{(512*BKS)-15}{AKL + BPL}$$

and

$$NDRBK = \frac{((512*BKS)-15)*DF}{AKL + (DBPL*DF) + 4 + DO}$$

where:

AKL is the Alternate Key length in bytes

DBPL is the data bucket pointer length where no duplicates are allowed:

DBPL = 4 for pointers to the first 65,535 blocks in file

DBPL = 5 for pointers to the blocks in file numbered between 65,536 and $2^{24}-1$

DBPL = 6 for pointers to the blocks in file numbered between 2^{24} and $2^{32}-1$

DF is the duplicate factor:

DF = 1 if you allow no duplicates

DF = average number of records with 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 = 0 if you allow no duplicates
DO = 4 if you allow duplicates

No record movement or space/deletion overhead occurs in index buckets.

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 Chapter 9). The ideal values resulting from the equations in "Bucket Sizes for Primary Index," Section 6.5.1, must be reduced by a packing efficiency factor.

   Studies have shown that the factor for Alternate Keys is normally about 0.5. However, this factor applies only to the lower levels of the index, and not to the Root. The packing efficiency of any index's Root is always one.

   Therefore, the index depth equation for Alternate indexes is:

   $(0.5^{**}n)*(\text{NIRBK}^{**}n) \leq \text{NRF/NDRBK}$

Example Using the file in the Primary Key example and adding a ten-byte First Alternate Key, allowing no duplicates, we fill in the following grid (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:

	NIRBK**3			NIRBK**2	
DBKS	IBKS	IOB (bytes)	DBKS	IBKS	IOB (bytes)
1	1	1024	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 higher 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 two blocks and passes that to RMS-11 to create the file.

However, two blocks contain 1024 bytes; that leaves 1009 bytes for record storage after the bucket overhead is subtracted.

1009/107 = 9 records per data bucket

Therefore, the buckets which were originally supposed to contain only five records now can contain nine.

Bucket size is set by RMSDEF or by your application program:

MACRO-11

- Single-area files

  Use the initialization macro or $STORE to set the BKS field in the FAB of the file to the chosen number of blocks before issuing $CREATE.

- Multi-area files

  Use the initialization macro or $STORE to set the BKZ field in the Allocation XAB for each area to the chosen number of blocks before issuing $CREATE.

NOTE: The default value is one block per bucket.

BASIC-PLUS-2
Use the BUCKETSIZE clause in the OPEN statement that creates the file.

PDP-11 COBOL
In the file-description-entry (FD), use the BLOCK CONTAINS clause.

RPG II
Use the RPGASN utility to override the default value set by the compiler.

DIBOL
Use RMSDEF; DIBOL does not create Indexed file.

6.6 Allocation

RMS-11 requests the file 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 = 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 "Bucket Size for Primary Index," Section 6.5.1 for the method of determining this value), and

   NBK@0 is the number of buckets in Level 0

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 Section 6.5.1 NDRBK value by two. You will need twice as many data buckets.

2. Calculate the number of buckets in Level 1:

   NBK@1 = NBK@0/NIRBK

   where:

   NBK@1 is the number of buckets in Level 1, and

   NIRBK is the number of index records per bucket in the index (see "Bucket Size for Primary Index," Section 6.5.1).

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 Section 6.5.1 NIRBK value by two 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@0 is the allocation quantity in blocks for Level 0

   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 "Bucket Sizes for Alternate Indexes," Section 6.5.2, for equations.

NOTE

Alternate indexes are normally populated in random key value order. Therefore, you should divide the Section 6.5.2 NDRBK and NKDBK values by two 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 + ... + AQAK*n*

   where:

   n is the last Alternate Key defined for the file.

NOTE

Alternate indexes are normally populated in random key value order. Therefore, you should divide the Section 6.5.2 NDRBK and NIRBK values by two except for the Root level.

The Prologue of an Indexed file can be from two 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 five Alternate Keys.
- Each Area Descriptor block covers up to eight 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 = Index 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:

1. NDRBK = ((512*3)-15)/(8 + (4*1) + 4) DF = 1
   = 89 data records per bucket DO = 0

   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 Alternate index

Finally:

ALQ = PLG + AQPK + AQAK1 = 3 + 43,518 + 6,849
= 50,370 blocks for the whole file

This allocation can be done by the RMSDEF utility or by your application program as follows:

MACRO-11

- Single-area files

  Use the initialization macro or $STORE to set the ALQ field in the FAB of the file to the chosen number of blocks before issuing $CREATE.

- Multi-area files

  Use the initialization macro or $STORE to set the ALQ field in the Allocation XAB for each area to the chosen number of blocks before issuing $CREATE.

BASIC-PLUS-2
Use the FILESIZE clause in the OPEN statement that creates the file.

PDP-11 COBOL
Use the /AL:*n* switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
Use the RPGASN utility to override the default value set by the compiler.

DIBOL
Use RMSDEF; DIBOL does not create Indexed files.

NOTE

If possible, you should allocate an Indexed file contiguously. To ensure that the entire file is logically contiguous, allocate it all when you create it and then copy it (if you used multiple areas) into a contiguous file with PIP.

On IAS/RSX-11M, you can achieve the same result by:

- calculating the starting LBNs for each area (see "Placement Control," Section 6.4)
- requiring the operating system to place those areas as directed. The RMSDEF utility enables you to specify starting LBNs and that you want the creation to fail if the system cannot allocate the blocks in the places you indicated.

See "Virtual-to-Logical-Block Mapping," Section 8.3, for the impact of extending a contiguous file on RSTS/E.

MACRO-11
Use the initialization macro or $SET to set the FOP field in the FAB of the file to include FB$CTG before issuing $CREATE for a single-area file. However, if you are using an Allocation XAB to place one area, you set the XB$CTG value in the AOP field of the Allocation XAB.

BASIC-PLUS-2
Specify CONTIGUOUS in the OPEN statement that creates the file.

PDP-11 COBOL
Use the /CO switch on the File ID.

RPG II
Use RMSDEF; RPG II does not create contiguous files.

DIBOL
Use RMSDEF; DIBOL does not create Indexed files.

6.6.2 Default Extension Quantity

If the file cannot be totally allocated at creation, 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 RMSDEF or a higher level language. RMS-11 responds to a DEQ of zero by requesting four times bucket size in blocks from the file processor each time it automatically extends the file.

The DEQ for the file can be set by the RMSDEF utility or by your application program as follows:

MACRO-11

- Single-area files

  Use the initialization macro or $STORE to set the DEQ field in the FAB of the file to the chosen number of blocks before issuing $CREATE.

- Multi-area files

  Use the initialization macro or $STORE to set the DEQ field in the Allocation XAB for each area to the chosen number of blocks before issuing $CREATE.

  The DEQ field in the FAB serves as run-time file DEQ; that is, RMS-11 uses it for any area whose DEQ is zero.

BASIC-PLUS-2
: Use RMSDEF; BASIC-PLUS-2 does not support DEQ specifications.

PDP-11 COBOL
: Use the /EX:*n* switch on the file specification in the ASSIGN clause or the VALUE OF ID.

RPG II
: Use the RPGASN utility to override the default value set by the compiler.

DIBOL
: Use RMSDEF; DIBOL does not create Indexed files.

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 RMSIFL, RMSCNV, or an application task. 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:

- ascending order by Primary Key
- fill number

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**$n \geq$ NRF/NDRBK

But when $p < 1$, the equation becomes:

(p**n)(NIRBK**n) $\geq$ 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 "Bucket Sizes for Alternate Indexes," 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. The utility:
  - sorts your input file into ascending order by the output file's Primary Key, if it 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 Insert (MA) switch.
- Write a MACRO-11 program to populate the file and specify:
  - in the FAB, Deferred Write (FB$DFW in FOP field) when you open the file
  - in the RAB when you connect to the file:
    - Mass Insert (RB$MAS in ROP field)
    - Sequential Access Mode (RB$SEQ in RAC field)

  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 inserted records have the full range of Primary Key values, you should use fill numbers.
- If the inserted records are sorted into ascending Primary Key value and added at the logical end-of-file, use Mass Insert.

6.7.2.1 Fill Number — 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 fill number 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 two blocks; you set the fill number to 768 bytes. When you tell RMS-11 to obey the fill number, it only uses 768 out of 1024 bytes in each bucket—the buckets are logically three-quarters size.

You use the fill number 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 a fill number 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 inserts might not warrant the use of fill numbers, 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 fill numbers.

Example The file contains 240-byte fixed-length records with a Primary Key field 24 bytes long. To optimize random insertions, the fill number for data buckets should therefore be at most:

```
    bucket-length
less bucket overhead (15)
less 247
```

This number leaves room for one data record.

This same fill number for index buckets leaves room for nine index records. A more reasonable fill number for index buckets is:

```
    bucket-length
less bucket overhead
less 2 times 27 bytes
```

This number leaves room for two index records, where:

```
Index record = Primary Key Length + Bucket Pointer Length
             = 24 + 3 = 27
```

See "Bucket Sizes for Alternate Indexes," Section 6.5.2, for a more complete discussion.

NOTE

RMS-11 ignores a fill number less than 50% of the bucket length and uses the 50% figure.

The fill number for a file can be set by the RMSDEF utility or by your application program as follows:

MACRO-11
: Use the initialization macro or $STORE to set the DFL and IFL fields in the key XABs to the chosen fill number before issuing $CREATE.

BASIC-PLUS-2
: BASIC-PLUS-2 does not support fill number specifications.

PDP-11 COBOL
: PDP-11 COBOL does not support fill number specifications.

DIBOL
: DIBOL does not support fill number specifications.

RPG II
: RPG II does not support fill number specifications.

You use fill numbers when you populate a file by:

- the RMSCNV or RMSIFL utility with the /LO switch specified.
- a MACRO-11 program where the value RB$LOA is set into the ROP field of the active Record Access Block.

Most higher level language tasks cannot cause RMS-11 to obey fill numbers.

6.7.2.2 Mass Insert — You use Mass Insert 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 bit is on, RMS-11 performs a put operation normally (see "Putting a Record," 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 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 gets 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 Insert 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 Insert performance by using Deferred Write (see "I/O Techniques," Section 7.1.3).

Chapter 7
Indexed Task Design

The record and file processing capabilities described in Chapter 1 are available for Indexed files. This chapter discusses the operations and their implementation and restrictions with Indexed files.

7.1 Record Operations

RMS-11 performs a record operation at the request of a program. See also the discussions of read- and write-type record operations in Chapter 5. The available operations include:

Connect
Delete
Disconnect
Find
Flush
Get
Put
Rewind
Update

In all record operations, RMS-11 establishes Current Record (if any) and Next Record (if applicable). If any record operation fails, RMS-11 normally sets Current Record to NONE and does not change Next Record. "Record Access Stream," Section 1.3.1.5, introduces the concepts of Current Record and Next Record.

7.1.1 Connect

A connect operation affects the context for the Record Access Stream as follows:

Current Record
: There is no Current Record. Any operation requiring 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.

7.1.2 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 (discussed in "I/O Techniques," Section 7.2).

A delete operation affects the context for the Record Access Stream as follows:

Current Record
None. Any operation requiring a Current Record fails at this point.

Next Record.
Unchanged.

7.1.3 Disconnect

A disconnect operation destroys the context for the Record Access Stream. You cannot resume this context by reconnecting the stream.

7.1.4 Find

"Getting and/or Finding a Record," Section 5.3.2, describes how RMS-11 performs a 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$EOF.
- In Random Access Mode, the error code is ER$RNF.
- In Access by RFA, 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 Record Access Stream as follows:

- find in Sequential Access Mode:

  Current Record
  Set to value of the record found.

  Example You've 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 find operation, the Current Record is set to this record.

  Next Record
  Set to record logically following Current Record in 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, Next Record is the record in the file with the next higher Primary Key value.

- find in Random Access Mode or Access by RFA:

  Current Record
  Set to the record found, that is, the record identified by the Record's File Address.

  Next Record
  Unchanged.

  Example In the previous examples, you did a sequential find 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, if you do another sequential find, Current Record will be set to the second record in Primary Level 0, **not** the record following the one found by RFA.

You use find instead of a get operation because:

- it is quicker 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.
- it does not change Next Record in Random Access Mode or Access by RFA. This convention allows you to branch off sequential processing for purpose of updating or deleting, and yet 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 in Random Access Mode.

7.1.5 Flush

"Records Operations," Section 1.2.4, summarizes the flush operation.

A flush operation does **not** affect the context for the Record Access Stream.

7.1.6 Get

"Getting and/or Finding a Record," Section 5.3.2, describes how RMS-11 performs a 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$EOF.
- In Random Access Mode, the error code is ER$RNF.
- In Access by RFA, 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 Record Access Stream as follows:

- get in Sequential Access Mode **not** immediately preceded by a successful find operation:

  Current Record
  : Set to value of the record read. See "Find in Sequential Access Mode" for example.

  Next Record
  : Set to record logically following Current Record in index of reference. See note and example in "Find in Sequential Access Mode."

- get in Sequential Access Mode immediately preceded by a successful find operation:

  Current Record
  : Unchanged (from Current Record set by find operation).

Next Record
Set to record logically following Current Record in index of reference (possibly changing Next Record set by find operation).

NOTE

A find in Access by RFA changes the key of reference to the Primary Key. But the change does not become apparent unless you follow the find by a sequential get operation. In this case, the Current Context for the stream is affected as follows:

Current Record
Unchanged from Current Record set by find operation.

Next Record
Set to record logically following Current Record **in the Primary Key sequence**.

A get by RFA has the same affect.

- get in Random Access Mode

Current Record
Set to record specified, that is, the record read.

Next Record
Set to record logically following Current Record in index of reference.

- get in Access by RFA:

Current Record
Set to record specified by Record's File Address.

Next Record
Set to record logically following Current Record **in the Primary index**. This differs from find by RFA which does not change Next Record.

7.1.7 Put

"Putting a Record," Section 5.3.1 describes how RMS–11 performs a put operation.

A put operation affects the context for the Record Access Stream as follows:

- put in Sequential Access Mode:

Current Record
None. Any operation requiring a Current Record fails at this point.

Next Record
Undefined. The record retrieved by a sequential find or get at this point is not specified.

- put in Random Access Mode:

  Current Record
  None. Any operation requiring a Current Record fails at this point.

  Next Record
  Unchanged.

7.1.8 Rewind

A Rewind operation sets the context of the Record 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
Set to first record in file according to the specified key of reference.

7.1.9 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 the prerequisite get or find operation. Then, RMS-11 writes the bucket over its original location on the disk, unless you specified Deferred Write (described in "I/O Techniques," Section 7.2). "Updating A Record, Section 5.3.3," describes the update operation in detail.

An update operation requires a valid Current Record. Therefore an update 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 Record Access Stream as follows:

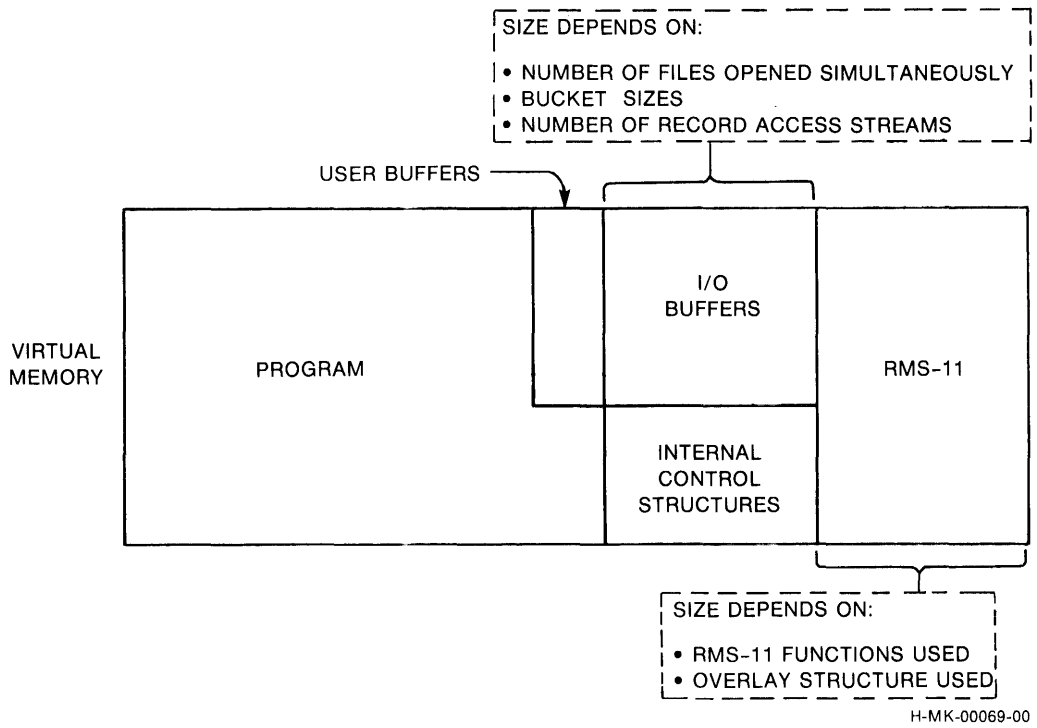
Current Record
None. Any operation requiring a Current Record fails at this point.

Next Record
Unchanged.

7.2 Record Transfer Modes

You can manipulate records either in the I/O buffer or in your program's user buffer (see Figure 7-1). 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: RMS-11 Task Structure



7.2.1 Move Mode

Move Mode is the default Record Transfer Mode for all programming languages on 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.

Move Mode is the default Record Transfer Mode for all programming languages on all file organizations.

7.2.2 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 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 get records in Locate Mode. See your language documentation to determine if the language supports Locate Mode and if it does, what the exact programming techniques are.

7.3 I/O Techniques

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

7.3.1 IAS/RSX-11M 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 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 language documentation for asynchronous techniques.

NOTE

If you intend to use asynchronous RMS-11 record operations and/or Asynchronous System Traps (ASTs) in other parts of your program, see the section on your operating system in Appendix A.

7.3.2 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 may improve the performance of sequential (by Primary Key) write-type 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.

NOTE

Deferred Write, although not illegal, is essentially invalidated while an Indexed file is shared by multiple tasks or streams—except when you are also using Mass Insert. In the non-Mass-Insert, 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.

Example Your records are 114 bytes long and the bucket size is two blocks. During sequential write-type operations, Deferred Write could cause I/O operations per bucket to drop from nine to one.

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

Only your application program can specify Deferred Write:

MACRO-11
Use the initialization macro or $SET to set the value FB$DFW in the FOP field of the FAB of the Indexed file before initiating the $CREATE or $OPEN operation.

BASIC-PLUS-2
BASIC-PLUS-2 does not support Deferred Write.

PDP-11 COBOL
PDP-11 COBOL does not support Deferred Write.

RPG II
RPG II does not support Deferred Write.

DIBOL
DIBOL does not support Deferred Write.

7.3.3 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 Insert 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).

7.3.4 Multiple Record Access Streams

RMS–11 allows each program to use from one to 255 streams on an Indexed file.

7.3.5 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 following get operation, though it does not start at the Root and go down the index again.

7.4 File Operations

You can perform the following file operations on Indexed files. File operations do not involve records and can only perform synchronously.

7.4.1 Close

A close operation disconnects all Record Access Streams connected to a file before it releases access.

7.4.2 Create

In addition to the file specification, RMS-11 passes the following information to the file processor when it creates a file:

- An initial allocation of blocks for each area in the file. You specify both the areas and their allocations in your instructions to RMS-11.
- The location on a specific device where the processor should allocate those blocks. You also supply this information.
- The following file attributes:

  File organization
  Record format
  Forms control
  Record size
  Number of virtual blocks in the file
  Bucket size
  Default extension quantity

  The other file attributes, such as key and area descriptions for Indexed files, in the Prologue of the file.

7.4.3 Open

You can specify the file you want to open in two different ways:

By filespec
: The first time you open a file, you must use the file specification (see Appendix A).

By File ID
: When you create or open a file by filespec, RMS-11 returns an identifying notation to your program. You can store this File ID, either in memory or in a file, and use it to open the file from that point on.

On IAS/RSX-11M systems, open by File ID is significantly faster than open by filespec, because the process bypasses directory reads and other overhead. However, on RSTS/E, open by File ID is no faster than open by filespec.

7.4.4 Erase

"File Operations," Section 1.2.5.3, introduces the concept of erasing files.

7.4.5 Extend

"File Operations," Section 1.2.5.3, introduces the concept of extending files.

Chapter 8
Common Optimization Techniques

Chapter 2 introduced four application design considerations. Two, 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 sequence of operations (discussed in Section 8.2)
- using all features of the environment where the task runs. Especially important is optimization of virtual-to-logical-block mapping (discussed in Section 8.3), but there are other factors (discussed in Section 8.4).

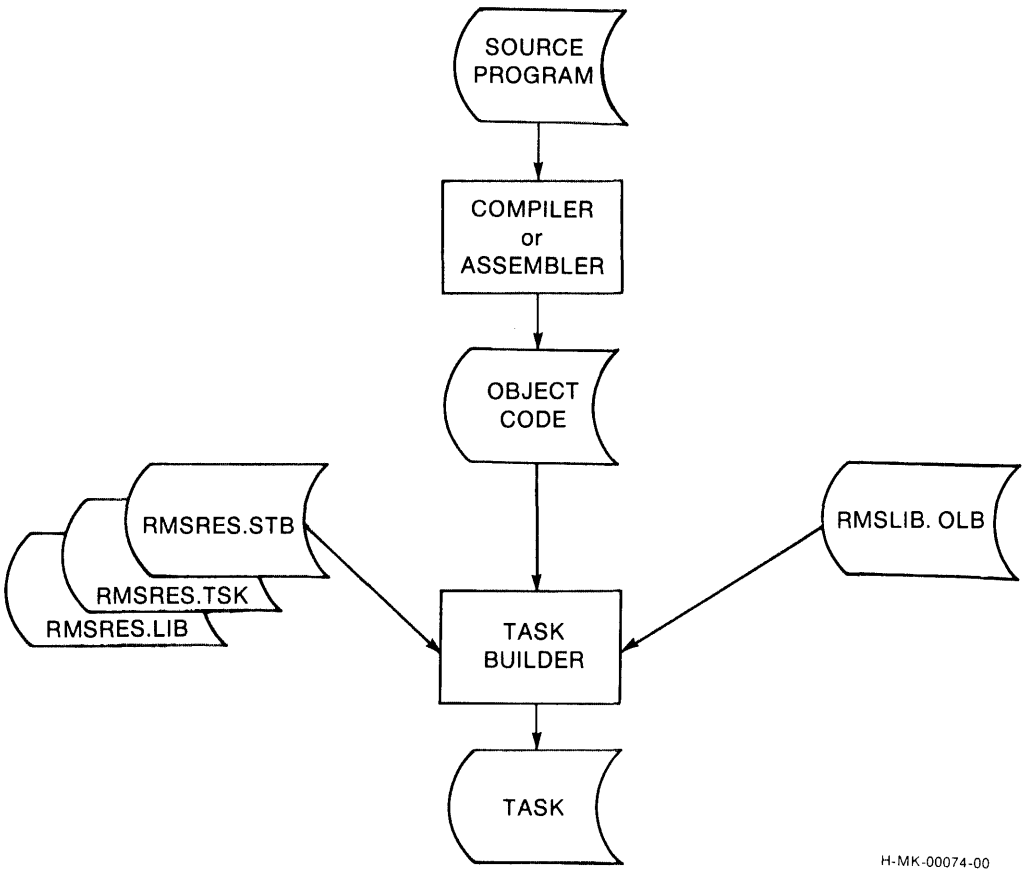
8.1 Task Building with RMS-11 Routines

The software routines that perform the RMS-11 functions are distinct from your programming language. These routines must be combined with your program with the following steps (see Figure 8-1):

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 utility, provided with your operating system, 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 select the form of RMS-11 that is joined with your program to make a task. This section should guide your choice.
3. When the Task Builder is finished, your task is ready to run.

Figure 8-1: Source-to-Task Sequence



The RMS-11 routines that become part of your task can be overlaid or not. *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 — The Task Builder concatenates the RMS-11 routines with your program, that is, without overlays (see Figure 8-2A), if you add the following term to the command line:

```
,LB:[1,1]RMSLIB/LB[1]
```

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.

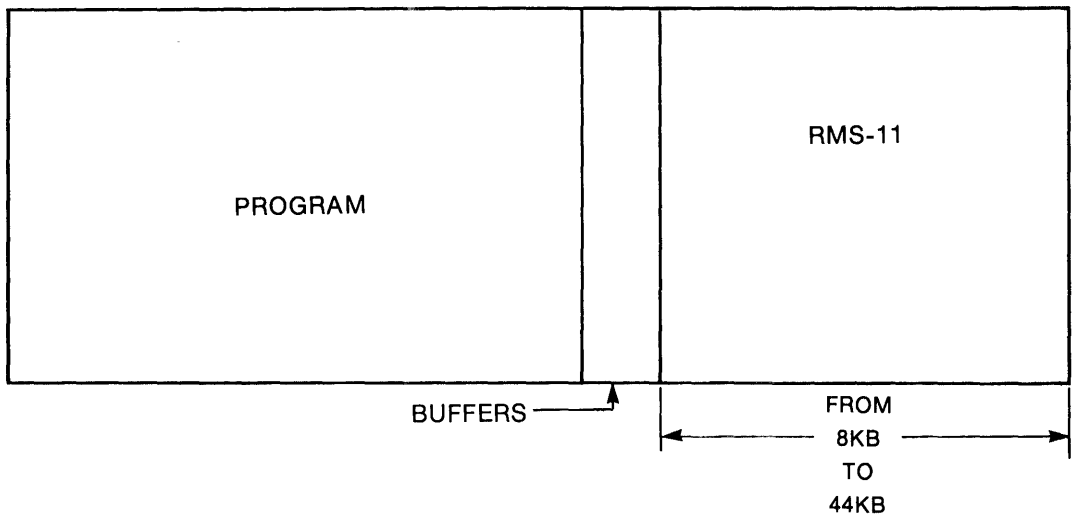
Overlaid RMS-11 — If the sum of your program, including Run-Time System, and RMS-11 code is greater than 64KB, there is not enough user address space on the PDP-11 for your task to run without overlays.

[1]Valid for synchronous operations only. If you are using IAS/RSX-11M asynchronous I/O operations, you must use the following terms:

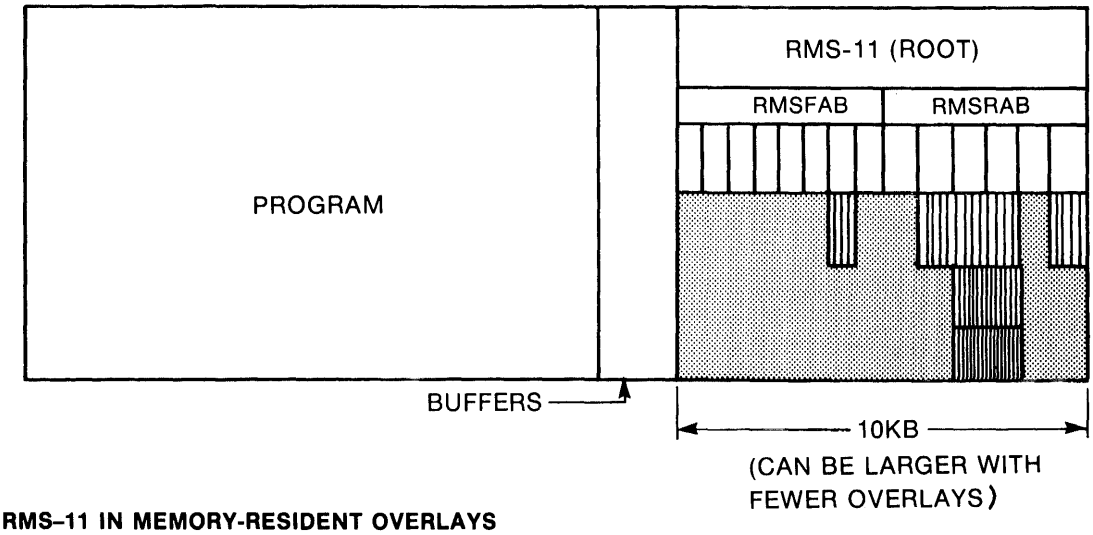
```
,LB:[1,1]RMSLIB/LB:ROEXEC:RORSET:ROWATB,RMSLIB/LB
```

Figure 8-2: RMS-11 in Tasks

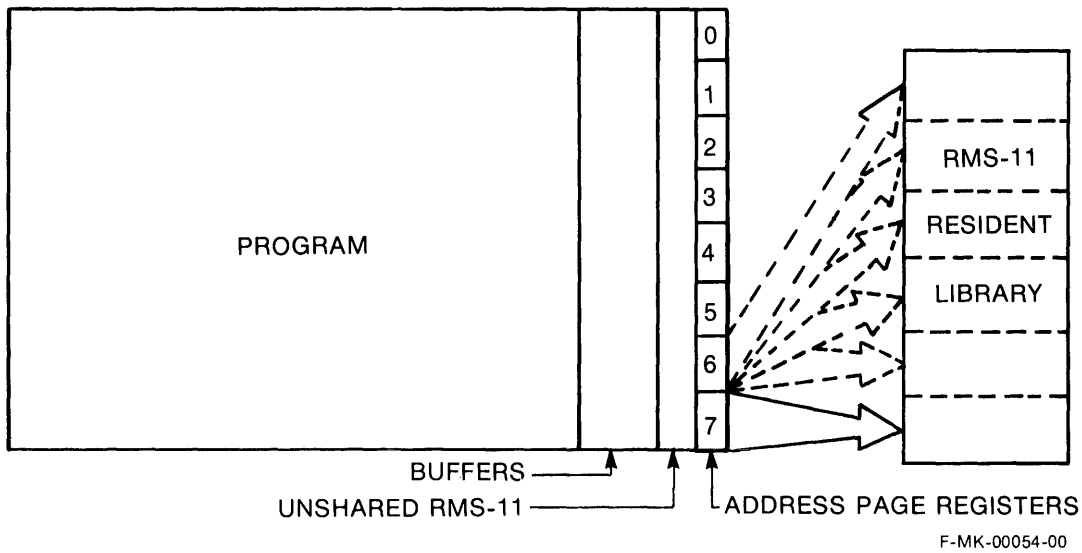
A. NONOVERLAID RMS-11



B. RMS-11 IN DISK-RESIDENT OVERLAYS



C. RMS-11 IN MEMORY-RESIDENT OVERLAYS



F-MK-00054-00

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:

Disk-Resident Overlays
: The overlay segments are part of the task image (see Figure 8-2B), 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.

Memory-Resident Overlays
: The overlay segments are part of a task image (see Figure 8-2C) 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.

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. See Figure 8-3 for a sample overlay structure and an ODL file that describes it.

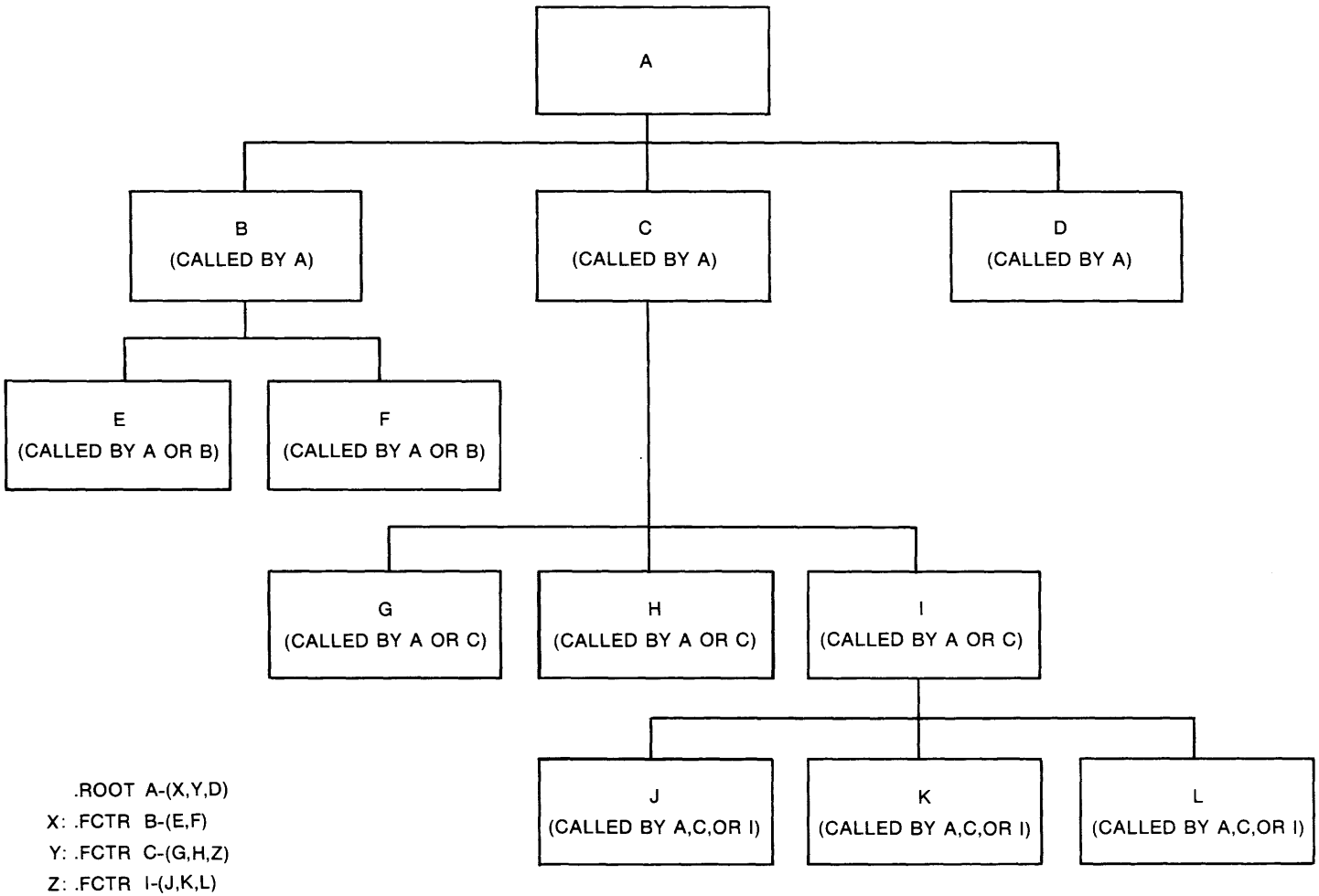
You must generate an ODL file for each overlaid task and supply it to the Task Builder. However, you do not create ODL statements for the RMS-11 portion of your task. The RMS-11 installation process provides overlay descriptions in two forms:

- 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:.

Figure 8–3: Sample Overlay Structure and ODL File



Each higher level programming 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 utility. The file contains the appropriate command line(s) for the utility 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.

See your language documentation for more details.

However, if you are a MACRO-11 programmer, 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. Normally, you should concatenate RMSROT with your program's root so that it is memory-resident while the task runs. However, you can concatenate RMSROT with one of your own overlay segments if both of the following are true:

  - RMSROT is memory-resident before your program opens any RMS-11 files and RMSROT remains memory-resident until all RMS-11 files are closed.

  - Each time RMSROT is brought into memory, your program initializes the RMS-11 control areas with the $INIT macro.

  RMSALL represents the overlay structure of modules that perform RMS-11 operations. RMSALL can be:

  - concatenated with your nonoverlaid program. The RMS-11 overlay structure then becomes the only tree in the task.

  - concatenated with one of your program's overlay segments. RMSROT must be memory-resident also before your segment initiates any RMS-11 operation. This approach avoids the division of your task's address space into separate co-tree areas.

  - set up as a co-tree. Both trees, yours and RMS-11's, have a longest series of overlay segments that must be resident in memory at the same time. This series determines the maximum amount of memory the overlay structure requires. The concatenation of these co-tree maximums must be less than 64KB.

  Example `.ROOT USER-RMSROT-RMSALL`

  RMSROTand RMSALL are concatenated with a nonoverlaid user root.

Example `,ROOT USER-RMSROT-*(RMSALL)`

RMSROT and RMSALL are concatenated with a nonoverlaid user root. An autoload indicator is included for RMSALL, but this indicator is not necessary because the secondary RMS-11 ODL file includes all necessary autoload indicators.

Example `,ROOT USRROT-RMSROT-USRSEG,RMSALL`

RMSROT is concatenated with the user root. RMSALL is defined as a co-tree with the user overlay structure as the main tree. Each tree is segregated in the task's virtual address space.

Example `,ROOT USRROT-RMSROT-(USRSEG,RMSALL)`

RMSROT is concatenated with the user root. RMSALL is overlaid with the user overlay structure. Each tree can use all of the task's virtual address space, less the root, but modules in one tree *cannot* call modules in the other tree. Therefore, no segment in USRSEG can perform RMS-11 operations.

- 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.

Example

```
          ,ROOT    USRROT-RMSROT-USRSEG,RMSALL
USRSEG:   ,FCTR    (USR1,USR2,USR3)
@LB:[1,1]RMS11X
          ,END
```

8.1.1.1 Standard ODL Files — DIGITAL provides the following standard ODL files. Do not change these files.

RMS11S.ODL
: Structured to add a little more than 8KB to task size, this file features only Sequential and Relative file organization routines in nineteen overlay segments.

RMS11X.ODL
: Structured to add a little more than 9KB to task size, this file features Sequential, Relative, and Indexed file organization routines in 57 overlay segments.

RMS12X.ODL
: Structured to add no more than 12KB to task size, this file features Sequential, Relative, and Indexed file organization routines in 17 overlay segments.

  The size of the RMS-11 overlay structure and the number of overlay segments were changed by modifying the RMS11X.ODL overlay structure as follows:

  - File open and Record Access Stream connect routines combined in one overlay segment.

- File close and Record Access Stream disconnect routines combined in one overlay.
- Sequential file record operation routines combined in one overlay.
- Relative file record operation routines combined in one overlay.
- Routines that insert records into Primary Level 0 combined in one overlay.
- Primary Level 0 bucket splitting routines combined into two overlay segments separate from record insertion routines.
- Routines that insert Secondary Index Data Records (SIDRs) into Alternate Levels 0, including bucket splitting, combined into one overlay.
- Routines that insert index records into Levels 1+, including bucket splitting, combined into one overlay.

This modification reduced overlays for Indexed organization routines from 32 to 8.

Appendix C contains copies of ODL files for RMS–11 overlay structures that occupy 16KB and 20KB. You may enter one or both of these files on your system and use them while building a task. Appendix C also contains a graphic illustration of the overlay structures created by RMS11S.ODL and RMS11X.ODL.

8.1.1.2 Prototype ODL Optimization — The prototype 9KB ODL file is named RMS11.ODL. This file is similar to the standard RMS11X.ODL file, except that it contains comments and instructions in addition to ODL statements. You can optimize this overlay structure to accommodate task requirements.

You change an ODL file as follows:

- Combine segments that overlay each other into a single overlay; this change reduces the number of overlays and increases task speed at the cost of task size (the task gets bigger). For example, some RMS–11 record operations, notably the Indexed file put and update operations, require a series of overlay segments in the 9KB overlay structure. You could make these operations run faster by combining some or all of these segments.
- Eliminate modules because your task does not use those functions, and the task's disk file gets smaller though the task image in memory may not change size.

NOTE

Know what you can eliminate, particularly in the ODL files for higher-level-language programs: the compiler can call routines that you are not aware of. For example, BASIC–PLUS–2 tasks use Block I/O for virtual arrays. If you eliminate something the task needs, the Task Builder prints an UNDEFINED SYMBOLS error message.

When you are going to modify the prototype, first copy the file into another file with a different name and/or account: not everyone on your system wants the same optimizations you do, so make them specific to your application.

8.1.1.2.1 Techniques — While you are changing the prototype, obey all Task Builder syntax rules and conventions (see the Task Builder manual for your operating system) and observe the following methods:

NOTE

Under no circumstances should you change or move the modules included in any factor.

Concatenate overlaid modules within a factor one at a time.

Example The factor

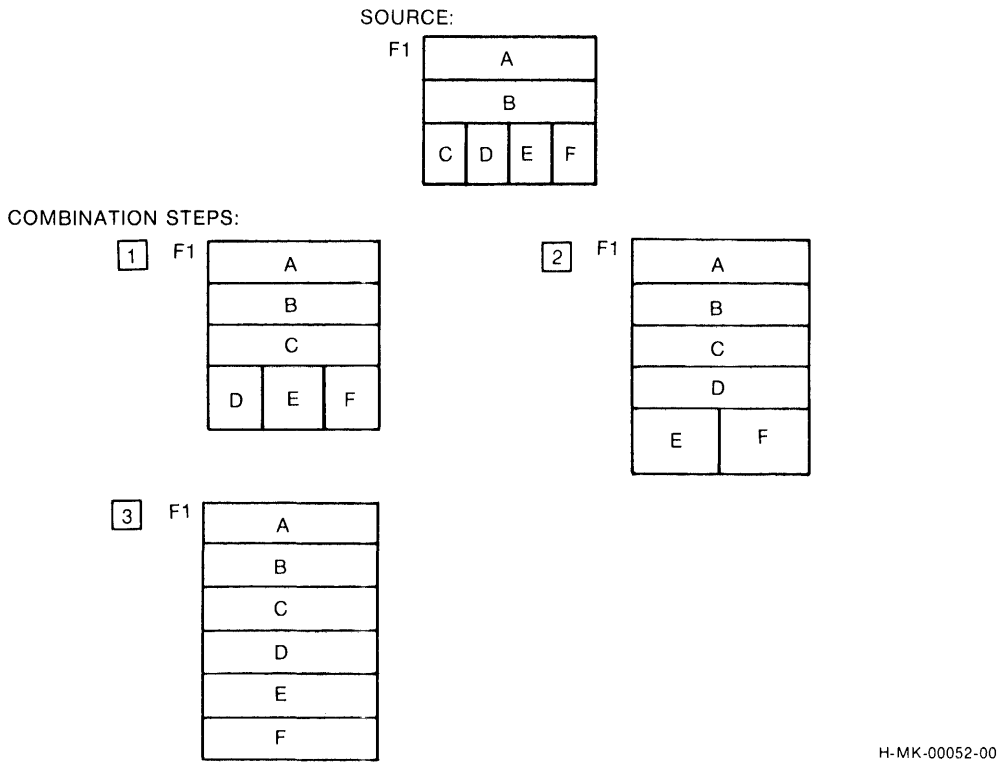
F1: .FCTR A-B-(C,D,E,F)

can be optimized incrementally for performance, with one overlay made memory-resident with A and B in each of the following steps (except the last); see also Figure 8-4.

1. F1: .FCTR A-B-C-(D,E,F)
2. F1: .FCTR A-B-C-D-(E,F)
3. F1: .FCTR A-B-C-D-E-F

Each of these factors is valid and represents a level in the size and speed of the task.

Figure 8-4: Incremental Optimization Example



Combine factors with caution.

- Do not combine around overlays: either concatenate the overlaid modules or put them at the end of the module chain.

Example The factors:

```
F1:  .FCTR A-B-(C,D)
F2:  .FCTR X-Y-(Z,W,R)
```

cannot be combined like this (see also Figure 8–5):

Wrong

```
F1:  .FCTR A-B-(C,D)-F2
F2:  .FCTR X-Y-(Z,W,R)
```

because this implies:

```
F1:  .FCTR A-B-(C,D)-X-Y-(Z,W,R)
```

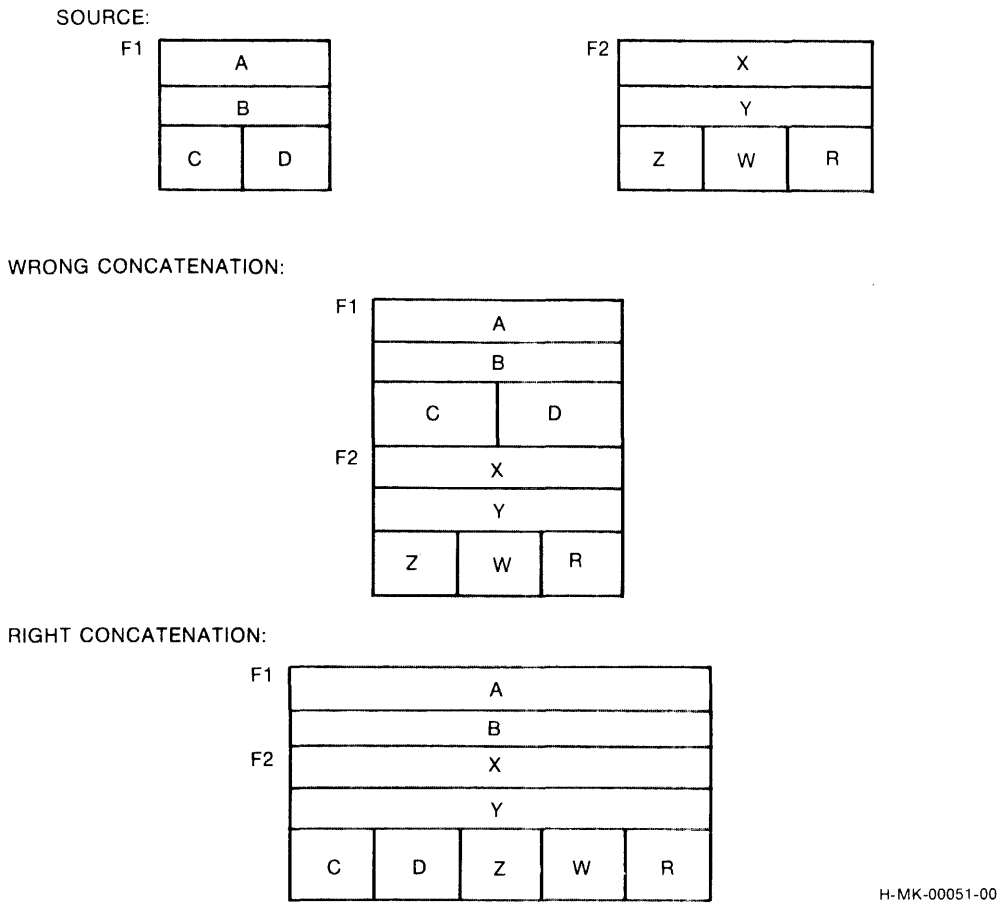
The Task Builder cannot successfully build this task because it would have to combine modules X and Y with modules A and B allowing room for the variable lengths of C and D in between.

Right

```
F1:  .FCTR A-B-F2
F2:  .FCTR X-Y-(C,D,Z,W,R)
```

Note that each original root (A–B and X–Y) is still the root for its associated overlays (C,D and Z,W,R, respectively).

Figure 8–5: Concatenation around Overlays Example



- Remove duplicate module names. Duplicates will arise because some factors use common modules to preserve overlay tree linkages.

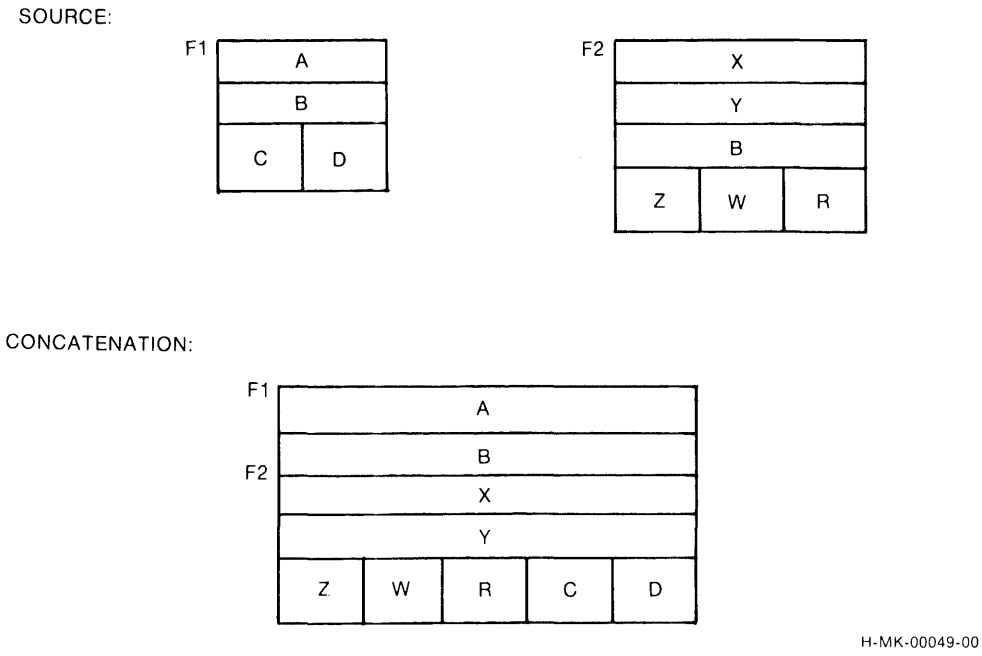
  Example The factors

  ```
  F1:  .FCTR A-B-(C,D)
  F2:  .FCTR X-Y-B-(Z,W,R)
  ```

  are combined in this way (see also Figure 8-6):

  ```
  F1:  .FCTR A-B-F2
  F2:  .FCTR X-Y-(Z,W,R,C,D)
  ```

Figure 8-6: Duplicate Module Example No. 1



Example The factors

```
F1:  .FCTR A-(F2,F3)
F2:  .FCTR B-C-(X,Y,Z)
F3:  .FCTR D-(X,Y,Z,R)
```

can be combined in this way (see also Figure 8-7):

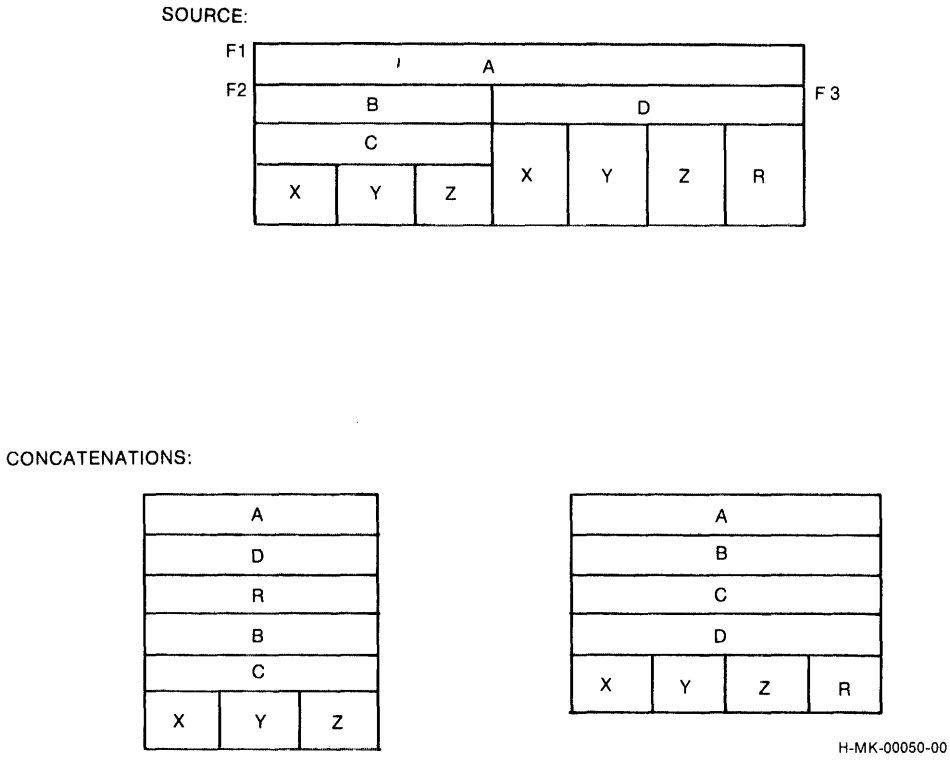
F1: .FCTR A-F3-F2
F2: .FCTR B-C-(X,Y,Z)
F3: .FCTR D-R

or in this way:

F1: A-F2-F3
F2: B-C
F3: D-(X,Y,Z,R)

This example simulates the optimization of Indexed file put and update operations discussed in *Examples of ODL Optimization* later in this section.

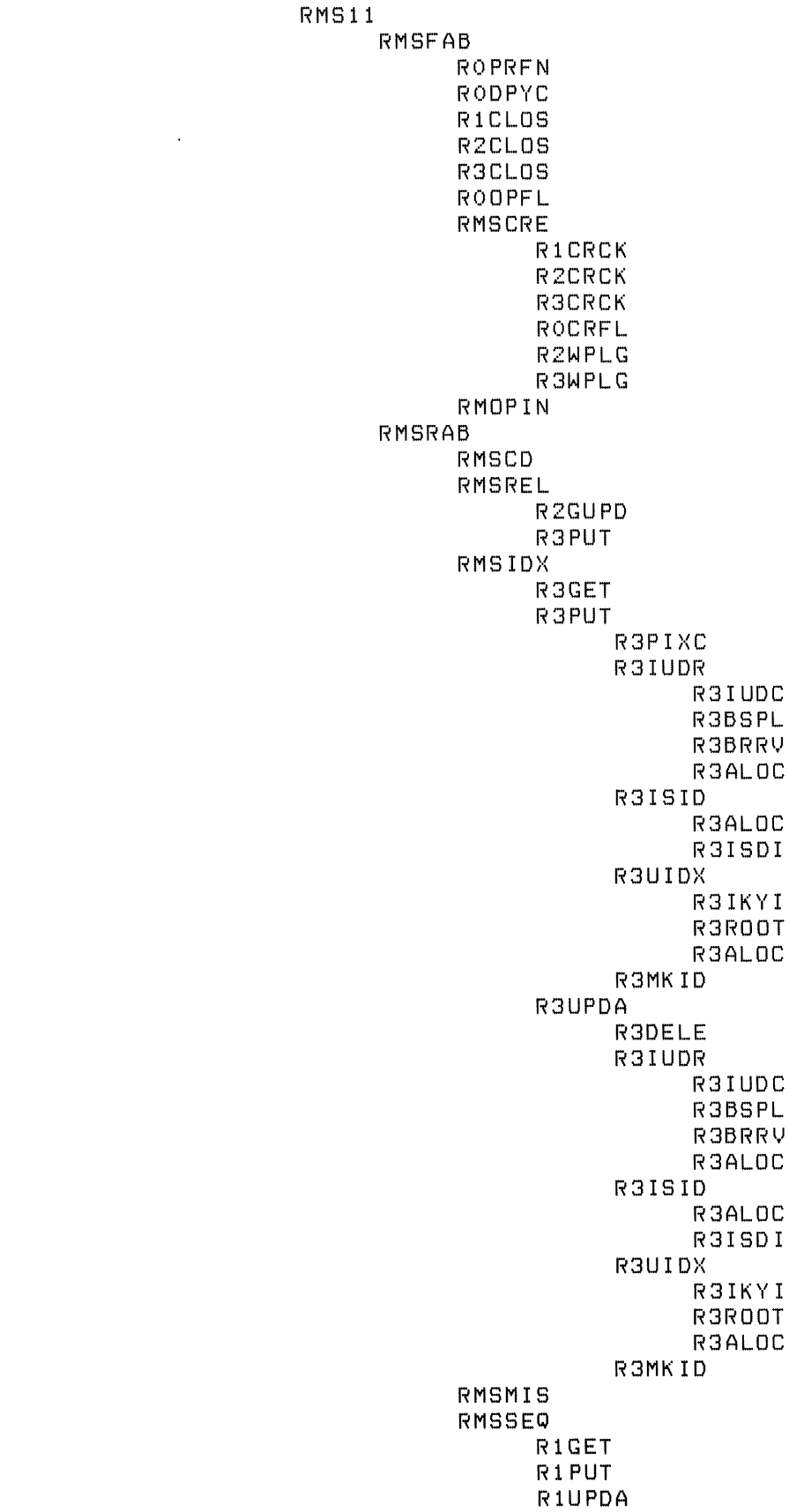
Figure 8–7: Duplicate Module Example No. 2



- Preserve the route down the overlay tree to each module, that is, all modules above the one in question must be in memory before it is brought in. See Figure 8–8 for examples.

  Example Before RMS-11 calls the R3IKYI module to insert index records into Levels 1 and higher for a put operation, it must have in memory the overlay segments RMSRAB, RMSIDX, R3PUT, and R3UIDX.

Figure 8–8: Overlay Structure Diagram



8.1.1.2.2 Possible Task Builder Errors — You can receive the following error messages from the Task Builder after you modify an ODL file:

n UNDEFINED SYMBOLS SEGMENT $*name*
Put the specified module names back in the appropriate factors.

MODULE $*name* MULTIPLY DEFINES SYMBOL $*name*
Delete the specified module name from the last factor in the overlay segment containing it.

DIGITAL established a convention for RMS-11 module names. Therefore, if the Task Builder prints the form $*name*, look in your ODL file for the module name:

R*xname*

where:

$x = 0$ for routines common to multiple file organizations
$x = 1$ for Sequential organization routines
$x = 2$ for Relative organization routines
$x = 3$ for Indexed organization routines

8.1.1.2.3 Calculating Changes in Task Size — As you change the overlay structure, you may be changing the task size:

- on disk, if you combine segments and eliminate duplicate modules
- in memory, if you affect the largest overlay segment

To determine your affect on these sizes, you must know the sizes of the RMS-11 factors and within the factors, the modules. There are two sources:

1. Your best source is a task-build *map*. A map not only shows overlay segment sizes, but their relative levels. Figure 8-8 is derived from a task-build map. You generate a map by including a second output file specification in your Task Builder command line.
2. Use the Librarian utility provided by your operating system to list the modules in LB:[1,1]RMSLIB.OLB. Add module sizes to get factor sizes.

8.1.1.2.4 Examples of ODL Optimization — Source ODL statements in this section were taken from the prototype RMS11.ODL file supplied with each RMS-11 kit.

NOTE

- Some overlay segments for Indexed file routines are larger than any of the segments containing Sequential and Relative file routines. Therefore, changes in Sequential and Relative overlay segments impact task size only if you have eliminated these large Indexed operations. However, the changes do decrease overlay I/O operations and possibly, the size of the task's disk file.

- The term LB: in this section has the same meaning on IAS/RSX-11M systems as LB:[1,1].

Sequential File Record Operations — The prototype has Sequential file record operations in four overlay segments:

- Factor RMSQOP contains the root for the record operations.
- Factor RMIN10 contains get and find operations.
- Factor RMOU1P contains the put operation.
- Factor RMOU1U contains the update operation.

To reduce the number of overlay segments, modify the RMSQOR factor, referenced by the root factor RMSQOP:

```
RMSQOR: ,FCTR (RMIN10,RMOU1P,RMOU1U)
```

as follows:

- To combine the get and find operations with the root factor, change RMSQOR to:

  ```
  RMSQOR: ,FCTR RMIN10-(RMOU1P,RMOU1U)
  ```

  GAIN 1 less overlay segment (plus an increase in segment size)

- To combine all operations with the root factor, change RMSQOR to:

  ```
  RMSQOR: ,FCTR RMIN10-RMOU1P-RMOU1U
  ```

  The RMOU1P and RMOU1U factors duplicate the module R1PSET. Therefore you must change RMOU1U from:

  ```
  RMOU1U: ,FCTR LB:RMSLIB/LB:R1UPDA:R1UBLD:R1PSET
  ```

  to:

  ```
  RMOU1U: ,FCTR LB:RMSLIB/LB:R1UPDA:R1UBLD
  ```

  GAIN 3 less overlay segments (plus an increase in segment size)

Relative File Record Operations — The prototype has Relative file record operation in three segments:

- Factors RM0X26, RMIO2C, and RMIO2A contain the root for the record operations.
- Factor RMIO2G contains the get, update, and delete operations.
- Factor RMIO2P contains the put operation.

To put all record operations in a single overlay, make the following changes:

1. Change RMIO2A from:

```
RMIO2A: ,FCTR LB:RMSLIB/LB:R2FIND:R2GSET-(RMIO2G,RMIO2P)
```

   to:

```
RMIO2A: ,FCTR LB:RMSLIB/LB:R2FIND:R2GSET-RMIO2G-RMIO2P
```

2. Change the RMIO2U factor referenced by RMIO2G from:

```
RMIO2U: ,FCTR LB:RMSLIB/LB:R2UPDA:R2PSET
```

   to:

```
RMIO2U: ,FCTR LB:RMSLIB/LB:R2UPDA
```

   because both RMIO2P and RMIO2U use the module R2PSET.

GAIN 2 less overlay segments (plus an increase in segment size)

Indexed File Record Insertion Operations — During both put and update operations, RMS-11 may insert records into buckets of all levels of the Primary and Alternate indexes. The routines that perform these record insertions are located in module sets, one set for each type of insert (Primary Level 0, Alternate Level 0, and index levels). One of the modules is a control module; the other(s) perform the insert. In the prototype overlay structure, each module of each set is a separate overlay.

The simplest improvement of Indexed file record operations combines the control and execution modules of each set into one overlay segment. Use any or all of the following steps:

- To combine the Primary Level 0 insertion pair, change:

```
RMOU34: ,FCTR LB:RMSLIB/LB:R3IUDR-(RMOU3F,RMOU3G,RMOU3A)
```

  to:

```
RMOU34: ,FCTR LB:RMSLIB/LB:R3IUDR-RMOU3F-(RMOU3G,RMOU3A)
```

  The bucket splitting routines in RMOU3G and RMOU3A stay overlaid. The usefulness of this change depends on file activity. If you populate the file using a fill number so that the file has room for random record insertions, thus avoiding bucket splits, then this optimization correlates with that savings.

  GAIN 1 less overlay segment (plus an increase in segment size)

- To combine the Alternate Level 0 insertion pair, change:

```
RMOU35: ,FCTR LB:RMSLIB/LB:R3ISID-(RMOU3H,RMOU3A)
```

to:

```
RMOU35: ,FCTR LB:RMSLIB/LB:R3ISID-RMOU3H-RMOU3A
```

There is no gain in leaving RMOU3A overlaid; the Task Builder allocates the space in the task anyway.

GAIN 2 less overlay segments (plus an increase in segment size)

- To combine the index bucket insertion pair, change:

```
RMOU36: ,FCTR LB:RMSLIB/LB:R3UIDX:R3IKEY-(RMOU3M,RMOU3A)
```

to:

```
RMOU36: ,FCTR LB:RMSLIB/LB:R3UIDX:R3IKEY-RMOU3M-RMOU3A
```

There is no gain in leaving RMOU3A overlaid; the Task Builder allocates the space in the task anyway.

GAIN 2 less overlay segments (plus an increase in segment size)

TOTAL GAIN 7 less overlay segments (plus an increase in segment size)

Indexed Get and Update Record Operations — To combine the get and update operations, change:

```
RMIO31: ,FCTR (RMIO3G,RMIO3U,RMIO3P)
```

to:

```
RMIO31: ,FCTR (RMIO3G-RMIO3U,RMIO3P)
```

This change combines the branch factors for the get and update operations and leaves them overlaying the put operation branch.

The factor RMIO3G contains no overlays, but the factor RMIO3U has several layers of overlays. You can further optimize update operations with one or both of the following steps:

- Change:

```
RMIO3U: ,FCTR LB:RMSLIB/LB:R3UPDA:R3USET:R3RPLC-(RMOU3Q,RMIO3D)
```

to:

```
RMIO3U: ,FCTR LB:RMSLIB/LB:R3UPDA:R3USET:R3RPLC-RMIO3D-RMOU3Q
```

- Make all of the following changes:

  1. Change:

     ```
     RMOU34: .FCTR LB:RMSLIB/LB:R31UDR-(RMOU3F, RMOU3G, RMOU3A)
     ```

     to:

     ```
     RMOU34: .FCTR LB:RMSLIB/LB:R31UDR-RMOU3F-RMOU3G-RMOU3A
     ```

  2. Change:

     ```
     RMOU35: .FCTR LB:RMSLIB/LB:R31SID-(RMOU3H, RMOU3A)
     ```

     to:

     ```
     RMOU35: .FCTR LB:RMSLIB/LB:R31SID-RMOU3H-RMOU3A
     ```

  3. Change:

     ```
     RMOU3Q: .FCTR (RMOU34,RMOU35,RMOU3G,RMOU3P)
     ```

     to:

     ```
     RMOU3Q: .FCTR RMOU34-RMOU35-(RMOU3G,RMOU3P)
     ```

     These changes concatenate the routines that insert records into index and data levels, while it leaves the routines that handle bucket splitting overlaid.

Combinations of File Organizations — Some applications use more than one type of file, getting from one and putting to the other (or some other combination of record operations). Optimization of ODL files for these applications requires the combination of modules from different branches of the overlay structure.

The overlay structure for record operations starts with the factor:

```
RMSREC: .FCTR RMSRAB-(RMSSQ0,RM0X2G,RM0X3G,RMSBLK,RMMISC)
```

Below this factor, the routines for a record operation are included in your task depending on whether you select a valid or a dummy factor.

You can combine Sequential file get and Indexed file put operations with the following steps:

1. Select valid RMSQOP factor for Sequential get operations (RMSQOP is referenced by RMSSQ0):

   ```
   RMSQOP: .FCTR LB:RMSLIB/LB:R1WTLS:R1CKEF:R1DELE-RMSQOR
   ```

2. Select valid RMIO3P factor for Indexed put operations (RMIO3P is referenced by RMIO31 which is included in RM0X36):

   ```
   RMIO3P: .FCTR LB:RMSLIB/LB:R3PUT:R3PSET-(RMOU3Q,RMOU33)
   ```

3. Select all factors associated with RMIO3P:

```
RMOU3Q
      RMOU34
            RMOU3F
            RMOU3G
            RMOU3A
                  RMEXTD
      RMOU35
            RMOU3H
            RMOU3A
                  RMEXTD
      RMOU36
            RMOU3M
            RMOU3A
                  RMEXTD
      RMOU3P
            RMOU3A
                  RMEXTD
RMOU33
```

4. Select dummy factors for RM0X26 (Relative file record operations), RMSBLK (Block I/O), and RMMISC (flush, free, truncate, and rewind operations):

```
RM0X26: ,FCTR RMDDSQ
RMSBLK: ,FCTR RMDBLK
RMMISC: ,FCTR RMDDMC
```

You can accomplish the same result by modifying the RMSREC factor as follows:

Change:

```
RMSREC: ,FCTR RMSRAB-(RMSSQ0,RM0X26,RM0X36,RMSBLK,RMMISC)
```

to:

```
RMSREC: ,FCTR RMSRAB-(RMSSQ0,RM0X36)
```

GAIN 7 less overlay segments (plus an increase in segment size)

The basic optimization is complete: the overlay structure reflects only those record operations used by the task.

A further improvement combines modules that remain overlaid. Each of the following steps is an optimization that you can perform without regard to the others.

- To optimize the RMSREC factor, change it from:

```
RMSREC: ,FCTR RMSRAB-(RMSSQ0,RM0X26,RM0X36,RMSBLK,RMMISC)
```

to:

```
RMSREC: ,FCTR RMSRAB-(RMSSQ0-RM0X36,RM0X26,RMSBLK,RMMISC)
```

This change combines the branch overlay segments for Sequential file record operations and for Indexed file record operations into a single overlay.

GAIN 1 less overlay segment (plus an increase in segment size)

If you changed the RMSREC factor according to the previous step, this step becomes:

Change:

```
RMSREC: .FCTR RMSRAB-(RMSSQO, RMOX3G)
```

to:

```
RMSREC: .FCTR RMSRAB-RMSSQO-RMOX3G
```

- To optimize the RMIO3P and other factors, combine the modules associated with a normal put operation, while leaving the bucket splitting modules overlaid. Change them from:

```
RMIO3P: .FCTR LB:RMSLIB/LB:R3PUT:R3PSET-(RMOU3Q,RMOU33)
RMOU3Q: .FCTR (RMOU34,RMOU35,RMOU36,RMOU3P)
RMOU34: .FCTR LB:RMSLIB/LB:R3IUDR-(RMOU3F,RMOU3G,RMOU3A)
```

to:

```
RMIO3P: .FCTR LB:RMSLIB/LB:R3PUT:R3PSET-RMOU34-(RMOU3Q,RMOU33)
RMOU3Q: .FCTR (RMOU35,RMOU36,RMOU3P,RMOU3G,RMOU3A)
RMOU34: .FCTR LB:RMSLIB/LB:R3IUDR-RMOU3F
```

GAIN 2 less overlay segments (plus an increase in segment size)

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 one of the Resident Libraries. The associated Resident Library must be resident in memory before the tasks can be executed.

While it is executing one of these tasks, the operating system uses one or 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 at the segments of the Resident Library that contain the routines for the operation.

This mapping is called *memory-resident overlaying*. Since 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 Building the RMS-11 Resident Libraries — The RMS-11 installation process provides two RMS-11 Resident Libraries:

RMSRES
: This 48KB Library consists of a root segment and five segments containing the RMS-11 routines that support all file organizations. While a task using the Library is running, one of its APRs is constantly pointing at the root segment while the other points to the appropriate routines (see Figure 8-2C).

RMSSEQ
: This 8KB Library consists of one segment of routines that support RMS-11 Sequential files only. The operating system uses one APR to map this Library.

Each RMS-11 Resident Library is position independent within the virtual address space of your task. You can assign the APRs for a Resident Library or allow the Task Builder to use default assignments: see your Task Builder manual.

IAS/RSX-11M — The RMS-11 installation process provides the following files:

RMSRES.TSK and RMSSEQ.TSK
: The task images of the Resident Libraries. This file must be resident in memory before any task using the Library can run (see "Installing an RMS-11 Resident Library," Section 8.1.2.2).

RMSRES.STB and RMSSEQ.STB
: The symbol table files for the Resident Libraries. When the Task Builder links a task to a Library, it uses the appropriate STB file to resolve global references (see "Task Building against an RMS-11 Resident Library," Section 8.1.2.2).

RSTS/E — The RMS-11 installation process provides the following files:

RMSRES.TSK and RMSSEQ.TSK
: The task images of the Resident Libraries. The appropriate file is used by the Task Builder when you build against a Resident Library (see "Task Building against an RMS-11 Resident Library," Section 8.1.2.2).

RMSRES.STB and RMSSEQ.STB
: The symbol table files for the Resident Libraries. When the Task Builder links a task to a Library, it uses the appropriate file to resolve global references (see "Task Building against an RMS-11 Resident Library," Section 8.1.2.2).

RMSRES.LIB and RMSSEQ.LIB
: Specially formatted files containing the task images of the Resident Libraries. This file must be resident in memory before any task using the Library can run (see "Installing the RMS-11 Resident Library," Section 8.1.2.3).

8.1.2.2 Task Building against an RMS-11 Resident Library — After the Resident Libraries are built, you can build tasks, directing the Task Builder to resolve global references with a Library, with one of the following sequences of commands:

TKB *command string*	or	TKB *command string*
TKB>/		TKB>/
ENTER OPTIONS:		ENTER OPTIONS:
TKB>LIBR=RMSRES:RO		TKB>LIBR=RMSSEQ:RO
TKB>//		TKB>//

where:

command string is described in the Task Builder manual for your operating system.

You must also do one of the following:

- Specify RMSRLX.ODL for RMSRES and RMSRLS.ODL for RMSSEQ as the RMS-11 secondary ODL file-name in your primary ODL file.
- Add the following term to the Task Builder command string if you are using synchronous RMS-11 operations:

  ,LB:[1,1]RMSLIB/LB:R0AUTO,RMSLIB/LB

  If you are using IAS/RSX-11M asynchronous I/O operations, you must use the following terms:

  ,LB:[1,1]RMSLIB/LB:R0EXEC:R0AUTO,RMSLIB/LB

The Task Builder looks for the appropriate .TSK and .STB files on the library device LB: in account [1,1]. If you have moved the RMS-11 Resident Libraries to another account, you must use the RESLIB option so you can specify an account number. See your Task Builder manual for an explanation of this option.

When you use RMSRES, the Task Builder adds some RMS-11 code to your program to form the task. The utility then uses the RMSRES.STB file to resolve the rest of the global references.

NOTE

- A Library does not have to be installed or resident for you to task build against it.
- The time required to build a task using a Resident Library is significantly less than the time required when you use disk-resident overlays.
- If you are not using Indexed file put and update operations, you can eliminate approximately 1K of the RMS-11 code RMSRES adds to your task with the following steps.

  1. Copy the RMS-11 standard ODL file RMSRLX.ODL (used when task building against RMSRES) into a file of a different name.

2. In the renamed version of RMSRLX.ODL, change the factor:

```
RMSALL: .FCTR *(RMXPUT,RMXUPD)
```

to

```
RMSALL:  .FCTR name
         .NAME name
```

where:

name is a six-character string of your choosing.

3. Substitute the name of the modified ODL file for RMSRLX.ODL in your primary ODL file and build the task again.

- If you use an RMS-11 secondary ODL file-name other than RMSRLX.ODL or RMSRLS.ODL, the Task Builder prints the following error message:

```
MODULE $name AMBIGUOUSLY DEFINES SYMBOL $name
```

8.1.2.3 Installing an RMS-11 Resident Library — After a Resident Library is built and before any task using the Library can run, the Library must be resident in memory. The installation process depends on the operating system; see the following paragraph that applies to your operating system.

IAS — The operating system automatically installs the Resident Library when you run a task using it. See the *IAS Operator's Procedure Manual.*

RSTS/E — Use the UTILTY ADD LIBRARY command. See the *RSTS/E System Manager's Guide* for a description of this command.

RSX-11M — Use the SET /MAIN command to declare a partition 24KW in size. Then use the INS command to load the Resident Library into that space.

See the *RSX-11M/M-PLUS MCR Operations Reference Manual* for a description of the commands.

8.1.3 Deciding between Types of Overlays

The method you use to link RMS-11 routines with your programs depends on three things:

- amount of physical memory available on your computer system
- amount of virtual address space in your task exclusive of RMS-11 routines
- number of jobs that must be run simultaneously

RMS-11 routines require memory (see Figure 8-2):

- nonoverlaid RMS-11 routines are linked into a task when they are referenced; the virtual address space required varies from 8KB to 44KB, depend-

ing on the file organizations supported and the record operations used. This RMS-11 code cannot be shared with other tasks.

- disk-resident overlays reside in the user address space. The smallest overlay structure requires about 8KB of the space; you can use larger structures. Each task using disk-resident overlays requires physical memory for its own overlay structure.
- RMSRES uses 48KB of physical memory that can be shared among all tasks using that Resident Library. A task using the Library reduces its available address space by 16KB, because two APRs are addressing the Library. The task also contains approximately 1KB of RMS-11 routines.

Physical memory is a secondary consideration when you are dealing with Resident Libraries. Primarily, a Library saves time because it eliminates the I/O operations associated with disk-resident overlays.

The Resident Library approach becomes cost-effective in terms of physical memory when you have five or more RMS-11 tasks running simultaneously:

1. Assume the 9KB overlay structure. Five tasks running in memory at the same time need a little more than 45KB of physical memory for RMS-11 alone.
2. Assume the Resident Library. The tasks require about 1KB of their own space for RMS-11, but the system must use 48KB for the Library.

The difference in memory requirements is more than compensated for by speed.

However, by reducing the memory available for jobs, a Resident Library can restrict the number of jobs that can run at the same time and thereby increase swapping or other overhead.

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 overlay 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 Overlay Structure

The overlay process causes a significant portion of the I/O performed for a program with disk-resident overlays. You should structure the task to maximize the time each segment stays in memory and minimize the number

of overlay 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 simultaneously 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 higher level languages automatically perform connect and disconnect operations during the execution of file open and close statements.

Record operations

The primary overlay 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 gets from a Sequential file, then puts to an Indexed file, you could reduce the number of overlays required for those specific operations (see Section 8.1.1.2).

Example If your task gets from an Indexed file, then updates the record, you should optimize those operations (see Section 8.1.1.2).

Where possible, though, sequence operations to minimize overlays.

8.2.2 Task Builder Considerations

The Task Builder utility constructs a task and ensures that its 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 via an ODL file.

You want to reduce the time the Task Builder runs while it builds your task. The first thing you can do is reduce the number of overlays in the task. See "Disk-Resident Overlays," Section 8.1.1. Each overlay adds time to task building because it requires a symbol table to 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 (which is the default anyway).

Example On a PDP-11/70 using the RSX-11M operating system, task builds were timed:

Map Selection	Time for Task Build
No Map	x
Short Map	$1.18x$ to $1.49x$
Long Map	$1.51x$ to $1.91x$

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 into a Logical Block Number (LBN) 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.

CONVENTION

The cover term *file directory* in this manual has the same meaning as the following system-specific terms:

System	Term
IAS	directory entry and file header(s)
RSTS/E	User File Directory entries
RSX-11M	directory entry and file header(s)

8.3.1 Retrieval Pointers on Disk

The file directory contains the retrieval pointers for a file. The representation depends on your operating system.

8.3.1.1 IAS/RSX-11M — The file 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 Logical Block Number 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.

8.3.1.2 RSTS/E — The file directory includes file retrieval blocks. Each block contains seven retrieval pointers. A pointer consists of the Logical Block Number of the first block in a file cluster. Since clusters are groups of logically

contiguous blocks and the number of blocks in a cluster (clustersize) is a file attribute, the RSTS/E file processor can calculate which logical blocks reside in a cluster.

However, if the file is contiguous, the file processor bypasses the file retrieval blocks and uses:

- the LBN of the first block in the file
- the requested VBN as an index into the file

8.3.2 Retrieval Pointers in Memory

An 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:

- Since the RSTS/E operating system stores windows in linked blocks, the system may request several I/O operations before the correct window is found.
- The IAS and RSX-11M operating systems build *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 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% of the I/O operations.

8.3.3 Optimizing Window Turning

When you reduce window turning, you improve performance. The methods for doing this are specific to the operating systems.

8.3.3.1 IAS/RSX-11M — You can reduce the I/O operations associated with window turning as follows:

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 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 *RMS-11 MACRO-11 Reference Manual.*

In each of these methods, you can specify an amount 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 allows a maximum of 127 pointers in each window.

NOTE

You can also set window size with a BASIC-PLUS-2 program, using the WINDOWSIZE keyword.

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 — Either make the file contiguous, or if that's not possible, reduce the number of extents in the file, making each extent as large as possible.

One retrieval pointer can map up to 65,536 logical blocks. When the file processor reads a pointer from the header, the software checks if 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 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, IAS and RSX-11M can map any file with a default window if the file is sufficiently contiguous.

Areas in Indexed Files — Areas localize successive block requests and reduce window turning.

Size of F11ACP — The large version of the F11ACP does not require overlays of its own routines to perform window turning, where the small version does.

8.3.3.2 RSTS/E — There are two ways to optimize the I/O operations associated with window turning:

- Directory caching involves your system configuration.
- Contiguity involves the file's attributes.

Directory Caching — RSTS/E has a *Directory Caching* feature (also called *FIP Buffering*) you can select when you generate your system. This feature uses the *Extended Buffer Pool (XBUF)* to save (*cache*) directory blocks in memory. If the system needs a certain directory block, to do a window turn,

for instance, and that block is already cached, the block is retrieved from the cache without a disk I/O operation. See the *RSTS/E System Manager's Guide* for more explanation of this feature.

NOTE

In RSTS/E Version 7.0, directory caching is related to the data caching feature (discussed in "Other Optimizations," Section 8.4).

Contiguity — Use one of the following methods to increase the number of contiguous blocks in the file:

- Make all blocks in the file contiguous and there is no window turning.

  Cost When RMS-11 detects that there is not enough room in a file to complete an operation, it automatically requests the file processor to extend the file by the Default Extension Quantity. However, if the file is contiguous, RSTS/E returns a privilege violation. RMS-11 aborts the current record operation, passing to the program the error code ER$PRV. All put and update operations that require an extension of the file will fail after that.

  At this point, the system manager or a privileged user can force the file to become noncontiguous. The file assumes all the window turning problems of noncontiguous files described previously.

Therefore, you should avoid extending (or trying to extend) contiguous files by:

- completely allocating the file when it is created
- converting the file into a larger one before its space is exhausted (use the RMSIFL or RMSCNV utility)
- Increase the file clustersize: the fewer the clusters, the fewer the retrieval pointers, and the less the operating system must turn windows to cover a virtual block.

  Cost Each allocation for the file must contain a whole number of clusters. As clustersize increases, so does the chance that enough contiguous blocks cannot be found to allocate or extend the file.

  This effort is probably worthwhile only if sequential access has high priority.
- Divide an Indexed file into areas, segregating the upper index levels into physically compact and contiguous sets of blocks.

8.4 Other Optimizations

You can improve the environment where your RMS-11 task runs with data caching and by:

- allocating more resources to the task
- improving disk usage

8.4.1 Data Caching

Data Caching, an option during RSTS/E system generation, uses the Extended Buffer Pool (XBUF) to save file blocks in memory. This facility should provide:

- Noticeable performance improvement for applications sequentially accessing Sequential and Relative files.
- Some performance improvement for applications randomly accessing indexed files. A critical factor is the relationship between bucket sizes and cache clustersizes.

See the *RSTS/E System Manager's Guide* for more details.

8.4.2 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. Of course, 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.

- IAS and RSX-11M

  Priorities

  Checkpointing

  Round-Robin Scheduling

  Swapping

- RSTS/E

  Swapping

  Priorities

8.4.3 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.
  - Assign exclusive use of the disk driver to your RMS-11 task. If you must share the driver, use one that overlaps disk seeks.
- If your system has multiple disk drives which are not heavily used by other people, spread an application's files, including disk-overlaid tasks, across the devices. Thus, while a job runs, one disk device does not access more than one file. At least, put data files on a disk device other than the one containing a disk-overlaid task image.

  Of course, 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 less the disk head moves to access it.

  However, this procedure changes file IDs.

Chapter 9
RMS–11 Utilities

TO IAS USERS

You do not have all the functions described in this chapter available via the DIGITAL Command Language interface. However, you can use these functions if you:

- install RMS–11 Version 1.8 on your computer system
- use a terminal in MCR mode
- employ the syntax shown in this manual

The RMS–11 utilities are provided as independent tasks to those who do not have full access to RMS–11 functionality, either because they are not programmers or are programming in a higher level language (other than MACRO–11).

The utilities normally installed are:

RMSBCK
: The RMS–11 file back-up utility, RMSBCK, transfers any RMS–11 disk file to another disk or to magnetic tape in a special format that cannot be read and/or modified by a user program. These RMS–11 back-up files can only be accessed by the restoring utility RMSRST. You may use RMSBCK and RMSRST to guard against hardware failure or software error by having your data backed up on another volume.

RMSCNV
: The RMS–11 file conversion utility, RMSCNV, moves records between two RMS–11 files of any organization or record format.

RMSDEF

The RMS-11 file definition utility, RMSDEF, creates RMS-11 files, defining their attributes in an interactive sequence of questions and requests for data.

NOTE

RMSDFN, the first file definition utility, is still available, but is considered to be less useful than RMSDEF. See Appendix D for a discussion of the RMSDFN utility.

RMSDSP

The RMS-11 display utility, RMSDSP, lists the RMS-11 file attributes and structural data or the names of RMS-11 back-up files on magnetic tape.

RMSIFL

The RMS-11 Indexed file load utility, RMSIFL, builds an RMS-11 Indexed file using records from another RMS-11 file of any organization. The utility uses techniques derived from the basic structure of Indexed files, rather than the standard RMS-11 file and record operations used by RMSCNV.

RMSRST

The RMS-11 file restoration utility, RMSRST, reads the back-up files created by RMSBCK and restores them with form, structure, content, and attributes identical to the original files.

Each utility is described in a section, from Purpose to Call and Termination to Cautions. However, common items, such as command lines and error messages, appear later in this section. But first a discussion on the uses of these utilities.

NOTE

- The RMS-11 utilities require ANSI-standard labels on magnetic tapes. You must mount a tape volume to inform your operating system's file processor that you are accessing an ANSI-standard tape. See Appendix F for more details on RMS-11's magnetic tape handling.

- The RMS-11 utilities do not support the use of RSTS/E special account characters ($, !, %, _, #, @) in file specifications.

9.0.1 Using The RMS-11 Utilities

This section describes some services you can perform with the RMS-11 utilities. Once you are familiar with these tasks, you will find other ways in which they can help you.

RMSBCK and RMSRST — Your data is important: it is the prime, if not the only, reason you are using a computer system; it costs money to collect, maintain, and process; it makes you money in many ways, both direct and indirect. And it is important that you never lose this data for all of these reasons.

Your data cannot be locked away all the time: like money, it must be active to be useful and growing, and in this state of activity, it is vulnerable to the chance of damage, by hardware, software, or operators. Therefore you make your data secure by making a copy of it and storing the copy away from the day-to-day environment.

The RMSBCK and RMSRST utilities are the two aspects of data security. You use RMSBCK on a daily, weekly, or monthly basis, depending on the volatility of your data. And you use RMSRST when you replace lost or corrupted data; the utility brings your data back to the same integrity it had when the file was backed up.

Example You are a wholesale distributor of auto parts. Everyday your operators process around 600 orders, involving order entry, invoice processing, and accounts receivable, plus your purchasing transactions, including inventory control, order processing, and accounts receivable.

Your data changes hourly, but you decide that a daily back up is sufficient. Therefore, each night, your second-shift system operator initiates the back-up procedure; some operating systems can do this automatically. Your data is copied to tape (or disk) by RMSBCK. Each day you are assured that if something goes wrong, you can restore the files to last night's status, using RMSRST.

RMSCNV — The RMSCNV utility has a variety of uses:

- You can back up any RMS-11 files by converting them to magnetic tape Sequential files; you restore the files by running RMSCNV or for Indexed files, RMSIFL, in the other direction.
- You can print any RMS-11 file endowed with carriage return control by converting it to a line printer or a Sequential Stream file that can be queued to a printer.

NOTE

RMS-11 does not convert numeric or binary data to ASCII representation.

- You can complete the restoration process begun by RMSRST, which cannot re-establish areas or contiguousness:
  - Create the file with the attributes you want using RMSDEF.
  - Populate the file using RMSCNV (or RMSIFL, for an Indexed file) with the single-area, noncontiguous file as input.

- You can perform daily processing using a small file designed for that purpose. Then, at the end of the day, you can use RMSCNV to add those records to a larger master file.
- You can enhance your transition to a total RMS-11 environment by converting ASCII Stream files to RMS-11 file organizations:

  1. Create the new file with RMSDEF.
  2. If you are building a Relative or Indexed file, sort the non-RMS-11 file by relative record number, if possible, or Primary Key, respectively.
  3. Build the new file with RMSCNV.

  Example You have an ASCII Stream file named BPLUS.DAT. You want to build a Relative file named NEWREL.DAT. Create the Relative file with RMSDEF, then call RMSCNV, using this command string:

  ```
  NEWREL.DAT/FO:REL=BPLUS.DAT
  ```

RMSDEF — The RMSDEF utility has a variety of uses:

- Used with RMSCNV or RMSIFL, RMSDEF creates any target file you want.
- Used with higher level language applications[1], RMSDEF extends the file creation capabilities, enabling you to use features that are not explicitly defined in the languages.

  Example Null key values.

  Example Areas.

RMSDSP — Use RMSDSP to see an RMS-11 file's attributes and structural data. Then you can:

- create another file just like it with RMSDEF.
- see the kind of file your higher level language task created so that you can match and enhance it with RMSDEF.

  Example You built an application of BASIC-PLUS-2 tasks, but you want to define file attributes not supported by the language. First, you run the task that creates your file(s), and then you display the attributes of these files with RMSDSP, preferably using a summary list file. You then know the minimum attributes your RMSDEFined file(s) must have to fit with your BASIC-PLUS-2 application. If those attributes do not match, your tasks fail with a "File Attributes Not Matched" error when they try to open the file(s).

9.0.2 Utility Conventions

Even though each of the RMS-11 utilities is a separate task, they share conventions, formats, and techniques. This section covers those common items.

[1] Especially DIBOL since a DIBOL task cannot create an RMS-11 Relative or Indexed file.

NOTE

The IAS and RSX-11M operating systems produce and support different versions of a file-name/file-type combination. These versions are totally separate files and can have different RMS-11 attributes.

Example A file named PAYROLL.DAT has five versions in account [303,351] on device DB0:

- DB0:[303,351]PAYROLL.DAT; 0 is a Sequential file with fixed-length records.
- DB0:[303,351]PAYROLL.DAT; 1 is a Relative file with variable-length records.

Example You are trying to convert one Relative file into another. You use the command line:

```
CNV REL2.DAT = REL1.DAT
```

Because you did not include the FO switch, RMSCNV creates a Sequential file named REL2.DAT with the next higher version number. The utility then runs successfully, apparently obeying your command string. The result is two versions of REL2.DAT, one a Relative file, the other a Sequential file.

9.0.2.1 Command vs. Interactive — Most of the RMS-11 utilities operate via command string interpretation while another interacts with you, performing its function in a sequence of questions and requests for data:

Command	Interactive
RMSBCK	RMSDEF
RMSCNV	
RMSDSP	
RMSRST	
RMSIFL	

Beyond their user interface, the command and interactive utilities differ in:

HELP Messages

Command utilities each have one HELP message. Interactive utilities have a unique HELP message for each question or request for data.

Error Messages

Command utilities share a set of error messages, some common to all, some unique to a particular utility. These error messages are generated in a standard format (discussed in Section 9.0.2.6). Interactive utilities, however, have separate messages for each question or request for data. All error messages are listed in Appendix E.

9.0.2.2 Installed vs. Uninstalled

CONVENTION

The cover term *installed* in this manual has the same meaning as the following system-specific terms:

System	Term
IAS	installed
RSTS/E	invocation by Concise Command Language (CCL)
RSX-11M	installed

RMS-11 utilities can be run whether or not they are installed. Only the manner of invocation changes and both calls are described in each utility's section. Your operating system user's guide explains the installation procedure.

However, one invocation capability is not described in each section because it is unique to the RSX-11M/IAS operating systems. Whether the utility is installed or not, the command string may include the switch:

/UIC=[*act nbr*]

With this switch in its command string, each RMS-11 utility runs under the account specified by [*act nbr*] until the task is terminated.

9.0.2.3 Indirect Files — An indirect file contains a sequence of commands that can be interpreted by a task, usually a system-supplied task such as a utility. These commands appear in the indirect file exactly as they are entered from your terminal. As such, they can be command strings or answers to questions.

The commands in an indirect file are executed when the file's name is provided to a utility preceded by an at sign (@). RMS-11 utilities assume an extension of .CMD if none is included in the filespec.

Example An indirect file contains a series of RMSBCK command strings. To invoke such an indirect file, you enter the command:

```
BCK @BCKCMDS.CMD or BCK @BCKCMDS
```

As an installed task, RMSBCK accesses the file BCKCMDS.CMD, executes the commands, and returns control to the operating system.

RSX-11M allows you to use indirect files directly from the MCR environment without first invoking a utility. Such an indirect file may contain command lines for more than one utility and is called by entering only the file specification preceded by an at sign (@):

```
@INDIRECT.CMD
```

For complete information on how to use indirect files, refer to your system documentation.

9.0.2.4 Command String Continuation — You should type lengthy command strings in segments, each on a separate line, all but the last ending in a hyphen (-). The hyphen causes the utility to reprompt without attempting to execute the command string.

In particular, RSX-11M automatically terminates a command line at the end of the terminal's input buffer. The utility then attempts to execute a command you may not have completed.

Example
```
>BCK DK0:*,*/QU=FIL1.DAT,DK1:[50,1FIL2.DAT,-
BCK>FIL3.*/CD:17-SEP-78
>
```

Example
```
BCK
BCK> DK0:*,*/QU=FIL1.DAT,DK1:[50,1-
BCK>]FIL2.DAT,FIL3.*/CD:17-SEP-78
BCK>^Z

Ready
```

You can continue a command line at any point within the line; however, no command line, regardless of continuations, may contain more than 158 characters, not including continuation hyphens.

9.0.2.5 Patch Level — Each RMS-11 utility contains an identifier that specifies software version number and the patch level of the utility itself. The utility prints this identifier on your terminal in response to the /ID switch, in the form:

```
VERSION 1.8nn
```

where *nn* indicates the patch level of the utility you are running.

Immediately after the RMS-11 installation process, whether it is included in the system installation or not, each utility has a patch level of 00. Thereafter, DIGITAL notifies you if a patch is necessary. Included in the notice are:

- the level of the utility before the patch is applied
- the patch procedure
- the patch level of the utility after the process is complete

9.0.2.6 Command Utility Error Messages — When RMS-11 command utilities encounter an error condition during execution, they print error messages at your terminal.

The two primary types of errors are *fatal* and *nonfatal.* Appendix E lists all error messages.

Fatal Error Messages — When an error requires the termination of processing, RMS-11 command utilities print a fatal error message, in the form:

?utl — message

where:

?	indicates a fatal error occurred
utl	is the three-character utility name
message	briefly describes the error

Example ?CNV -- ILLEGAL NUMBER OF INPUT FILES

Example ?DFN — ILLEGAL RMS RECORD FORMAT

Nonfatal Error Messages — When a nonfatal error occurs, RMS-11 command utilities print an error message and continue processing. The nonfatal error message takes the form:

utl — message

where:

utl	is the three-character utility name
message	briefly describes the error

Example The following command string is issued to RMSRST:

```
DK0:*.*=FIL1.DAT/QU,DK1:[50,1]FIL2.DAT,FIL3.*/CD:17-SEP-78
```

If the utility cannot find the file FIL1.DAT in the default account, it prints the nonfatal error message:

```
RST -- FILE NOT FOUND - SY0:[200,11]FIL1.DAT;1
```

Since additional input files were specified, RMSRST continues processing.

Crash Dump — If RMS-11 command utilities encounter a situation they cannot recover from, they produce *crash dump* information in the following form:

```
**********RMS-11 UTILITIES DAMAGE ASSESSMENT ROUTINE***
                    .
                    .
                    .
                   text
                    .
                    .
                    .
PLEASE DETACH AND SUBMIT WITH YOUR SPR
```

If you are using a CRT terminal, do not copy the information. If you are using a hard-copy terminal, do include the output with a Software Performance Report (SPR).

Submit an SPR to the proper DIGITAL office. Put in the following information:

1. The command line that led to the crash dump.
2. An attribute listing of the file(s) involved. Use RMSDSP with the FU switch.
3. A copy of the file(s) involved. If you cannot send a tape, use RMSCNV to print the file on a line printer.
4. Your operating system and version.
5. Any programming languages used to process the file(s) involved and the version you are using.
6. Other information pertinent to the situation.

9.0.3 Documentation Conventions

The following conventions adopted in this chapter make documenting and understanding the RMS-11 utilities easier.

Description of Syntax

Convention	Definition
REQUIRED WORDS	All words that are in uppercase must be in the command in the places shown.
user-defined words	When you use the command, you supply actual names for all words that are in lowercase. See the next section for specific user-defined words.
[*may be used*]	You may, but do not have to use words included within brackets. The characters ..] indicate that the series continues to include all possibilities.
Punctuation	You must use all punctuation marks included in the command.

Specific User-Defined Words — The following symbols for user-defined words have the specific meanings indicated:

Convention	Definition
nnnnn	a decimal number; the number of *n*s indicates the maximum number of digits that can be entered.
value	an argument selected from a specific and limited set of arguments; the number of letters that appear from the word *value* indicate the maximum number of characters that can be entered. For example, *v*, *val*, and *value* mean that one, three, and five characters are allowed.
dd-mmm-yy	a date; for example, 17-OCT-78

Symbols

Convention	Definition
CR	the Carriage Return character (ASCII 015_8)
(RET)	the RETURN key on your terminal; you press the RETURN key at this point in the operation you are performing.
LF	the Line Feed character (ASCII 012_8)
(LF)	the LINE FEED key on your terminal; you press the LINE FEED key at this point in the operation you are performing.
(CTRL/C)	the combination of the CTRL key and the C key. You press both keys at the same time, and ^C appears on the terminal.
(CTRL/Z)	the combination of the CTRL key and the Z key. You press both keys at the same time, and ^Z appears on the terminal.

Prints vs. Types — When the software outputs characters on a CRT or hard-copy device, the software is said to *print*.

When you must input characters on a terminal, you are asked to *type* those characters.

9.1 RMSBCK Command Utility

9.1.1 Purpose

You ensure your data from hardware failure or software error with the RMSBCK command utility. Each time you invoke this utility, specially formatted back-up copies of the specified files are placed on a back-up medium. Thereafter, if the original files are lost or damaged, you can use the RMSRST utility and these back-up copies to replace the files.

9.1.2 Effect

RMSBCK copies standard RMS-11 files from one medium to another (disk to disk or disk to tape), translating the data into a special back-up format. The back-up copy contains the source file's attributes: account number, creation and revision dates, protection code, and so on. However, they do not include placement control instructions; a file cannot be restored to the same physical place on a device that it was backed up from. See also "RMSRST," Section 9.7.

Back-up files can only be accessed properly by the RMSRST utility. User programs therefore cannot change back-up data.

RMSBCK only uses magnetic tapes with ANSI-standard labels. However, the back-up data written by the utility between the labels does not comply with ANSI standards.

RMSBCK provides four processing features:

Explicit and Implicit File Selection

The RMSBCK command string permits one or more input filespecs with or without wild card characters:

- Filespecs without wild card characters explicitly identify input files.
- Filespecs with wild card characters implicitly identify a collection of files. When using wild card characters, you can restrict the number of files selected for processing by including a date switch. A date switch causes RMSBCK to examine either the creation date or the revision date of each file selected as a result of the wild cards: the utility backs up a file only if the internal file date conforms to the date you specified.

Data Integrity Checks

RMSBCK can check data integrity extensively as each back-up file is created. You can direct the utility either to read back file contents after they are written into the back-up file or to both read back and check file contents on a byte-by-byte basis after they are written. RMSBCK automatically retries read errors if processing continues to a normal termination.

These integrity checks allow you to choose how reliable the back-up files are:

- You can just rely on software and hardware accuracy and back up your files in minimum time.
- You can verify that you can read the back-up files after they are created, adding the read time to the minimum time.
- You can guarantee the back-up files can be read and that they exactly match the source files, adding both read and compare times to the minimum.

Extended Diagnostic Messages

During file processing, either with or without data integrity checking, RMSBCK provides an extended diagnostic message capability known as Query mode. When the utility encounters an error condition described in Section 9.1.4.2, the utility prints a diagnostic message at your terminal. Immediately following the message, RMSBCK prints the query `CONTINUE (Y,N)`. Each query requires you to type a response (Y or N) indicating whether the utility is to continue or discontinue processing. With this procedure, you can ensure that RMSBCK does not terminate the processing of a collection of files when errors are encountered in the processing of any one file. See the QU switch in Section 9.1.4.2 for examples of diagnostic messages.

Optionally, you can disable the Query mode. The utility prints a diagnostic message if it encounters an error condition, but does not give you the option of continuing processing. Rather, the utility terminates automatically.

Summary Listing

You can specify that RMSBCK summarize file processing, data integrity checking, and error messages. RMSBCK lists this summary on your terminal or writes it into a file, creating a record of the utility's processing. See the SL switch in Section 9.1.4.2 for a sample of the summary.

9.1.3 Call and Termination

9.1.3.1 Installed Utility

BCK [*command string*]

If you include a command string, the utility attempts to execute it and then returns control to the current keyboard monitor or command interpreter.

If you do not specify a command string, RMSBCK assumes control of the user interface and prints the prompt:

```
BCK>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSBCK has executed a command string, it reprints the prompt.

9.1.3.2 Uninstalled Utility

RUN $RMSBCK

RMSBCK assumes control of the user interface and prints the prompt:

```
BCK>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSBCK has executed a command string, it reprints the prompt.

9.1.4 Command String

9.1.4.1 General Form

outfile[/*switch*]..]=*infile*[/*switch*]..][,*infile*[/*switch*]..]...]

where:

outfile is the filespec of a back-up file to be created by the RMSBCK utility:

- If the back-up medium is a disk, both the file-name and extension must be wild card characters (*.*): the file-names and extensions of the back-up files will be the same as the associated input files. The version can be omitted or can be a wild card. In both instances, RMSBCK uses the version of the associated input file.

If you do not use wild card characters, RMSBCK prints the following error message and terminates execution of the command string:

```
?BCK -- NO RENAME ALLOWED
```

- If the back-up medium is magnetic tape, the back-up files produced by a command string are contained within a single file on the tape. This file is known as a *container file* since it contains back-up files.

  You must name the output container file in the command string, but the name need not correspond to the name of any input file. The version can be omitted: a value of 0 is used.

  If you do not name the container file, RMSBCK prints the following error message and terminates execution of the command string:

```
?BCK -- EXPLICIT CONTAINER FILE NAME NECESSARY
```

NOTE

You must use the container file-name in the infile specification for RMSRST.

RMSBCK does not prevent multiple container files with the same name on the same volume. Therefore, you should specify unique file-names, extensions, and version numbers for output container files written on the same volume.

infile is the filespec of a disk file of which a back-up copy is to be created. You can use wild card characters to specify implicitly a collection of files.

If you specify a nondisk device in the infile specification, RMSBCK prints the following error message and terminates execution of the command string:

```
?BCK -- ILLEGAL DEVICE - dvn:
```

where: *dvn* is the device name and number you used.

switch may be one code shown in Table 9–1 and described in Sections 9.1.4.2 through 9.1.4.4.

NOTE

A command string may also consist of the word "HELP" or a question mark (?). RMSBCK responds with a HELP message.

Table 9-1: RMSBCK Utility Switches

Type	Switch	Description	Default Process
String	? or HELP	Print HELP message.	No help.
Global	/ID	Identity current version of the RMSBCK utility.	No id.
	/QU	Enable Query mode.	QU
	/SL[:*filespec*]	Provide summary listing (in file, if specified).	No summary.
Outfile	/RA*	Read after writing.	NORA
	/RC*	Check after writing.	NORC
	/RW	Rewind magnetic tape before writing.	NORW
	/SU	Supersede old files.	NOSU
Infile	/CD:*dd-mon-yy*[:*v*]	Back-up files based on creation date.	No date checking.
	/RD:*dd-mon-yy*[:*v*]	Back-up files based on revision date.	No date checking.

* The RA and RC switches are not available on RSTS/E. That operating system does not allow the RMSBCK task to rewind a magnetic tape device to beginning of file after the utility accesses that file for writing.

9.1.4.2 Global Switches

- ID causes RMSBCK to print its current version number, in the form:

  ```
  BCK -- VERSION 1.8nn
  ```

  where:

  nn is the patch level of the utility (see "Patch Level," Section 9.0.2).

  This switch may appear alone as a command string.

- QU enables the Query mode. When the Query mode is enabled, the RMSBCK utility allows you to continue or terminate processing when one of the following occurs:

  - A read error on an input file.
  - A read-after-write error on an output file when the RA switch is specified.
  - A check-after-write error when the RC switch is specified.
  - The table allocated internally for data integrity checks or automatic retry of read errors is full.

  When one of these errors occurs, the utility prints a diagnostic message specifying the type of error and the name of the file being processed. If you answer Y (yes), RMSBCK continues processing the current file and com-

mand string. If you answer N (no), the utility terminates processing immediately, bypassing the rest of the command string.

Example BCK -- CHECK AFTER WRITE ERROR ON OUTPUT FILE - *filespec*
VBN *vbn1* TO *vbn2*, CONTINUE (Y,N)?

Example BCK -- INTEGRITY CHECK TABLE FULL, CONTINUE (Y,N)?

You can disable the Query mode through the NOQU switch. The RMSBCK utility prints a diagnostic message when it encounters an error condition, but does not give you the option of continuing processing.

Default If you specify no version of the QU switch, Query mode is enabled.

- SL[:*filespec*] causes RMSBCK to summarize activity during the execution of the command string. This summary contains:
  - the command string
  - the names of files successfully backed up
  - the names of files not backed up, with associated error messages
  - diagnostic messages produced in Query mode
  - a summary of any input errors and of errors remaining in output back-up files following automatic retry of data integrity checks (when you specify RA or RC).

  If you do not specify an argument (*filespec*) with the SL switch, RMSBCK prints the summary listing on your terminal. However, if you supply a filespec, the utility creates the specified file and writes the summary listing as the contents of the file.

  Example

```
RMSBCK - VERSION 1.800 27-JUN-1979 16:55:14
DB1:*.*/SL:BCKUP.LOG=DB0:*.MAC

BCK -- FILE PROCESSING COMPLETE - DB0:[303,351]RMSPRE.MAC; 74
BCK -- FILE PROCESSING COMPLETE - DB0:[303,351]UTLMLB.MAC; 146
BCK -- FILE ALREADY EXISTS - DB1:[303,351]DSRMS.MAC; 1
BCK -- FILE ALREADY EXISTS - DB1:[303,351]VEXT.MAC; 1
BCK -- FILE PROCESSING COMPLETE - DB0:[303,351]BLKIO.MAC; 12
BCK -- FILE ACCESS ERROR ON DB0:[303,351]B.MAC; 2,ERROR CODE=177734
BCK -- FILE ALREADY EXISTS - DB1:[303,351]WR.; 10
BCK -- FILE PROCESSING COMPLETE - DB0:[303,351]SWTABL.MAC; 55
```

Default If you specify no version of the SL switch, RMSBCK prints messages on your terminal only if it encounters a fatal or nonfatal error condition.

9.1.4.3 Outfile Switches

- RA directs RMSBCK to read the back-up file after writing it. The utility reads back each block of the back-up file: if the device hardware detects a read error, RMSBCK's response depends on whether Query mode is enabled or not. See QU under "Global Switches," Section 9.1.4.2.

NOTE

You cannot use the RA and RC switches in the same command string. RMSBCK prints the following error message and terminates execution of the command string.

```
?BCK -- CONFLICTING OPTION - /sw
```

where:

sw is RA or RC depending on which is second in the command string.

Default If you do not include the RA switch, RMSBCK does no read-after-write data integrity checking.

- RC directs RMSBCK to check the back-up file after writing it. The utility reads each virtual or tape block of the back-up file back into memory. At the same time, it reads into memory the corresponding virtual block(s) of the source file. RMSBCK then compares the contents of the two buffers. If an error occurs during this process (a hardware read error or a mismatch between the contents of the two buffers), RMSBCK's response depends on whether the Query mode is enabled. See QU under "Global Switches," Section 9.1.4.2.

  See note under RA in this section.

  Default If you do not include the RC switch, RMSBCK does not check the back-up file after writing it.

- RW directs RMSBCK to rewind a magnetic tape before writing back-up container files. Rewinding logically deletes files existing on the tape.

  If you specify RW when the back-up medium is a disk, RMSBCK prints the following error message and terminates execution of the command string:

  ```
  ?BCK -- ILLOGICAL USE OF OPTION - /RW
  ```

  Default If you do not include the RW switch, RMSBCK adds new container files to the logical end of a tape.

- SU causes RMSBCK to supersede files in the output account with the same file-name, extension, and version as a new back-up file.

  This switch applies only if the back-up medium is a disk. If SU is specified for magnetic tape, RMSBCK prints the following error message and terminates execution of the command string:

  ```
  ?BCK -- ILLOGICAL USE OF OPTION - /SU
  ```

  Default If you do not include the SU switch, RMSBCK does not supersede files in the output account. If the utility encounters a file with the same file-name, extension, and version as an input file, it prints the following nonfatal error message and continues processing.

  ```
  BCK -- FILE ALREADY EXISTS - filespec
  ```

9.1.4.4 Infile Switches

- CD:*dd-mon-yy*[:*v*] causes RMSBCK to back up files based on their creation dates. You can combine this switch with wild card characters to identify a group of files, depending on the value of the switch argument *v*:

Value	Selection
	Those files that satisfy the wild card specification and that were:
None	created on the specified date
A	created after the specified date
B	created before the specified date

RMSBCK treats a file with no or a null creation date as though that file were created before January, 1900.

Default If you do not include the CD switch, RMSBCK applies no date criterion when it selects files for back up (unless you specified the RD switch).

- RD:*dd-mon-yy*[:*v*] causes RMSBCK to back up files based on their revision dates, that is, the dates they were last accessed. You can combine this switch with wild card characters to identify a group of files, depending on the value of the switch argument *v*:

Value	Selection
	Those files that satisfy the wild card specification and that were:
None	revised on the specified date
A	revised after the specified date
B	revised before the specified date

RMSBCK treats a file with no or a null revision date as though that file were revised before January, 1900.

Default If you do not include the RD switch, RMSBCK applies no date criterion when it selects files for back up (unless you have specified the CD switch).

9.1.4.5 Command String Examples

- MT0:ALPHA.BKP;1/RC=ALPHA.DAT

RMSBCK writes a back-up copy of the file ALPHA.DAT on the ANSI-labeled magnetic tape mounted on device MT0: The container file created on the tape is named ALPHA.BKP;1. The utility checks each block of the output back-up file after it is written and since Query mode is enabled by default, reports a qualifying error condition via a diagnostic message and a query. You must indicate with a Y or N whether or not processing should continue.

- MT1:SAFE.BKP=*.*/QU/SL

RMSBCK backs up on ANSI-labeled magnetic tape the highest version of all files in the default account on SY: into a container file named

SAFE.BKP; 0. Query mode is explicitly enabled; therefore, a qualifying error condition causes RMSBCK to output a diagnostic message and wait for your response whether to continue or terminate. While RMSBCK processes the input files, it prints a summary listing on your terminal, specifying the names of all files.

- MT0:MASTER.BKP=*.DAT/CD:31-DEC-77:B/SL:BACKUP.LST

  RMSBCK backs up onto ANSI-labeled magnetic tape the highest versions of all files in the default account on SY: that were created before December 31, 1977 and have an extension of DAT. Query mode is enabled by default. The utility also creates a summary listing file named BACKUP.LST in the default account on SY:. The file contains a listing of the names of the files processed along with copies of diagnostic messages and queries output during processing.

- DK1:[100,10]*.*; */SU=DK1:[100,20]*.*; */RD:01-JUN-78:A/NOQU

  RMSBCK backs up all files under account number [100,20] on DK1: that were accessed after June 1, 1978. The back-up copies are written into account [100,10] on the same disk (DK1:), superseding files with the same filenames, extensions, and versions. Query mode is explicitly disabled; therefore, a qualifying error causes RMSBCK to print a diagnostic message and terminate processing.

9.1.5 Cautions

- RMSBCK cannot back up a system disk as a bootable volume.
- The back-up medium must be on-line and mounted before you issue a command string.
- When the back-up medium is a disk, you cannot rename output files. That is, the name of each back-up file will be the same as the name of the input file.
- When the back-up medium is magnetic tape, you can use only ANSI-labeled tapes.
- When the back-up medium is magnetic tape, RMSBCK uses 2048-character blocks. You cannot change this. If you want to use larger blocks, use RMSCNV with the BL switch to convert the input file to a Sequential tape file.

  Example You have an 86,000-block Indexed file (400,000 82-byte records). The back-up copy of this file requires at least two magnetic tape volumes. However, if you convert the file to tape with 4096-character blocks, you need only one 2400-foot tape volume. You can restore the file quickly with RMSIFL.

- If you include in a command string one name of a file listed in several accounts via the PIP /ENTER switch, RMSBCK copies the entire file even

though it exists in another account. But when you restore the file with RMSRST, the link established by PIP is broken.

Example You have a file of customer names and addresses stored in account [30,1], but it has different names in different accounts:

[10,1]CLIENTS.DAT
[20,1]CUSTOMERS.DAT
[30,1]ACCTSRECV.DAT

These filespecs point to the one file in account [30,1].

If you back up [20,1]CUSTOMERS.DAT, RMSBCK copies the file out of [30,1]. However, when you restore the file with RMSRST, it is written into account [20,1], and the restored file is no longer linked with the other names, that is, [10,1]CLIENTS.DAT and [30,1]ACCTSRCV.DAT.

- RMSBCK handles unused blocks in a file according to the file organization:
  - If the file is Sequential or Relative, RMSBCK does not copy the unused blocks.
  - If the file is Indexed, RMSBCK copies the unused blocks.
- Do not use either the infile or outfile specification as an argument in the SL switch.

9.2 RMSCNV Command Utility

9.2.1 Purpose

The RMSCNV command utility provides a versatile mechanism for performing any of the following functions:

- creating a new Sequential file from the records of an existing Sequential, Relative, or Indexed file.
- superseding an existing Sequential file with the records of another Sequential file or the records of a Relative or Indexed file.
- appending to a Sequential file the records of another Sequential file or the records of a Relative or Indexed file.
- writing into an existing Relative or Indexed file the records of another file of any organization.

9.2.2 Effect

All the functions performed by RMSCNV involve the movement of records from one file to another file. The utility reads records from the specified input file and writes them into the specified output file. The manner in which records are read and written depends on the file organizations of both the input and output files and on switches you specify in the command string.

Output File Organization	Processing
SEQUENTIAL	RMSCNV: • creates the output file with the attributes of the input file if the append (AP) switch is not specified. The utility then reads records from the input file and writes them sequentially into the new output file. If the output file already exists: – RMSCNV creates the next higher version of the file. – RMSCNV terminates with the following error message: ?CNV -- FILE ALREADY EXISTS • supersedes an existing file if you specify the SU switch. The utility reads records from the input file and writes them sequentially over the records already in the output file, starting with the first record position in the file. RMSCNV creates the file with the attributes of the input file if it does not exist. • appends records onto an existing file if the AP switch is used. The utility reads records from the input file and writes them sequentially into the output file, starting with the record position following the last record already in the file. RMSCNV terminates with the following error message if the output file does not exist: ?CNV -- FILE NOT FOUND - *filespec*
RELATIVE	RMSCNV reads records from the input file and writes them into successive record cells of the output file, beginning with cell 1. If the utility encounters a cell containing a record, it terminates with a FATAL RMS ERROR message (STS = 175220_8). All records written to that point are still in the file. You should examine the input and output files to determine the extent of processing. RMSCNV terminates with the following error message if the output file does not exist: ?CNV -- FILE NOT FOUND - *filespec*
INDEXED	RMSCNV reads each record from the input file, then applies the output file's record structure, that is, key placement within the record, to the data; this structure is an attribute of the output file and is not dependent on the input file organization. The record is then inserted into the output file on the basis of the value found in the Primary Key field. Finally, RMSCNV updates the output file's Primary and any Alternate indexes to reflect the presence of the record. RMSCNV terminates with the following error message if the output file does not exist: ?CNV -- FILE NOT FOUND - *filespec*

Input File Organization	Processing
SEQUENTIAL	RMSCNV reads the records in the Sequential Access Mode, starting with the first record in the file.
RELATIVE	RMSCNV reads the records in the Sequential Access Mode, starting with record cell 1.
INDEXED	RMSCNV reads the records in the Sequential Access Mode, following the key of reference specified in the command string (the Primary Key is default).

9.2.3 Call and Termination

9.2.3.1 Permanently Installed Utility

CNV [*command string*]

If you include a command string, the utility attempts to execute it and then returns control to the current keyboard monitor or command interpreter.

If you do not specify a command string, RMSCNV assumes control of the user interface and prints the prompt:

```
CNV>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSCNV has executed a command string, it reprints the prompt.

9.2.3.2 Uninstalled Utility

RUN $RMSCNV

RMSCNV assumes control of the user interface and prints the prompt:

```
CNV>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSCNV has executed a command string, it reprints the prompt.

9.2.4 Command String

9.2.4.1 General Form

outfile[/*switch*]..]=*infile*[/*switch*]

where:

outfile is the filespec of the output file that is to receive the records of the input file. Wild card characters are not permitted in any field of the specification. The default version for Sequential files with the AP or SU switch, Relative, and Indexed files is the highest version. When creating a Sequential file, RMSCNV uses the highest version number plus 1.

infile is the filespec of the input file that is the source of records to be written to the output file. Wild card characters cannot appear in any field of this filespec.

switch may be a code shown in Table 9–2 and described in Sections 9.2.4.2 through 9.2.4.4.

NOTE

A command string may also consist of the word "HELP" or a question mark (?). RMSCNV responds with a HELP message.

Table 9-2: RMSCNV Utility Switches

Type	Switch	Description	Default Process
String	? or HELP	Print HELP message.	No help.
Global	/ID	Identify current version of the RMSCNV utility.	No id.
	/SL[:*filespec*]	Provide summary listing (in file, if specified).	No summary.
Outfile	/AP	Append records to Sequential file.	No append.
	/BL[:*nnnn*]	Set magnetic tape block size.	512 bytes.
	/FO:*val*	File organization.	Sequential file.
	/LO	Follow fill numbers when writing Indexed file.	Fill buckets.
	/MA	Use Mass Insert and sequential put operations to optimize performance.	No Mass Insert; random put operations.
	/PD[:[#]*x*]	Pad input records to output record length	Abort if different lengths.
	/SU	Supersede existing Sequential file.	No supersede.
	/TR	Truncate input records to output record length	Abort if different lengths.
	/WF	Write or read fixed control area.	Ignore fixed control area.
Infile	/KR:*n*	Key of reference number.	Primary key (*n*=0).

9.2.4.2 Global Switches

- ID causes RMSCNV to print its current version number, in the form:

  ```
  CNV -- VERSION -- 1.8nn
  ```

  where:

  nn is the patch level of the utility (see "Patch Level," Section 9.0.2).

  This switch may appear alone as a command line.

- SL [:*filespec*] directs RMSCNV to summarize processing. If a file specification is not included in the switch, RMSCNV prints the summary on your terminal. If you specify a file, the utility creates it and writes the summary listing into it.

  The summary includes:

  - the command string
  - copies of error messages produced during execution of the utility.

- an indication that input records could not be written into an output Indexed file because duplicate keys (for one or more key fields) were not permitted:

  If the summary appears on your terminal, the indicator is the message:

  ```
  SOME DUPLICATE RECORDS NOT WRITTEN
  ```

  If the summary is written into a file, RMSCNV supplies the indicator DUP RCD=, followed by the first 72 characters of the record that could not be written. Each record left out of the output file is shown in the summary listing file.

Default If you do not include the SL switch, RMSCNV prints messages on your terminal only when it encounters a fatal or nonfatal error.

9.2.4.3 Outfile Switches

- AP directs RMSCNV to append records to an existing Sequential file. RMSCNV adds records read from the input file to the end of the output file. AP and SU cannot be specified in the same command string.

  NOTE

  If the output file is not sequentially organized, RMSCNV ignores the AP switch.

  Default If you do not include the AP switch, RMSCNV's action depends on the presence of the SU switch:

  - If you specified SU, the utility obeys the switch.
  - If you did not specify SU, RMSCNV:
    - creates the next higher version of the file
    - terminates with the following error message:

      ```
      ?CNV -- FILE ALREADY EXISTS
      ```

- BL [:*nnnn*] directs RMSCNV to control the physical block size of the output file when it is being created on magnetic tape. Any size specified by *nnnn* must be from 18 through 8192 characters and should be a multiple of four (otherwise, RMS-11 rounds it to the next higher multiple of four).

  Default If you do not include the BL switch, RMSCNV uses a tape block size of 512 characters.

- FO:*val* specifies the organization of the output file as one of the following:

  SEQ — Sequential file
  REL — Relative file
  IDX — Indexed file

  Default If you do not include the FO switch, RMSCNV assumes Sequential organization. The utility responds as described in "Effect," Section 9.2.2.

- LO directs RMSCNV to write records into the buckets of an Indexed file according to the fill numbers established when the file was created. See "Data Allocation" in RMSDEF, Section 9.3.4.6 and "Fill Number" in Section 6.7.2.1.

  Default If you do not include the LO switch, RMSCNV inserts as many records into each bucket as possible.

- MA directs RMSCNV to activate the RMS–11 Mass Insert I/O technique (see Section 6.7.2.2) and then use sequential put operations to insert records into the outfile.

  If you use the MA switch, the infile:

  - must be sorted into ascending order according to the Primary Key field of the output file. Otherwise, RMSCNV prints the following error message and terminates processing.

    ```
    ?CNV -- INPUT RECORDS NOT IN ASCENDING ORDER
    ```

  - should contain records with Primary Key values greater than the Primary Key value of any record already in the file. The Mass Insert technique requires that records be inserted at logical end-of-file. If the file is empty, there are no Primary Key values to exceed.

    If the input records do not meet this requirement, RMSCNV continues to process them. However, the utility does not achieve the improved performance that is otherwise expected from Mass Insert.

  Example You have two Indexed files, IDX.IN and IDX.OUT. The Primary Key of IDX.IN is defined as bytes 0 through 15 of each record. The Primary Key of IDX.OUT is defined as bytes 0 through 8 of each record. Since RMS–11 orders records in Indexed files according to the Primary Key value, you can convert IDX.IN into IDX.OUT using the MA switch with the default key of reference.

  Note that if the Primary Key of IDX.OUT coincides with an Alternate key of IDX.IN file, you can still use the MA switch as long as you specify the proper Alternate Key in the KR switch (see Section 9.2.4.4).

  Default If you do not include the MA switch, RMSCNV uses random put operations to insert each record into the output file.

- PD [:[#]*x*] directs RMSCNV to pad records read from the input file to the output file's record length before writing them to the output file. Padding character is specified as follows:

Switch	**Character**
PD	NULL (byte value 000_8)
PD:*x*	*x* is ASCII A–Z, 0–9, or special character except #, ?, and @
PD:*x*	*x* is octal number 000–377 (43 for #, 77 for ?, and 100 for @)

  You use the PD switch only when the output file specifies fixed-length records.

  Default If you do not include the PD switch, and the input records are shorter than the output file's records may be, RMSCNV terminates with the following message:

  ```
  ?CNV -- INPUT AND OUTPUT RECORD SIZES DO NOT CORRESPOND
  ```

- SU directs RMSCNV to supersede an existing Sequential file. RMSCNV utility supersedes a file in the output account with the same file-name, extension, and version number. AP and SU cannot be specified in the same command line.

  Default If you do not include the SU switch, RMSCNV's action depends on the presence of the AP switch:

  - If you specified AP, the utility obeys the switch.
  - If you did not specify AP, RMSCNV:
    - creates the next higher version of the file
    - terminates with the following error message:

```
?CNV -- FILE ALREADY EXISTS
```

- TR directs RMSCNV to truncate records read from the input file to the output file's record length before writing them to the output file. The trailing bytes of the records are truncated.

  Default If you do not include the TR switch, and the input records are longer than the input file's record may be, RMSCNV terminates with the following error message:

```
?CNV -- INPUT AND OUTPUT RECORD SIZES DO NOT CORRESPOND
```

- WF directs RMSCNV to handle variable-with-fixed-control (VFC) format records in either the input or output file. The utility processes the possible combinations as follows:

Input	Output	Processing
VFC	VFC	If the fixed control areas are the same size, RMSCNV performs a straight-forward copy process; if the areas are not identical in length, the utility terminates without converting the input file, printing the message:

```
?CNV -- INPUT AND OUTPUT FIXED CONTROL HEADER SIZES DO NOT CORRESPOND
```

VFC	FIX VAR STM	The fixed control area of each input record is written as the first *n* bytes of each output record, where *n* equals the size of the fixed control area. The variable portion of the input record completes the output record.
FIX VAR STM	VFC	The first *n* bytes of each input record are written into the fixed control area of each output record, where *n* is the length of the fixed control area. The remaining data in the input record is written into the variable portion of the output record.

If you include the WF switch and neither file specifies VFC records, RMSCNV terminates with the following error message:

```
?CNV -- ILLEGAL USE OF /WF WITH RECORD FORMAT
```

Default If you do not include the WF switch and one of the files contains VFC records, the fixed control area of each record is ignored:

Input	Output	Processing
VFC	any	Only the variable portion of each record is read from the input file and written to the output file.
any	VFC	Data is read from the input file, but is written only into the variable portion of each output record.

9.2.4.4 Infile Switches

- KR: *n* indicates the key that establishes the order in which records are sequentially read from the Indexed input file and written to the output file, where *n*=0 for the Primary Key, *n*=1 for the First Alternate Key, and so on until *n*=9 for the Ninth Alternate Key.

  RMSCNV will not process any input Indexed file with a key of reference greater than nine. It terminates with the error message:

```
?CNV -- INVALID /KR VALUE
```

  Default If you do not include the KR switch, RMSCNV uses the Primary Key as the key of reference.

9.2.4.5 Command String Examples

- PAYROL.DAT/FO:IDX=NAME.FIL

  RMSCNV reads each record of the input file NAME.FIL and examines the contents in the Primary Key field (defined as an attribute of the output file PAYROL.DAT). Then the utility inserts the record into PAYROL.DAT.

- NAME.DAT/FO:REL=MASTER.DAT/KR:1

  RMSCNV reads the Indexed input file MASTER.DAT, using the First Alternate Key of the file to establish the sequence of access. Then the utility writes the records sequentially into an empty Relative output file NAME.DAT, starting with record cell one.

- ALPHA.BAR/FO:SEQ/AP=BETA.BAR/KR:2

  RMSCNV reads the Indexed input file BETA.BAR, using the Second Alternate Key of the file to establish the sequence of access. Then the utility appends the records onto the end of the existing Sequential output file ALPHA.BAR.

  If the AP switch had not been specified, RMSCNV would have created the next higher version of ALPHA.BAR and written the records from BETA.BAR into it.

- DELTA.DAT=GAMMA.DAT

  RMSCNV creates the Sequential file DELTA.DAT and copies the records from GAMMA.DAT into it.

- NEWPAY.DAT/WF/FO:REL=OLDPAY.DAT

  RMSCNV reads fixed-length records from the Indexed input file OLDPAY.DAT, using the Primary Key of the file to establish the sequence of access. Then the utility writes the records sequentially in a VFC format into the existing Relative output file NEWPAY.DAT. Record format is an attribute of each file. As each record is written, the first bytes become the fixed control area.

- MT3:INVENT.BCK/TR/BL:1024/AP=SY:INVENT.DAT/KR:0

  RMSCNV reads the Indexed input file INVENT.DAT, using the file's Primary Key to establish the sequence of access. Then the utility writes the records to magnetic tape, truncating them to the output file's record length before adding them to the end of the existing file, and formatting the tape blocks to 1024 bytes each.

9.2.5 Cautions

- If an existing Sequential file is used for output and neither the supersede (SU) or the append (AP) switch is specified, the utility creates the next higher version of the file and uses it.
- A Relative or Indexed output file must exist before RMSCNV is invoked. The file may be created through an application program or the RMSDEF utility.
- If you did not allow duplicate keys for one or more key fields of an Indexed output file, RMSCNV bypasses any input record that would cause such duplication to occur. The utility notifies you of its action as described under "SL[:*filespec*]" in Section 9.2.4.2.
- If any input record is shorter than the Primary Key of an Indexed output file, RMSCNV terminates with the error message:

  ```
  ?CNV -- ILLEGAL RMS RECORD SIZE
  ```

  Any records processed before the too-short record may or may not have been written to the output file, depending on a variety of circumstances,including buffer sizes and I/O techniques. You must examine the input file to determine its contents; a way to do this is:

  CNV TI:=*filespec*/KR:0

  Note that you cannot fix the too-short record and then run RMSCNV with the same command string: the output Indexed file may contain some or all of the input records read before the exception record. You have to delete and recreate the output file.

- When both the input and output files have variable-length records, RMSCNV requires that the Maximum Record Size defined for the input file

be less than or equal to the Maximum Record Size defined for the output file. The utility terminates with the following error message if the MRS of the input file is greater than that of the output file:

```
?CNV -- ILLEGAL RMS RECORD SIZE
```

Example An RMS-11 file with variable-length records, Maximum Record Size of 30 bytes, is named VAR30.DAT. Another similar file, named VAR50.DAT, has a Maximum Record Size of 50 bytes.

The following command string is valid:

```
VAR50.DAT=VAR30.DAT
```

The following command string is not valid:

```
VAR30.DAT=VAR50.DAT
```

- When both the input and output files have fixed-length records, RMSCNV requires either the TR or PD switch if the fixed record lengths are not equal. If neither switch is specified, the utility terminates with the error message:

```
?CNV -- INPUT AND OUTPUT RECORD SIZES DO NOT CORRESPOND
```

- When the input file contains variable-length records and the output file contains fixed-length records, RMSCNV requires both the TR and the PD switches. If both are not specified, the utility terminates with the error message:

```
?CNV -- SWITCH /TR OR /PD OR BOTH ARE NEEDED FOR THIS CONVERT
```

- You can use unit record devices for input and output files, restricted of course by their capabilities:

  Input from terminal or card reader
  Output to terminal or printer

  You use a terminal for either output or input file with the filespec "TI:". You end terminal input with the CTRL/Z.

- Do not use either the infile or outfile specification as an argument in the SL switch.

9.2.6 I/O Techniques

RMSCNV uses the following I/O techniques to maximize its performance:

- If either the output or input file is a Sequential file, RMSCNV sets the Multi-Block Count to five blocks (see "I/O Techniques," Section 3.3.3.5).
- When the output file is an Indexed file:
  - RMSCNV attempts to allocate extra buffers to cache the Root buckets of all keys specified for the file. If the utility doesn't have enough task space to cache all Roots, it continues processing with the two buffers required for an Indexed file.
  - RMSCNV uses Deferred Write (see Section 7.3.2).

9.3 RMSDEF Interactive Utility

9.3.1 Purpose

The interactive RMSDEF utility creates RMS-11 files, allowing you to control all attributes of the files being created.

9.3.2 Effect

You specify file attributes by responding to requests for data and questions from RMSDEF. The method of questioning is outlined in Figure 9-1. The figure shows the general flow of processing while Section 9.3.6 shows actual messages and legal responses as well as the meaning of each attribute. You can also get help from the utility by typing a question mark (?) in response to any question or request for data.

RMSDEF can also build an indirect command file while you operate it. This command file can be used thereafter to (re)create file(s) and can be modified to create other similar files. See Sections 9.3.3.1 and 9.3.4.1.

NOTE

- Command files generated by RMSDEF V1.5 are not compatible with RMSDEF V1.8.
- Command files generated by RMSDEF are not compatible across operating systems.

RMSDEF, however, does not write records into the file. The actual data contents of the file must be loaded after the RMSDEF utility creates the file. You can use either an application program, the RMSCNV utility, or the RMSIFL utility to write records into the file.

NOTE

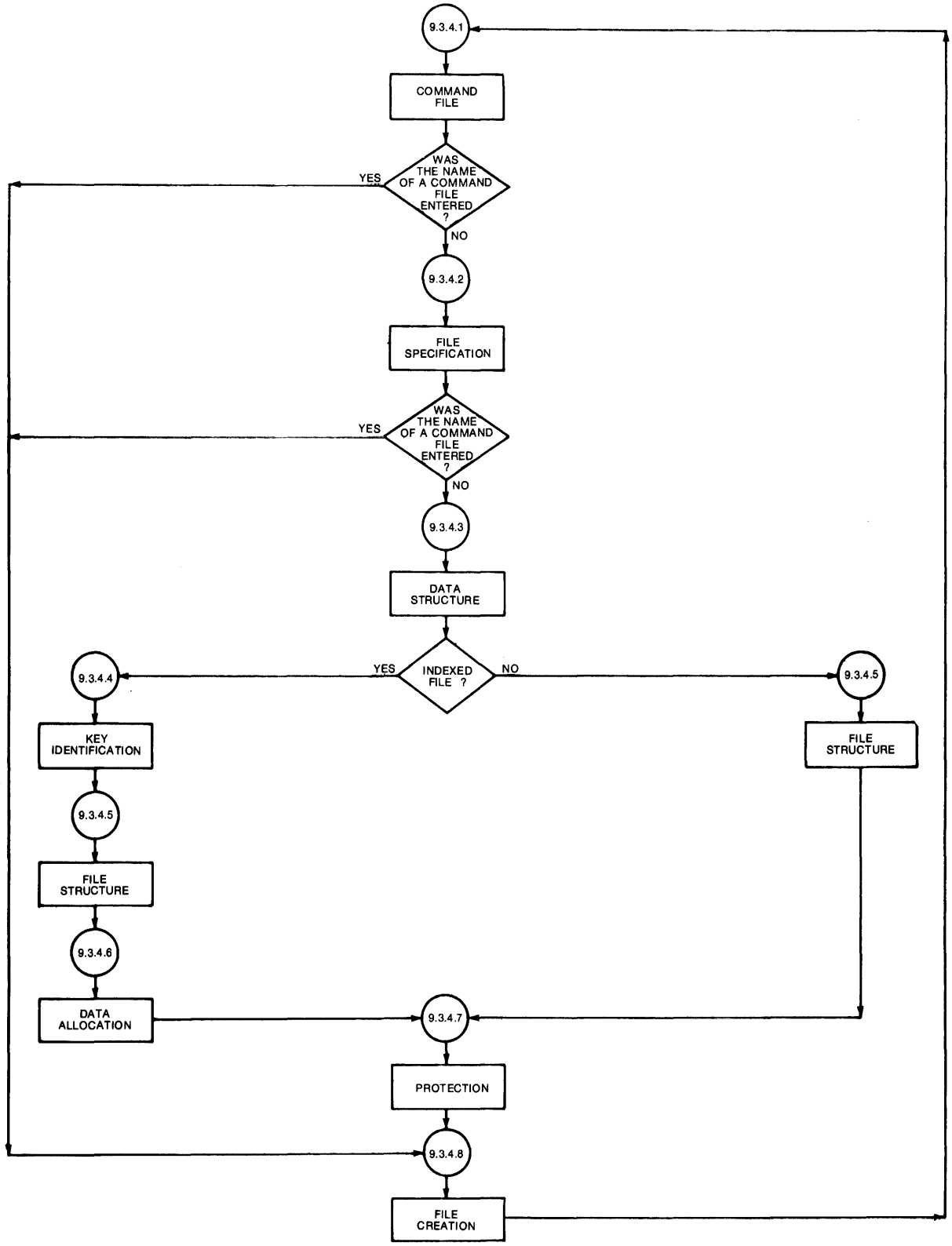
RMSDEF interprets all quantities you enter as **decimal** numbers.

The following list indicates the information that RMSDEF always requests as well as the requests that depend on specifications typed.

1. Command File?
   a. If yes, file specification
   b. If yes, create only command file or create both RMS-11 and command file?

2. File Specification
3. Data Structure
    a. Minimum
        1. File organization
        2. Record format
        3. Maximum record size
        4. CARRIAGE RETURN control?
    b. Possible
        1. Size of fixed control area for VFC records
        2. Maximum number of records in a Relative file
        3. Block-spanning records in a Sequential file?
        4. FORTRAN character control if no CARRIAGE RETURN control?
4. Key Definition (Indexed files only)
    a. Minimum
        1. Position of key
        2. Size of key
        3. Data type
        4. Name of key
        5. Duplicate keys?
    b. Possible
        1. Change keys if duplicatable? if Alternate Key
        2. Null key value? if Alternate Key
        3. Null key character if null key value
5. File Structure
    a. Minimum
        1. Areas? (Indexed files only)
        2. Placement control?
        3. Initial allocation quantity
        4. Default extension quantity
        5. Contiguous?

Figure 9-1: RMSDEF Processing Flowchart



b. Possible

1. Location if placement control
2. Exactly if placement control?
3. Type of alignment if placement control and areas

6. Data Allocation (Indexed files only)

a. Minimum

1. Number of bytes in data buckets filled
2. Number of bytes in index buckets filled

b. Possible

1. Area containing index Level 0 for each key if areas
2. Area containing index Levels 2+ for each key if areas
3. Area containing index Level 1 if areas

7. Protection

9.3.3 Call and Termination

9.3.3.1 Permanently Installed Utility

DEF [command string]

where *command string* is one of the following:

@filespec — if you want RMSDEF to read a command file named *filespec* that was previously generated by RMSDEF (see Section 9.3.4.1). The utility processes the file, printing messages only when it comes to the file creation phase (see Section 9.3.4.8).

HELP or ? — if you want an introductory HELP message for the RMSDEF utility.

/ID — if you want the version number of the RMSDEF utility installed on your system. RMSDEF prints the following message:

```
THIS IS THE RMS DEFINE UTILITY, VERSION 1.8nn
```

where *nn* is the patch level of the utility (see "Patch Level," Section 9.0.2).

If you do not include a command string, the utility prints:

```
DO YOU WANT TO GENERATE A COMMAND FILE FOR FUTURE USE(NO)?
```

See Section 9.3.4 for the complete dialog sequence.

9.3.3.2 Uninstalled Utility

RUN $RMSDEF

The utility prints:

```
DO YOU WANT TO GENERATE A COMMAND FILE FOR FUTURE USE(NO)?
```

See Section 9.3.4 for the complete dialog.

9.3.3.3 Terminating The Utility — You may terminate RMSDEF at any time by typing a CTRL/Z. The current keyboard monitor or command interpreter resumes control of your interface.

9.3.4 Process

9.3.4.1 Command File

1. The terminal prints:

   ```
   DO YOU WANT TO GENERATE A COMMAND FILE FOR FUTURE USE(NO)?
   ```

   Type one of the following:

Y	if you want to enter a filespec for an indirect command file and RMSDEF to write entries into the file as you move through the utility. Go to step 2.
N or RET	if you do not want to build a command file. Go to Section 9.3.4.2.
@filespec	if you have already built a command file with RMSDEF and want RMSDEF to read it now and create the specified file. Go to Section 9.3.4.8.
/ID	if you want the utility to identify itself with a version number. RMSDEF prints the following message:

   ```
   THIS IS THE RMS DEFINE UTILITY, VERSION 1.8nn
   ```

   where:

   nn is the patch level of the utility (see "Patch Level," Section 9.0.2).

2. The terminal prints:

   ```
   ENTER FILE SPECIFICATION FOR COMMAND FILE:
   ```

   Type a filespec. See Appendix A for RMS-11 restrictions on file specifications defined by your operating system documentation.

3. The terminal prints:

   ```
   DO YOU WANT TO CREATE THE FILE YOU WILL BE DESCRIBING(NO)?
   ```

Type one of the following:

Y	if you want RMSDEF to create the RMS-11 file being specified as well as the command file.
N or RET	if you do not want to create the RMS-11 file, only the command file that can be used to create the file later.

NOTE

The command file created by RMSDEF takes the following form shown. The utility follows each comment with the appropriate sequence of your entries, each on a separate line. Where you enter only RET to accept the default value, RMSDEF places the CR/LF character sequence on a separate line in the command file.

```
;THE FIRST QUESTION ASKS FOR THE FILE SPECIFICATION.
;THE NEXT QUESTIONS DEAL WITH FILE ORGANIZATION & RECORD ATTRIBUTES
;THE FOLLOWING QUESTIONS DEAL WITH KEYS (for Indexed files only)
;THE NEXT QUESTIONS DEAL WITH ALLOCATION AND PLACEMENT ATTRIBUTES
;THE NEXT QUESTIONS ASK ABOUT FILL SIZES FOR KEYS (for Indexed files only)
;THE FOLLOWING QUESTIONS DEAL WITH FILE PROTECTION
```

9.3.4.2 File Specification —

1. The terminal prints:

   ```
   ENTER FILE SPECIFICATION:
   ```

   Type one of the following:

filespec	if you want to create (or simulate the creation of; see Section 9.3.4.1, step 3) an RMS-11 file. Type a filespec as defined by your operating system documentation; see also Appendix A for RMS-11 restrictions on filespecs. Go to Section 9.3.4.3.
@filespec	if you have built a command file (see "Command File," Section 9.3.4.1) and want RMSDEF to use it and create the file specified. Go to Section 9.3.4.8.

2. The terminal prints:

   ```
   IF THE FILE ALREADY EXISTS, DO YOU WANT TO SUPERSEDE IT(NO)?
   ```

   It is possible that a file already exists with the file specification you entered in step 1. You must tell RMSDEF what to do if such a conflict

occurs when it tries to create the file you describe during this interactive session. On IAS/RSX-11M, a conflict results only if you specified a version number in step 1.

Type one of the following:

Y	if you want RMSDEF to supersede an existing file.
N or RET	if you do not want RMSDEF to create the file you will describe by superseding a file that exists.

CAUTION

1. If you say "N" here, and
2. If you are generating a command file, and
3. If you describe other files after this one in that command file, and
4. If, when you use the command file, a file with the filespec entered in step 1 exists:

RMSDEF prints the following error message and terminates processing of the command file.

```
A FILE WITH THE FILE SPECIFICATION YOU ENTERED ALREADY EXISTS
```

9.3.4.3 Data Structure

1. The terminal prints:

```
ENTER FILE ORGANIZATION(SEQ):
```

Type one of the following:

SEQ or RET	for Sequential organization
REL	for Relative organization
IDX	for Indexed organization

NOTE

If you indicated a magnetic tape device in the filespec, RMSDEF does not request a file organization. Since a magnetic tape file requires Sequential organization, the utility prints:

```
SINCE YOU SPECIFIED A NON-DISK DEVICE,
YOUR FILE ORGANIZATION MUST BE SEQUENTIAL!
```

2. The terminal prints:

```
ENTER RECORD FORMAT(VAR):
```

Type one of the following:

VAR or RET	if the records in the file have differing, or variable, lengths.
FIX	if the records in the file have the same, or a fixed, length.
VFC	if each record in the file has a control area with a fixed length and a data area of no standard length, that is, Variable with Fixed Control.
STM	if the records in the file have no specific format, but are delimited only by record terminator characters. The stream format is permitted for Sequential disk files only.
UDF	if there are no records (or you do not want RMS-11 to recognize records) in the file; this format is used only for Block I/O files, such as RMS-11 back-up files. The undefined format is permitted for Sequential disk or tape files only.

RMSDEF rejects a format that conflicts with the file organization specified, for instance, STM for Indexed files.

3. If you specified VFC in step 2, the terminal prints the following message; otherwise, go to step 4.

```
ENTER SIZE OF FIXED CONTROL AREA(2):
```

Type the decimal number of bytes in the fixed control area of each record in the file. The minimum size is one byte; the maximum size is 255 bytes; the default is two bytes.

4. The terminal prints:

```
ENTER MAXIMUM RECORD SIZE:
```

or

```
ENTER MAXIMUM RECORD SIZE(0):
```

Type a decimal number indicating the maximum number of bytes in any record in the file. RMS-11 checks this value whenever a record access operation is requested for this file: if the record specified exceeds the maximum size, RMS-11 returns an error. A size of zero disables the RMS-11 check, but a nonzero value is required for all Relative files and all files with fixed-length records.

5. If you specified REL in step 1, the terminal prints the following message; otherwise, go to step 6.

```
ENTER MAXIMUM NUMBER OF RECORDS(0):
```

Type a decimal number indicating the maximum number of records that this Relative file will contain. RMS-11 checks this value whenever a record access operation is requested for this file: if the relative record number specified exceeds the maximum record number, RMS-11 returns an error. A RET sets the number to zero, which disables the RMS-11 check. The zero allows the file to contain as many records as is physically possible (the technical maximum is 2.14748×10^9).

6. If you specified SEQ in step 1, the terminal prints the following message; otherwise, go to step 7.

```
WILL YOU ALLOW RECORDS TO CROSS BLOCK BOUNDARIES(YES)?
```

Type one of the following:

Y or RET	if you want records to cross block boundaries.
N	if you do not want records to span blocks. If you specified FIX in step 2 and a maximum record size greater than 512 in step 4, the terminal prints:

```
SINCE YOU SPECIFIED FIXED SIZE RECORDS, YOU MUST HAVE A MAXIMUM RECORD
SIZE LESS THAN 512 (THE SIZE OF 1 BLOCK) OR YOU MUST ALLOW RECORDS TO
CROSS BLOCK BOUNDARIES. PLEASE CHANGE ONE OF THESE.
```

RMSDEF repeats steps 4 and 6. Change either your Maximum Record Size or the answer to crossing block boundaries.

7. The terminal prints:

```
DO YOU WANT CARRIAGE RETURN CONTROL(YES)?
```

Type one of the following:

Y or RET	if you want each record to be preceded by a line feed character and followed by a carriage return character when it is written to a unit record device (printer, terminal, and so on). See the following note and go to appropriate section.
N	if you do not want CARRIAGE RETURN control and/or you do want FORTRAN character control. Go to step 8.

8. The terminal prints:

```
DO YOU WANT FORTRAN CHARACTER CONTROL(NO)?
```

Type one of the following:

Y	if you want the first byte of each record to be interpreted as a FORTRAN forms control character when the record is written to a unit record device.
N or RET	if you do not want FORTRAN character control.

NOTE

If you indicated a magnetic tape device in the file specification, at this point RMSDEF requests:

```
ENTER MAG TAPE BLOCK SIZE (512):
```

Type a decimal number between 18 and 8192 representing the number of bytes in each tape block. The number should be a multiple of four; if it is not, RMS-11 rounds the number up to the next multiple of four before writing it as an attribute. A (RET) sets the size to the default of 512 bytes.

The utility then bypasses other processing and immediately requests protection information (see Section 9.3.4.7).

9.3.4.4 Key Definition — As indicated by Figure 9–1, this section applies only to Indexed files. RMSDEF begins this portion of dialog with the message:

```
IT'S TIME TO DEFINE THE PRIMARY KEY
```

1. The terminal prints:

   ```
   ENTER DATA TYPE(STR):
   ```

   Type one of the following:

STR or (RET)	if your key value is a string of alphanumeric characters.
INT	if your key value is a two- or four-byte signed integer.
BIN	if your key value is a two- or four-byte unsigned binary number.
PAC	if your key value is a packed decimal number.

   See "Key Data Type," Section 6.2.3, for a discussion of key data types.

2. The terminal prints:

   ```
   ENTER POSITION OF KEY:
   ```

   Type a decimal number indicating the position of the first byte of the key within each record. For instance, if the key starts with the first byte of the record, its position is 0. The second byte has position 1 and so on.

   A position number must be specified for each segment of a segmented string key; the numbers are separated by commas and enclosed in parentheses. Integer, binary, and packed decimal keys cannot be segmented.

   Example A key has three segments that start at the first, fourth, and sixteenth positions. They are described as follows:

   ```
   (0,3,15)
   ```

3. The terminal prints:

```
ENTER SIZE OF KEY:
```

Type the decimal number of bytes in the key, that is, its length. Valid lengths depend on the data type entered in step 1:

Data Type	Length Restrictions
STR	Minimum = 1 byte Maximum = 255 bytes
INT	2 or 4 bytes
BIN	2 or 4 bytes
PAC	Minimum = 1 byte Maximum = 16 bytes

A length must be specified for each segment of a segmented string key; the numbers are separated by commas and enclosed in parentheses. A length must be typed for each position number specified in step 1, but the sum of all lengths cannot exceed 255. Integer, binary, and packed decimal keys cannot be segmented.

Example The lengths of the keys in the step 2 example are entered as follows:

```
(2,2,16)
```

4. The terminal prints:

```
ENTER NAME OF KEY(NONE):
```

Type one of the following:

RET	if you do not want to specify a key name.
name	if you want to name the key being defined; up to 32 ASCII characters are allowed.

5. The terminal prints:

```
WILL YOU ALLOW DUPLICATE KEYS(dflt)?
```

Type one of the following:

Y	if the file may contain more than one record with the same value for this key. Alternate Keys must be specified as duplicatable before they can be specified as changeable (in step 6).
N	if each record in the file must have a unique value for this key. RMS-11 returns an error if duplication is attempted; that is, put or update operations fail when the record has a value in this key field exactly like a record already in the file.

Defaults and the values of *dflt* are:

Primary Key No

Alternate Keys Yes

When defining the Primary Key, go to step 10. Otherwise, go to step 6.

6. If you specified Yes in step 5, the terminal prints the following message; otherwise, go to step 7.

```
WILL YOU ALLOW KEYS TO CHANGE(YES)?
```

Type one of the following:

Y or (RET)	if this Alternate Key can change values during an update operation; that is, the record may be read with one value for the key and rewritten with another value for the same key.
N	if this Alternate Key must not change after the record is originally created.

7. The terminal prints:

```
DO YOU WISH TO DEFINE A NULL KEY VALUE(NO)?
```

Type one of the following:

Y	if you want the file to contain some records that cannot be accessed via this key. See "Key Characteristics," Section 6.2.5. Go to step 8 only if you specified a string key (step 1). Otherwise, go to step 9; the other key data types have a null key value of 0.
N or (RET)	if you do not want to specify a null value for this key. Go to step 9.

8. The terminal prints:

```
ENTER NULL KEY VALUE CHARACTER:
```

Type one of the following:

c	the single character itself, if it is not #, ?, @, or SPACE.
#40	for SPACE.
#43	for the reserved character #.
#77	for the reserved character ?.
#100	for the reserved character @.
#nnn	any octal byte value ($000–377_8$), specified by *nnn*.

9. The terminal prints:

```
JUST FINISHED ALTERNATE KEY nnn
```

where:

nnn starts with 1 and is incremented when you answer Yes to the next question (step 10).

10. The terminal prints:

```
DO YOU WANT TO DEFINE MORE KEYS(NO)?
```

Type one of the following:

Y — if you want to define more keys for the file. You may define up to 254 alternate keys; however:

- The RMSCNV and RMSIFL utilities do not read higher than the ninth Alternate Key, that is, the KR switch must be less than or equal to nine (see Sections 9.2.4.4 and 9.5.4.4).
- Your application language may not support that many keys. See your language manual.

RMSDEF then requests the information in (steps 1–9) for each Alternate Key indicated; the Alternate Keys are defined in order, beginning with the First Alternate after the Primary Key is defined.

N or RET — if all keys for this file are defined.

9.3.4.5 File Structure

1. If you specified IDX for file organization, the terminal prints the following message; otherwise, go to step 2.

```
DO YOU WANT TO DEFINE AREAS(NO)?
```

Type one of the following:

Y — if you want parts of this file to be logically different, with different attributes. See "Areas," Section 6.3. The following questions (steps 2–10) are asked for each area.

N or RET — if you want this indexed file located in one area. RMSDEF requests the following information (steps 2–10) for the file once.

2. The terminal prints:

```
DO YOU WANT PLACEMENT CONTROL(NO)?
```

Type one of the following:

Y	if you want to specify an exact location on disk for this file or area. Go to step 3.
N or RET	if you do not want to specifically locate this file or area. Go to step 7.

3. On VAX systems only, the terminal prints:

```
IF YOU WANT TO SPECIFY VOLUME, ENTER ITS NUMBER WITHIN THE VOLUME
SET(0):
```

Type one of the following:

0 or RET	If you want the file or area placed: • on the volume indicated by the filespec you entered (see Section 9.3.4.2, step 1) — if that disk is **not** the root volume of a defined volume set. • on any volume in the set if the disk indicated by the filespec is the root volume of a defined volume set. A zero here essentially nullifies a YES answer in step 6.
n	if you want the file or area placed on another disk in the volume set indicated by the filespec you entered, where *n* is 1 through the maximum number set for the volume.

4. On VAX systems, the terminal always prints the following message. On IAS and RSX-11M, the terminal prints the following message only for Areas 1+ in Indexed files.

```
ENTER TYPE OF ALIGNMENT(LBN):
```

Type one of the following:

LBN or RET	if the location you will specify in step 5 is a Logical Block Number on the disk volume.
VBN	valid only for Areas 1+ in Indexed files, if the location you will specify in step 5 is a Virtual Block Number already established within the file; that is, you are trying to closely align this area with a defined area.

CYL — valid only on VAX systems, if the location you will specify in step 5 is the number of a cylinder on the disk volume. By specifying CYL, you are directing the file processor to place the first block of this file or area at or near the first logical block of the cylinder you will specify in step 5.

5. The terminal prints:

```
ENTER LOCATION:
```

Type the decimal number location of the first block for this file area. There is no default. For the first area defined, this number is:

- a Logical Block Number
- Device Cluster Number

For VAX users, if you specified CYL in step 4, you can enter here either:

- a valid number for a cylinder
- -1, if you want the area or file to start with the first block of any cylinder on the disk

6. The terminal prints:

```
EXACTLY(NO)?
```

Type one of the following:

Y — if this file or area must start in the exact LBN location specified in step 5. If this location is not available, RMSDEF prints an error message after it tries to create the file. Exact VBN locations are already taken, by definition.

N or RET — if you will accept the closest approximation of the LBN or VBN location specified in step 4, when the exact location is not available.

7. The terminal prints:

```
ENTER INITIAL ALLOCATION IN BLOCKS(0):
```

Type a decimal number indicating the initial size of the file or area in blocks. A RET sets the value to zero: the area will be created, but it must be extended before any records can be written into it. Since automatic file extension is time-consuming, the file should be fully allocated when it is created (see Section 6.6).

NOTE

If you have specified placement control (step 2), you should type a nonzero allocation quantity. An inital allocation of zero blocks essentially nullifies placement control: the file processor does not allocate any blocks to the file, so it does not use the placement control information.

Example You can place multiple files at the same LBN or DCN location, even answering YES to EXACTLY? (step 6), as long as you use zero allocation quantities for all but one.

8. If you specified REL or IDX for file organization, the terminal prints the following message; otherwise, go to step 9.

   ```
   BUCKET SIZE(1):
   ```

   Type a decimal number indicating the number of blocks in a bucket for this file or area. The minimum is the number of blocks that can contain one record (according to the size specified; see Section 9.3.4.3, step 4); the maximum is 32 or 15 blocks; the default is one. This number determines the number of blocks read into memory during each file access operation and therefore affects processing speed and the amount of memory a program accessing this file requires. See "Bucket Size" Section 6.5.

9. The terminal prints:

   ```
   ENTER DEFAULT EXTENSION QUANTITY IN BLOCKS(0)
   ```

   Type a decimal number indicating the number of blocks that should be added to the file or area each time RMS-11 extends it. The Default Extension Quantity (DEQ) should be a multiple of the bucket size. RMS-11 requests this number of blocks from the operating system; RSTS/E, however, actually extends the file to the next full cluster larger than the request.

   A RET sets the value to zero: RMS-11 adds only the minimum space required each time it expands the file or area. A definite, but reasonable extension quantity speeds processing.

10. The terminal prints one of the following messages, except on RSTS/E, for Areas 1+:

    ```
    DO YOU WANT A CONTIGUOUS AREA(NO)?
    ```

    or

    ```
    DO YOU WANT A CONTIGUOUS FILE(NO)?
    ```

    Type one of the following:

    Y — if you want the disk space for this file or area allocated in contiguous blocks. If the file processor cannot find that many contiguous blocks, it will not create the file, even if enough non-contiguous blocks are available.

    A contiguous file or area may be extended although the disk space added may not be contiguous with the original allocation.

    RSTS/E cannot extend a contiguous file. Be sure your initial allocation quantity (step 7) is sufficient.

11. If you answered Yes in step 1 (you have an Indexed file), the terminal prints the following messages; otherwise, go to the next appropriate section.

```
JUST FINISHED WITH AREA NUMBER nnn
DO YOU WANT TO DEFINE MORE AREAS(NO)?
```

Type one of the following:

Y	if you want to specify attributes for another area of your file. Areas are numbered sequentially, starting with zero. The areas are associated with the index and data portions of the file in the next section of the utility. Go to the step in this section indicated below:

Step	Operating System
2	IAS or RSX-11M
7	RSTS/E

N or RET	if you have defined enough areas for this file. Go to step 12.

12. If you defined one or more areas with a DEQ of zero, the terminal prints the following message; otherwise, go to the next section.

```
ENTER FILE DEFAULT EXTENSION QUANTITY(0):
```

Type a decimal number indicating the number of blocks that should be added to the file each time RMS-11 extends it. The DEQ should be a multiple of the bucket size. A RET sets the value to zero: RMS-11 will add only the minimum space required each time it expands the file. A definite, but reasonable extension quantity speeds processing.

9.3.4.6 Data Allocation — As indicated by Figure 9-1, this section applies only to Indexed files. RMSDEF begins this portion of dialog with the message:

```
IT IS TIME FOR AREA NUMBERS AND FILL FACTORS FOR KEYS.
```

The questions are asked for each key defined (see Section 9.3.4.4). RMSDEF begins the first session with the message:

```
THESE QUESTIONS ARE FOR THE PRIMARY KEY
```

Any further sessions begin with the message:

```
THESE QUESTIONS ARE FOR ALTERNATE KEY NUMBER nnn
```

where:

nnn starts with 1.

1. The terminal prints:

```
ENTER AREA NUMBER FOR DATA BUCKETS FOR THIS KEY(0):
```

Type an integer (0–n) indicating the area already defined (see Section 9.3.4.4) that should contain the data portion (Level 0) of this key's index.

4. The terminal prints:

```
WORLD(R ALLOWED):
```

Type one of the following:

(RET)	if you want this file available for Read access only by all accounts outside the owner, group, and privileged accounts.
NONE	if you do not want other accounts to have access to this file after it is created.
R, W, E, and/or D	if you want to specify a level of protection other than none and Read only. You may specify one or more of the letters representing Read, Write, Edit, and Delete without separation.

1. The terminal prints:

```
CLUSTERSIZE(0):
```

Type a decimal number of blocks in a cluster for this file. The number is a system-oriented extension quantity and must be equal to or a power of two greater than the device clustersize (maximum of 256). A (RET) sets the clustersize to zero, which means RMS-11 defaults to the volume clustersize, and -1 sets the clustersize to 256.

2. The terminal prints:

```
PROTECTION(60):
```

Type a decimal protection number for the file. See Part I of the *RSTS/E System User's Guide* for an explanation of and a guide to calculating protection numbers.

9.3.4.8 File Creation — The RMSDEF utility attempts to create the file.

Success — If RMS-11 does not return an error, the utility prints:

```
YOUR FILE HAS BEEN CREATED!! -- filespec
```

If you chose to create a command file (see Section 9.3.4.1), RMSDEF also prints the following message. Otherwise, the utility requests you to enter another file specification (Section 9.3.4.2).

```
YOUR FILE HAS BEEN PROCESSED AND A COMMAND FILE GENERATED!! -- filespec
DO YOU WANT TO CLOSE THE INDIRECT FILE(NO)?
```

Type one of the following:

Y	if you are finished specifying files for the command file. The utility returns to the question about command file generation (see Section 9.3.4.1).

N or RET — if you want to specify another RMS-11 file and you want the command file to include your input. RMSDEF continues to use the command file originally specified and to obey your answer to the "DO YOU WANT TO CREATE THE FILE YOU WILL BE DESCRIBING?" question (see Section 9.3.4.1, step 3). The utility returns to the request for a file specification (see Section 9.3.4.2).

You may start the process to create another file or type CTRL/Z to terminate the utility.

Error — RMSDEF allows you to recover from one type of creation error. The terminal prints an error description, followed by:

```
DO YOU WISH TO REENTER THE ENTIRE FILE SPECIFICATION (YES)?
```

Type one of the following:

Y or RET — if you know how to correct the error and/or want to enter another filespec. The utility requests the filespec and attempts to create the file using it.

N — if you do not know how to correct the error and/or want to start again. RMSDEF returns to the file specification request (Section 9.3.4.2).

Other errors result in one of the following messages:

- THIS FILE CANNOT BE CREATED
  RMS STATUS CODE = *nnnnnn* SYSTEM CODE = *nnnnnn*
- THIS FILE CAN NOT BE CREATED SINCE THE FILE ALREADY EXISTS AND YOU DID NOT SPECIFY SUPERSEDE.

"Error Code Mapping," Appendix B.2, defines the codes represented by *nnnnnn*.

The utility returns to the file specification request to let you restart the definition process (see Section 9.3.4.2).

9.4 RMSDSP Command Utility

9.4.1 Purpose

The RMSDSP utility provides a concise description of the RMS-11 file attributes of any file as well as the contents of back-up container files on magnetic tape.

9.4.2 Effect

The RMSDSP utility lists the following information for each file specified in a proper command string. The information is either printed on the calling terminal or written into a disk file, depending on the format of the command string.

LEGEND

filespec	a complete file specification
dd-mon-year hh:mm	the day, month, year, and time the file was created or revised

NOTE

RMSDSP does not print null dates.

rvn	the operating system's revision number of the file
code	one of the following: • one or more of the protection codes R (read), W (write), E (extend), and D (delete) • an integer specifying the protection status of the file, in the form <*nnn*>
format	FIXED, VARIABLE, VFC, STREAM, or ??? (undefined)
length	an integer number of bytes set as the maximum record length
character control	CARRIAGE RETURN or FORTRAN
bk	number of blocks
filsiz	number of blocks currently allocated to the file
unusd	number of blocks currently allocated to the file that are **not** in use
type	the key data type as STR, INT, BIN, or PAC
changes	NO CHANGES or CHANGEABLE

9.4.2.1 For Disk Sequential Files —

```
filespec                        FILE ORGANIZATION:       SEQUENTIAL
CREATED: dd-mon-year hh:mm      [REVISED:  dd-mon-year hh:mm (rvn)]
FILE PROTECTION:                    code
RECORD FORMAT:                      format=length
RECORD ATTRIBUTES:                  [character control]      [NO SPANNING]
FILE ATTRIBUTES:
        ALLOCATION=filsiz  EXTEND QUANTITY=nnnn
                           [CONTIGUOUS]
```

9.4.2.2 For Magnetic Tape Files —

```
filespec                        FILE ORGANIZATION:      SEQUENTIAL

FILE PROTECTION:                        code
RECORD FORMAT:                          format=length
RECORD ATTRIBUTES:                      [character control]
[NO SPANNING]
FILE ATTRIBUTES:
        BLOCK SIZE=nnnn
```

9.4.2.3 For Relative Files —

```
filespec                        FILE ORGANIZATION:      RELATIVE
PROLOGUE VERSION NUMBER: n
CREATED: dd-mon-year hh:mm      [REVISED:   dd-mon-year hh:mm (rvn)]
FILE PROTECTION:            code
RECORD FORMAT:              format=length
                            MAXIMUM RECORD NUMBER=nnnn
RECORD ATTRIBUTES:          [character control]
FILE ATTRIBUTES:
        ALLOCATION=filsiz   EXTEND QUANTITY=nnnn      BUCKET SIZE=bk
                            [CONTIGUOUS]
```

9.4.2.4 For Indexed Files —

```
filespec                        FILE ORGANIZATION:      INDEXED
PROLOGUE VERSION NUMBER: n
CREATED: dd-mon-year hh:mm  [REVISED: dd-mon-year hh:mm (rvn)]
FILE PROTECTION:            code
RECORD FORMAT:              format=length
RECORD ATTRIBUTES:          [character control]
FILE ATTRIBUTES:
        ALLOCATION=filsiz   EXTEND QUANTITY=nnnn          BUCKET SIZE=bk
                            [CONTIGUOUS]
NUMBER OF KEYS:             nnnn
```

9.4.2.5 For Indexed Files with FU Switch Specified — See the RMSDEF interactive utility in section 9.3 for an explanation of key attributes.

```
filespec                        FILE ORGANIZATION:      INDEXED
PROLOGUE VERSION NUMBER: n
CREATED: dd-mon-year hh:mm  [REVISED: dd-mon-year hh:mm (rvn)]
FILE PROTECTION:            code
RECORD FORMAT:              format=length
RECORD ATTRIBUTES:          [character control]
FILE ATTRIBUTES:
        ALLOCATION=filsiz   EXTEND QUANTITY=nnnn          BUCKET SIZE=bk
                            [CONTIGUOUS]

AREA NUMBER:  nnn
        ALLOCATION REMAINING = unused
        BUCKET SIZE = nn EXTEND QUANTITY = nnnn
           .
           .
           .
```

```
PRIMARY KEY:     type
        [NO] DUPLICATES changes          [NULL=ccc]
        ROOT VIRTUAL BLOCK NUMBER=nnnn  ROOT LEVEL=nnn
        1ST DATA BUCKET VIRTUAL BLOCK NUMBER=nnn
        KEY NAME:       ccccccccccccccccccccccccccccc
        POSITION:       nnn
        SIZE:           nnn
        TOTAL KEY SIZE=nnn        MINIMUM RECORD LENGTH=nnn
        DATA LEVEL FILL SIZE=nnnn        INDEX LEVEL FILL SIZE=nnnn
        DATA AREA=nn     INDEX AREA=nn  LOWEST LEVEL AREA=nn

ALTERNATE KEY:  n        type
        [NO] DUPLICATES changes          [NULL=ccc]
        ROOT VIRTUAL BLOCK NUMBER=nnnn  ROOT LEVEL=nnn
        1ST DATA BUCKET VIRTUAL BLOCK NUMBER=nnn
        KEY NAME:       ccccccccccccccccccccccccccccc
        POSITION:       nnn
        SIZE:           nnn
        TOTAL KEY SIZE=nnn        MINIMUM RECORD LENGTH=nnn
        DATA LEVEL FILL SIZE=nnnn        INDEX LEVEL FILL SIZE=nnnn
        DATA AREA=nn     INDEX AREA=nn     LOWEST LEVEL AREA=nn
                                          .
                                          .
                                          .
```

9.4.3 Call and Termination

9.4.3.1 Permanently Installed Utility

DSP [*command string*]

If you include a command string, the utility attempts to execute it and then returns control to the current keyboard monitor or command interpreter.

If you do not specify a command string, RMSDSP assumes control of the user interface and prints the prompt:

```
DSP>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSDSP has executed a command string, it reprints the prompt.

9.4.3.2 Uninstalled Utility

RUN $RMSDSP

RMSDSP assumes control of the user interface and prints the prompt:

```
DSP>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSDSP has executed a command string, it reprints the prompt.

9.4.4 Command String

Like all RMS-11 utilities, RMSDSP requires ANSI-standard labels on magnetic tapes. You must mount a tape volume to inform your operating system's file processor that you are accessing an ANSI-standard tape. See Appendix F for more details on RMS-11 magnetic tape handling.

If you do not mount your tape volume, RMSDSP does not accept wild card specifications and responds to an explicit file-name with the tape device's attributes.

9.4.4.1 General Form

[*outfile*=]*infile*[/*switch*]..][,*infile*[/*switch*]..]...]

where:

outfile is the filespec of a file to be created by the RMSDSP utility. RMSDSP then writes the attributes or contents of the specified input file(s) into the outfile. The default version of the outfile is the highest existing version number plus one.

If an outfile specification is not included in the command string, RMSDSP prints its information on the calling terminal.

No wild cards are allowed in the outfile specification.

infile is the filespec of either:

- Any file whose RMS-11 attributes you want displayed.
- A back-up container file on magnetic tape whose contents you want displayed. You must include the BP switch (see "Switches," Section 9.5.4.2) to indicate that the files are container files.

Wild cards are permitted in any field of the infile specification.

switch may be a code shown in Table 9-3 and described under "Switches" Section 9.5.4.2.

NOTE

A command string may also consist of the word "HELP" or a question mark (?). RMSDSP responds with a HELP message.

Table 9-3: RMSDSP Utility Switches

Switch	Description	Default
/ID	Identify current version of RMSDSP.	No id.
/BP	List contents of back-up container file(s).	Display attributes of magtape files.
/FU	Display all indexed file attributes.	Basic display only.

9.4.4.2 Switches

- ID causes RMSDSP to print its current version numbers in the form:

  ```
  DSP -- VERSION 1.8nn
  ```

  where *nn* is the patch level of the utility (see "Patch Level," Section 9.0.2).

  This switch may appear alone as a command string.

- BP causes RMSDSP to read the specified input files as magnetic tape container files (see "Command String" in RMSBCK, Section 9.1.4) and display the names of the RMS-11 back-up files contained therein. All infiles specified must be container files.

  Default If you do not include the BP switch,RMSDSP prints the attributes of the magtape files (see Section 9.5.2.2)

- FU causes RMSDSP to list the full attributes of an Indexed file, including definitions for individual keys and attributes of and number of unused blocks in areas, in addition to the basic data file attributes shown when the FU switch is not specified (see section 9.5.2.5). Area information is printed only when more than one area is defined.

  Default If you do not include the FU switch, RMSDSP lists only the number of keys defined for Indexed file(s) and no area information (see Section 9.5.2.4).

9.4.4.3 Command String Examples

- FILE.DAT

  RMSDSP prints on the calling terminal the attributes of the file named FILE.DAT found on SY:. Attributes are displayed only for the highest available version of the file.

- ATT.LST=*.*/FU

  RMSDSP creates the file ATT.LST and writes into it a full listing of attributes for all files on SY: in the default account.

- MT0:ALPHA.BKP,BETA.BKP/BP

  RMSDSP lists on the calling terminal the names of all files within the two container files ALPHA.BKP and BETA.BKP on the tape volume mounted on MT0:.

- A scratch tape is mounted on a TE-16 9-channel tape drive that is part of a PDP-11 computer system running under the RSX-11M operating system.

  The operator allocates the tape drive, then initializes the tape using ANSI-standard labels with the commands:

  ```
  ALL MM1:
  INITVOLUME MM1:DSPTST
  ```

The operator tries to examine the empty, unmounted tape with RMSDSP:

```
DSP MM1:*.*
?DSP -- ERROR WITH WILDCARDS
DSP MM1:
?DSP -- SYNTAX ERROR -
```

Then the operator mounts the tape, notifying the F11ACP that the tape is ANSI-standard:

```
MOUNT MM1:DSPTST
```

Again, the operator examines the empty tape with RMSDSP:

```
DSP MM1:

MM1:[303,351].;1        FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        [RWED,RWED,RWED,RWED]
RECORD FORMAT:          FIXED=512
RECORD ATTRIBUTES:
FILE ATTRIBUTES:
      BLOCK SIZE=512

DSP MM1:A.B.BAD
?DSP -- SYNTAX ERROR - BAD
DSP MM1:A.BAD
DSP -- FILE NOT FOUND - MM1:[303,351]A.BAD
```

The operator copies four RMS–11 files onto the tape using RMSCNV, then examines the tape:

```
DSP MM1:*.*

MM1:MIXSIM.B2S;0        FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        [RWED,RWED,RWED,RWED]
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512

MM1:MIXROO.B2S;0        FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        [RWED,RWED,RWED,RWED]
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512

MM1:MIXAS1.B2S;0        FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        [RWED,RWED,RWED,RWED]
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512

MM1:MIXAS2.B2S;0        FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        [RWED,RWED,RWED,RWED]
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512
```

The file MIXSIM.B2S is listed first although it is the last file copied to the tape. The F11ACP provides directory information from the tape's current position to the end and then rewinds the tape and reads to that position again.

The operator logs off the RSX-11M system, thereby dismounting and deallocating MM1:. The operator removes the tape from the drive and carries it to another PDP-11 running under the RSTS/E operating system. After physically mounting the tape on another TE-16 9-channel drive, the operator logs onto the RSTS/E system and tries to examine the tape with RMSDSP:

```
DSP MM1:*.*

Ready

DSP MM1:

?DSP -- SYNTAX ERROR -
```

The operator mounts the tape, informing the File Processor that the tape is ANSI-standard:

```
MOUNT MM1:DSPTST

Magtape is in ANSI format
DENSITY IS 800 , PARITY IS ODD

Ready
```

Now the operator examines the mounted tape:

```
DSP MM1:*.*

MM1:MIXR00.B2S          FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        <60>
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512

MM1:MIXAS1.B2S          FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        <60>
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512

MM1:MIXAS2.B2S          FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        <60>
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512

MM1:MIXSIM.B2S          FILE ORGANIZATION:  SEQUENTIAL

FILE PROTECTION:        <60>
RECORD FORMAT:          VARIABLE
RECORD ATTRIBUTES:      CARRIAGE RETURN NO SPANNING
FILE ATTRIBUTES:
      BLOCK SIZE=512
```

9.4.5 Cautions

- RMSDSP does not print allocation information for magnetic files. If you want this data, use PIP.

9.5 RMSIFL Command Utility

9.5.1 Purpose

The RMSIFL command utility provides the fastest, most direct, and most efficient method of populating an empty RMS-11 Indexed file. The utility can read records from any type of RMS-11 file and load them into an Indexed file that you have created. Unlike RMSCNV, RMSIFL does not use the standard RMS-11 access methods to build the output file. Rather, RMSIFL exploits the basic structure of Indexed files (discussed in Chapter 5) to insert data records and construct indexes fast. As it does this, the utility creates optimally dense buckets at all levels of each index. RMSCNV, on the other hand, can only optimize the Primary index when the input file is sorted by the output file's Primary Key. However, you must use RMSCNV if the output file contains records or you do not have room for the sort work files.

9.5.2 Effect

RMSIFL reads records from an input file and loads them into an output Indexed file in a series of phases.

1. The utility examines the command string (described in Section 9.5.4), rejecting it for syntax and/or other errors. When it has a valid command string, RMSIFL determines if the input file must be sorted into ascending order by the output file's Primary Key.

   RMSIFL bypasses the sort phase (phase 2) and skips to phase 3 if:

   - You included the Infile Switch NOSO (described in Section 9.5.4.4).
   - If the key of reference specified for an Indexed input file is equal to or contains the output Primary Key, based on position only. Nonstring key types must match exactly, in starting position and length. String keys must start in the same position, but the output Primary Key can be the same length or shorter than the input key of reference.

     You specify key of reference with the KR Infile Switch (described in Section 9.5.4.4).

     Example The ten-byte string input Alternate Key 2 starts at byte 0 and is specified as the key of reference. The eight-byte string output Primary Key starts at byte 0. RMSIFL does not sort.

     Example The ten-byte string input Primary Key starts at byte 0. The eight-byte string output Primary Key starts at byte 3. RMSIFL sorts the input file.

Example The three-byte packed decimal input Alternate Key 1 starts at byte 43 and is specified as the key of reference. The three-byte packed decimal output Primary Key starts at byte 42. RMSIFL sorts the input file.

RMSIFL notifies you that it has started processing by printing the following message:

```
PRIMARY KEY:
```

2. RMSIFL reads the input file in Sequential Access Mode, depending on the organization of the input file:

Input File Organization	Processing
SEQUENTIAL	RMSIFL reads the records in the Sequential Access Mode, starting with the first record in the file.
RELATIVE	RMSIFL reads the records in the Sequential Access Mode, starting with record cell 1.
INDEXED	RMSIFL reads the records in the Sequential Access Mode, following the key of reference.

RMSIFL passes each record to modules extracted from the SORT-11 utility. These modules order the input records by ascending value in the Primary Key field of the output file.

At the beginning of this phase, RMSIFL allocates disk space for three sort work files, each 1.5 times the input file size.

NOTE

Performance in this phase depends entirely on the SORT-11 modules. See the *PDP-11 SORT Reference Manual* for more information on the space required for temporary sort files and guidance on the factors affecting sort performance.

RMSIFL notes this phase by printing the following messages:

```
SORT HAS STARTED
SORT MERGE PHASE HAS FINISHED
```

If an error occurs while the SORT-11 modules are operating, RMSIFL terminates with the following message:

```
SORTS ERROR CODE IN OCTAL:nnnnn
```

Refer to the *PDP-11 SORT Reference Manual* for a description of the SORT-11 error codes.

At the end of this phase, RMSIFL deletes the sort work files.

3. RMSIFL loads the ordered input records into the output file, constructing Level 0 of the Primary index (discussed in "Data," Section 5.2.1) a bucket at a time. The manner in which RMSIFL reads input records depends on whether the input file was sorted in phase 2:

   - If RMSIFL sorted the input file, the utility accepts the input records one at a time from the sort modules. This technique avoids the need for a temporary file containing the input records sorted in output Primary Key order.
   - If RMSIFL did not sort the input file, the utility reads the records from the input file.

   RMSIFL examines each input record to ensure that it is compatible with the output file:

   - If the record is not compatible, RMSIFL processes it as an *exception record* according to the switch you specified in the command string. See the discussion of the ER:[*filespec*] and NOER Infile Switches in Section 9.5.4.3 for the causes of exception records and their handling.
   - If the record is compatible, RMSIFL inserts it into a Level 0 bucket being built in memory.

   As the utility places each record, the utility extracts the values for each of the Alternate Keys defined for the file, if any:

   a. RMSIFL combines the Alternate Key value with the Record's File Address (RFA) of the user data record that supplied the key value. RFAs are six bytes long. The utility then appends a byte with the key number.

   b. These fixed-length Alternate Key records are written into a separate temporary file marked for delete. RMSIFL creates this file with an initial allocation of 100 blocks and a Default Extension Quantity of 100 blocks. You can calculate the final size of this temporary file (ALQ) as follows:

      ALQ = (NDRF * NAK * (LKS + 7 + PAD))/512

      where:

      NDRF is the number of user data records in the input file,

      NAK is the number of Alternate Keys defined for the output file,

      LKS is the size of the longest Alternate Key defined for the output file (and shown by RMSDSP), in bytes, (plus 1 for packed decimal keys) and

      PAD is the number of null bytes added to word align the record:

      PAD = 0 if LKS is an even number

      PAD = 1 if LKS is an odd number

Remember to round ALQ to the nearest multiple of 100.

During this process, the utility assembles the index levels of the Primary index as well. RMSIFL puts an index record into a Level 1 bucket for each Level 0 bucket it writes to the file. When it writes a Level 1 bucket to the file, the utility inserts an index record in a Level 2 bucket, and so on, up to and including the Root.

4. RMSIFL sorts the Alternate Key temporary file by key number and ascending key value. Nonstring key data types are converted into a string-like representation for sorting only.

   At the beginning of this phase, RMSIFL allocates the disk space for sort work files, each 1.5 times the size of the Alternate Key temporary file.

   The utility notes this phase by printing the following messages:

```
ALTERNATE KEY(S)
SORT HAS STARTED
SORT MERGE PHASE HAS FINISHED
```

   At the end of this phase, RMSIFL deletes the sort work files.

5. RMSIFL builds the Alternate indexes one at a time, from Level 0 up to the Root, using the sorted temporary file for input.

   The utility notes the construction of each Alternate index by printing the following message:

```
ALTERNATE KEY(S) nnn
```

   where:

   nnn starts with 1.

6. RMSIFL prints the following messages when the output file is completely built, then terminates the processing of the current command string.

```
NUMBER OF INPUT RECORDS: nnnnnnnnn
NUMBER OF OUTPUT RECORDS: nnnnnnnnn
NUMBER OF EXCEPTION RECORDS: nnnnnnnnn
```

NOTE

RMSIFL extends the output file if it cannot contain the input records and the index(es). The utility minimizes this overhead by extending the file a set quantity each time it requires space. This quantity is the integral multiple of bucket size greater than 50. On RSTS/E, RMSIFL forces a contiguous output file to be noncontiguous, then extends it.

9.5.3 Call and Termination

9.5.3.1 Permanently Installed Utility

IFL [*command string*]

If you include a command string, the utility attempts to execute it and then returns control to the current keyboard monitor or command interpreter.

If you do not specify a command string, RMSIFL assumes control of the user interface and prints the prompt:

```
IFL>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSIFL has executed a command string, it reprints the prompt.

9.5.3.2 Uninstalled Utility

RUN $RMSIFL

RMSIFL assumes control of the user interface and prints the prompt:

```
IFL>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSIFL has executed a command string, it reprints the prompt.

9.5.4 Command String

9.5.4.1 General Form

outfile[*/switch*]..]=*infile*[*/switch*]..]

where:

outfile is the filespec of an existing Indexed file that is to receive records from the input file. Wild card characters are not permitted in any field of the specification. The default version is the highest version.

RMSIFL checks for the following conditions in the output file. If they are not met, the utility prints the indicated error message and terminates execution of the command string.

- The output file must exist before you run the utility; otherwise, RMSIFL prints the message:

  ```
  ?IFL -- FILE NOT FOUND - filespec
  ```

- The output file must have no data in it besides the Prologue written when the file was created. Otherwise, RMSIFL prints the message:

  ```
  ?IFL -- INDEXED OUTPUT FILE NOT EMPTY
  ```

- The output file must be an RMS-11 Indexed file. Otherwise, RMSIFL prints the message:

  ```
  ?IFL -- OUTPUT FILE MUST BE AN INDEXED FILE
  ```

- The output file must not have a bucket size greater than five blocks. Otherwise, RMSIFL prints the message:

  ```
  ?IFL -- THERE IS NOT ENOUGH IFL MEMORY FOR THE CURRENT
  COMMAND LINE
  ```

- The output file cannot have more than ten keys defined. Otherwise, RMSIFL prints the message:

  ```
  ?IFL -- TOO MANY KEYS - filespec
  ```

infile is the filespec of the input file that is the source of the records written to the output file. Wild card characters cannot appear in any field of the filespec.

You can specify only one input file, but it can have any RMS-11 file organization. RMSIFL does not affect the input file in any way.

The utility opens the input file access read, allow no-write. Therefore, no writing task can access the input file while RMSIFL is operating.

The input file cannot have more than ten keys defined. Otherwise, RMSIFL prints the message:

```
?IFL -- TOO MANY KEYS - filespec
```

You cannot use a back-up file produced by RMSBCK as input. RMSIFL terminates with the following error message:

```
?IFL -- OPEN ERROR ON FILE - filespec
?IFL -- FATAL RMS ERROR  STS = -01424,STV = 00000
```

switch may be a code shown in Table 9-4 and described in Section 9.5.4.2 through 9.5.4.4.

NOTE

A command string may also consist of the word "HELP" or a question mark (?). RMSIFL responds with a HELP message.

9.5.4.2 Global Switches —

- ID causes RMSIFL to print its current version number, in the form:

  ```
  IFL -- VERSION 1.8nn
  ```

  where:

  nn is the patch level of the utility (see "Patch Level," Section 9.0.2).

  This switch may appear alone as a command string.

Table 9-4: RMSIFL Utility Switches

Type	Switch	Description	Default Process
String	? or HELP	Print HELP message.	No help.
Global	/ID	Identify current version	No id.
Outfile	/ER	Write exception records on terminal.	Same.
	/ER:*filespec*	Write exception records into specified file.	Write exception records on terminal.
	/LO	Honor fill numbers.	Fill buckets.
	/NOER[:S]	Stop processing immediately if input record is incompatible.	Write exception records on terminal.
	/PD[:[#]*x*]	Pad input records to output record length.	Handle input as exception record if different lengths.
	/TR	Truncate input records to output record length.	Handle input as exception record if different lengths.
Infile	/CL:*nnn*	Set sort work files' clustersize.	Use algorithm.
	/DE:*dvn*[:*dvn*[:*dvn*[:]]]	Reassign devices for sort work files.	Create and use sort work files on SY:
	/KR:*n*	Key of reference number.	Primary Key (*n*=0).
	/NOSO	Do not sort before loading.	Sort input file before loading.

9.5.4.3 Outfile Switches

- ER:[*filespec*] directs RMSIFL to continue processing if it encounters a record in the input file that cannot be written into the output file. Exception records exist for one of the following reasons:

  - You specified the NOSO Infile Switch and the utility found a record that was not in ascending order by output Primary Key value. The error message is:

    ```
    IFL -- EXCEPTION RECORD:  RECORD OUT OF SEQUENCE
    ```

  - You did not allow duplicates in at least one key of the output file, but an input record contains a duplicate value in that key field. The error message is:

    ```
    IFL -- EXCEPTION RECORD:  DUPLICATE KEY WHERE NOT ALLOWED
    ```

  - An input record is not long enough to contain the Primary Key of the output file. The error message is:

    ```
    IFL -- EXCEPTION RECORD:  RECORD TOO SHORT FOR PRIMARY KEY
    ```

- An input record is too long to fit in a bucket of the output file. The error message is:

```
IFL -- EXCEPTION RECORD:  RECORD TOO LONG FOR BUCKET SIZE
```

- An input record is not compatible with the output file's fixed-length record format and you have not specified the PD and/or TR switches. The error message is:

```
IFL -- EXCEPTION RECORD:  RECORD DOESN'T FIT FIXED LENGTH
```

- An input record is longer than the output file's variable-length Maximum Record Size and you have not specified the TR switch. The error message is:

```
IFL -- EXCEPTION RECORD:  RECORD TOO LONG FOR MAXIMUM RECORD SIZE
```

- An input record contains an illegal packed decimal key: a digit half-byte is greater than 9 or the sign half-byte is less than 10. The error message is:

```
IFL -- EXCEPTION RECORD:  RECORD CONTAINS AN ILLEGAL PACKED KEY
```

If RMSIFL encounters an exception while processing the Alternate index(es), it flags as deleted the data record in the Primary Level 0. The utility does not remove the record from the file. The record may be compressed during later use of the Indexed file, as described in "Key Selection," Section 6.2.

You can add corrected exception records to the output file after RMSIFL terminates with either an application program or RMSCNV (possibly using your terminal as the input device; see Section 9.2.5).

If you do not specify an argument (*filespec*), RMSIFL prints the appropriate error message, then the exception record on your terminal.

If you specify a nondisk file or a terminal, the utility creates it as an RMS–11 Sequential file with variable-length records when the first exception record is encountered. Then, RMSIFL writes both the error message and the exception record into the file. RMSIFL uses this procedure for every exception record until it finishes processing.

If you specify a disk file, the utility creates it as an RMS–11 Sequential file with VFC records when the first exception record is encountered. Then, RMSIFL writes the exception record into the variable portion of the record and a four-byte exception code into the fixed control area of the record. The codes are:

Code	Meaning
001:	Record is out of Primary Key sequence
002:	Record contains an illegal duplicate value
003:	Record is too short to contain output Primary Key
004:	Record is too long to fit in output bucket
005:	Record is not correct size for output fixed-length record
006:	Record is too long for output Maximum Record Size
007:	Record contains an illegal packed decimal key.

See also the NOER switch.

Default If you do not include the ER switch, RMSIFL prints the appropriate error message, then the exception record on your terminal.

- LO directs RMSIFL to write records into buckets according to the fill numbers established when the file was created. See "Data Allocation" in RMSDEF, Section 9.3.4.6, and "Fill Number," Section 6.7.2.1.

  Default If you do not include the LO switch, RMSIFL inserts as many records as possible into each bucket.

- NOER[:S] directs RMSIFL to print an error message, delete the output file, and terminate processing immediately if it encounters an input record that cannot be written into the output file; see the ER:[*filespec*] switch for reasons and error messages.

  RMSIFL deletes the output file unless you specify the Save argument (:S) in the NOER switch. Termination at this point leaves the output file incomplete: it is not a valid file of any organization and cannot be reused for any RMS-11 purpose.

  Example Your output file is large and contiguous and you don't want to lose any of that contiguous space to other users.

  Default If you do not include the NOER switch, RMSIFL responds according to the version of the ER switch you used.

- PD[:[#]*x*] directs RMSIFL to pad records read from the input file to the output file's record length before writing them to the output file. Padding character is specified as follows:

Switch	Character
PD	NULL
PD:*x*	*x* is ASCII A-Z, 0-9, or special character except #, ?, and @
PD:#*x*	*x* is octal number 000 through 377 (40 for SPACE, 43 for #, 77 for ?, and 100 for @)

  You use the PD switch only when the output file specifies fixed-length records.

  Default If you do not include the PD switch, and the input records are shorter than the output file's records may be, RMSIFL treats them as exception records.

  See ER and NOER switch in this section.

- TR directs RMSIFL to truncate records read from the input file to the output file's Maximum Record Size before writing them to the output file. The trailing bytes of the records are truncated.

  Default If you do not include the TR switch, and the input records are longer than the output file's record may be, RMSIFL treats them as exception records.

  See ER and NOER switch in this section.

9.5.4.4 Infile Switches —

- CL:*nnn* specifies a clustersize for RMSIFL to use when it creates work files to sort input records and its special Alternate Key records. Large files require large clustersizes so that the User File Directory can contain the retrieval pointers.

  Example With a pack clustersize of 16, a file 80,000 blocks long cannot be created.

  The clustersize you specify must be a 0 or a power of two from 1 through 256; otherwise, RMSIFL terminates with the following message:

  ```
  ?IFL -- THE CLUSTERSIZE IS GREATER THAN 256, OR NOT A POWER OF 2
  ```

  If you specify a value less than the pack clustersize for the disk on which the sort work files are created, RMSIFL uses the pack clustersize.

  Default If you do not include the CL switch, RMSIFL uses the following algorithm to calculate a clustersize:

  1. Evaluate the expression:

     (FILESIZE * 1.5)/512

     where:

     FILESIZE is the size of the input file or the Alternate Key temporary file.

  2. Find the nearest power of two less than the value of the expression and not greater than 256.

- DE:*dvn*[:*dvn*[:*dvn*[:]]] directs RMSIFL to reassign the devices where it allocates the three sort work files. You can specify from one to three disk devices with the switch, where *dvn* is a physical device name and number or a logical name up to six alphanumeric characters. You separate device names by a colon (:); the final colon is optional.

  If you include the DE switch in the command string, RMSIFL reassigns all three sort work files:

  - If you specify one device name, the utility reassigns the three work files to that device.
  - If you specify two device names, the utility reassigns one work file to the first device and two work files to the second.
  - If you specify three device names, the utility reassigns one work file to each device.

  For optimum performance, you should assign each sort work file to a different device. However, if you have only two devices available, assign the first and third files to one device and the second file to the other device.

  Example /DE:DM0:DM1:DM2

  Example /DE:DM0:DM1:DM0

Default RMSIFL allocates the sort work files on SY:.

- KR:*n* directs RMSIFL to read an input Indexed file according to the key specified by *n*, where *n* is 0 for the Primary Key, 1 for the First Alternate Key, and so on, up to 9.

  The KR switch can eliminate the need for sorting the input file; see "Effect," Section 9.5.2.

  If you use the KR switch with a non-Indexed file, RMSIFL terminates with the following message:

  ```
  ?IFL -- /KR NOT ALLOWED FOR SEQUENTIAL OR RELATIVE FILE
  ```

  Default If you do not include the KR switch, and the input file is Indexed, RMSIFL reads the file via the Primary index.

- NOSO directs RMSIFL to bypass its sort phase of processing because you have already sorted the input file. If the input file is not in order by ascending value of the output file's Primary Key, RMSIFL's response depends on whether you have included the NOER switch or a version of the ER switch (see Section 9.5.4.3).

  Default If you do not include the NOSO switch, RMSIFL sorts all records from the input file into the proper order. See "Effect," Section 9.5.2.

9.5.5 Cautions

- RMSIFL ignores the fixed control area of VFC records in either input or output file.
- When both the input and output files have fixed-length records, RMSIFL requires either the TR or PD switch if the fixed record lengths are not equal. If neither switch is specified, the utility terminates with the error message:

  ```
  ?IFL -- INPUT AND OUTPUT RECORD SIZES DO NOT CORRESPOND
  ```

- When the input file contains variable-length records and the output file contains fixed-length records, RMSIFL requires both the TR and the PD switches. If both are not specified, the utility terminates with the error message:

  ```
  ?IFL -- SWITCH /TR OR /PD OR BOTH ARE NEEDED FOR THIS RECORD
  ```

9.6 RMSRST Command Utility

9.6.1 Purpose

The RMSRST utility reverses the action of the RMSBCK utility. Where RMSBCK creates back-up copies of files, RMSRST takes those back-up files as input and produces standard RMS-11 files as output. The structure, content, and attributes of these restored files are those of the original files when they were backed up.

9.6.2 Effect

RMSRST provides you with the following capabilities:

Explicit and Implicit File Selection
: You restore a single file or a collection of files with each call of the RMSRST utility. Multiple files are specified explicitly with a string of file specifications or implicitly with wild cards. You may select from a wild card group of files by name or on the basis of back-up date.

Choice of Restoration Account and/or Volume
: You may restore file(s) into any account on any disk device to which you have proper access privileges.

Extended Diagnostic Messages
: When the utility operates in the Query mode, RMSRST gives you the option of continuing or terminating processing when special errors occur (described in Section 9.6.4.2). You thereby ensure that the utility does not terminate the processing of a collection of files due to errors in the processing of one file. If the Query mode is explicitly disabled, RMSRST terminates when it encounters the special errors. See the QU switch in Section 9.6.4.2 for examples of diagnostic messages.

Data Integrity Checks
: RMSRST can perform extensive data integrity checking as each back-up file is restored. You can direct the utility either to read file contents after they are restored or to both read and check file contents on a byte-by-byte basis after they are written. RMSRST automatically retries read errors if processing continues to a normal termination.

  These integrity checks give you a choice about how reliable the restored files are:

  - You can rely on software and hardware accuracy and restore your files in minimum time.
  - You can verify that you can read the restored files, adding the read time to the minimum time.
  - You can guarantee that the restored files can be read and that they match the back-up files, adding both read and compare times to the minimum time.

Summary Listing
: You can specify that RMSRST list a summary of the files selected and restored and of error messages. This summary can be generated at your terminal or written into a file specified in the utility's command string. See the SL switch in Section 9.6.4.2 for a sample of the summary.

9.6.3 Call and Termination

9.6.3.1 Permanently Installed Utility —

RST [*command string*]

If you include a command string, the utility attempts to execute it and then returns control to the current keyboard monitor or command interpreter.

If you do not specify a command string, RMSRST assumes control of the user interface and prints the prompt:

```
RST>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSRST has executed a command string, it reprints the prompt.

9.6.3.2 Uninstalled Utility —

RUN $RMSRST

RMSRST assumes control of the user interface and prints the prompt:

```
RST>
```

You may type a command string or CTRL/Z to terminate the utility. When RMSRST has executed a command string, it reprints the prompt.

9.6.4 Command String

9.6.4.1 General Form —

outfile[/*switch*]..]=*infile*[/*switch*]..][,*infile*[/*switch*]..]...]

where:

outfile is the filespec of a file(s) to be restored by the utility. The file-name and extension must be wild cards, that is, *.*; therefore, you cannot rename files as they are restored by RMSRST. The version can be omitted or can be a wild card. In both instances, the version used is that of the original file.

RMSRST interprets the account field as follows:

Account Number	Files Restored To
None	Default account.
[*,*]	Original accounts from which the files were backed up; the numbers are stored in the back-up files.
Explicit	Specified account.

infile is the file specification of a back-up container file or a disk file in back-up format. You can use wild cards to specify multiple files. If

version is omitted and the input file resides on magnetic tape (a container file), the following defaults are applied:

- If wild cards appear in either or both file-name and extension, all versions of all files satisfying the wild cards are selected.
- If both file-name and extension are explicitly specified, the first physically occurring instance of the file with version 0 is selected.

NOTE

To restore a specific data file from magtape, you must use the container file-name as the infile specification and specify the data file-name in the SE infile switch.

switch may be a code shown in Table 9–5 and described in Sections 9.6.4.2 through 9.6.4.4.

NOTE

A command string may also consist of the word "HELP" or a question mark (?). RMSRST responds with a HELP message.

Table 9–5: RMSRST Utility Switches

Type	Switch	Description	Default Process
String	? or HELP	Print HELP message.	No help.
Global	/ID	Identify current version of RMSRST utility.	No id.
	/QU	Enable Query mode.	QU
	/SL[:*filespec*]	Provide summary of activity.	No summary.
Outfile	/FR	Change protection codes.	Original protection.
	/RA*	Read after writing.	NORA
	/RC*	Check after writing.	NORC
	/SU	Supersede existing files.	NOSU
Infile	/BD:*dd-mon-yy*	Restore disk files based on date of back up.	No date checking.
	/OA:[*act nbr*]	Select files based on original account.	OA:[*,*]
	/SE:*filespec* or /SE:(*filespec*,...)	Select specified file(s) for restoration from back-up container file.	SE:*.*;*

* The RA and RC switches are not available on RSTS/E. That operating system does not allow the RMSBCK task to rewind a magnetic tape device to beginning of file after the utility accesses that file for writing.

9.6.4.2 Global Switches —

- ID causes RMSRST to print its current version number, in the form:

```
RST -- VERSION 1.8nn
```

  where *nn* is the patch level of the utility (see "Patch Level," Section 9.0.2).

  This switch may appear alone as a command string.

- QU enables the Query mode (default). When the Query mode is enabled, the RMSRST utility allows you to continue or terminate processing when one of the following occurs:

  - A read error on an input file.
  - A read-after-write error on an output file when the RA switch is specified.
  - A check-after-write error when the RC switch is specified.
  - The table allocated internally for data integrity checks or automatic retry of read errors is full.

  When one of these errors occurs, the utility prints a diagnostic message specifying the type of error and the name of the file being processed. If you answer Y (yes), RMSRST continues processing the current file and command string. If you answer N (no), the utility terminates processing immediately, bypassing the rest of the command string.

  Example BCK -- CHECK AFTER WRITE ERROR ON OUTPUT FILE - *filespec*
  VBN *vbn1* TO *vbn2*, CONTINUE (Y,N)?

  Example BCK -- INTEGRITY CHECK TABLE FULL, CONTINUE (Y,N)?

  When you disable the Query mode through the NOQU switch, RMSRST prints the appropriate message and terminates processing after it encounters one of the errors.

  Default If you specify no version of the QU switch, Query mode is enabled.

- SL[:*filespec*] directs RMSRST to provide a summary of processing, including:

  - the command string
  - the names of files successfully restored
  - the names of files not restored, with associated error messages
  - diagnostic messages produced in Query mode
  - a summary of input errors and of errors remaining in restored output files following automatic retry of data integrity checks (when you specify RA or RC).

  If a file specification is not included in the switch, RMSRST prints the summary listing on your terminal. If a file is specified, the utility creates the file and writes the summary listing into it.

Example
```
RMSRST - VERSION 1.800 22-JAN-1979 14:19:00
*.*=DB1:*.*/SL:RST.LST

RST--FILEPROCESSINGCOMPLETE-SY:[303,357]SPEWDOC.CMD;43
RST -- FILE PROCESSING COMPLETE - SY:[303,357]TECO.TEC;1
RST -- FILE ALREADY EXISTS - SY:[303,357]USERGUIDE.RVW;3
RST -- FILE ALREADY EXISTS - SY:[303,357]REFMNLUPD.RVW;6
```

Default If you specify no version of the SL switch, RMSRST only prints messages on your terminal if it encounters a fatal or non-fatal error condition.

9.6.4.3 Outfile Switches —

- FR directs RMSRST to change the restored file's system protection codes. RMSRST modifies the protection code in the back-up file when it restores the data to disk. If the files are restored with the operating system they were backed up with, the utility changes the protection code so the default account, the one operating the utility, becomes the owner of the files. If the restoring operating system is different, RMSRST sets the protection code to the system default.

NOTE

If files are restored to an account other than the one used when they were backed up and FR is not specified, the utility assigns the files to the indicated account (see "General Form," Section 9.6.4.1), but the protection code is not changed, that is, the original owner is still the owner, and so on.

Default If you do not include the FR switch, RMSRST does not change the protection code from the value copied when the file was backed up.

- RA directs RMSRST to read the restored file after writing it. The utility reads back each block of a file after it has been completely restored: if the device hardware detects a read error, RMSRST's reaction depends on whether the Query mode is enabled or not. See QU under "Global Switches," Section 9.6.4.1.

NOTE

You cannot use the RA and RC switches in the same command string. RMSRST prints the following error message and terminates execution of the command string.

```
?BCK -- CONFLICTING OPTION - /sw
```

where *sw* is RA or RC depending on which is second in the command string.

Default If you do not include the RA switch, RMSRST does no read-after-writer data integrity checking.

- RC directs RMSRST to check the restored file after writing it. After a file has been completely restored, the utility reads each block of the file back into memory and compares it with the corresponding block from the input back-up file. If an error occurs during this process (a hardware read error or a mismatch between the contents of the two blocks), RMSRST's reaction to the error depends on whether the Query mode is enabled or not. See QU under "Global Switches," Section 9.6.4.1.

  See note under RA in this section.

  Default If you do not include the RC switch, RMSBCK does not check the restored file after writing it.

- SU causes RMSRST to supersede each file in the output account with the same file-name and extension as a file being restored.

  Default If you do not include the SU switch, RMSBCK does not supersede files in the output account. If the utility encounters a file with the same file-name, extension, and version as an input file, it prints the following nonfatal error message and continues processing.

  ```
  RST -- FILE ALREADY EXISTS - filespec
  ```

9.6.4.4 Infile Switches —

- BD:*dd-mon-yy* directs RMSRST to restore disk files based on back-up date. You can combine this switch with wild cards to identify specific files within a group of disk files. RMSRST selects for processing only those files that satisfy the wild card specification and that were backed up on the specified date.

  The BD switch cannot be used if the back-up files are stored on magnetic tape. Since RMSRST cannot access the creation date of a magnetic tape file, it prints the following error message if BD is included in a magnetic tape infile specification:

  ```
  ?RST - ILLOGICAL USE OF SWITCH - /BD
  ```

  Default If you do not include the BD switch, RMSRST applies no date criterion to the selection of files for restoration.

- OA:[*act nbr*] directs RMSRST to select files based on the account from which the original files were backed up. You can combine this switch with wild cards to identify specific files within a group of disk or tape files. RMSRST selects for processing only those files that satisfy the wild card specification and that were backed up from the specified account.

  Default If you do not include the OA switch, RMSRST applies no account number criterion to the selection of files for restoration.

- SE:*filespec* or SE:(*filespec*,...) directs RMSRST to select specified file(s) for restoration from magnetic tape container files only. This switch allows you to select explicitly one or more files from container files. The switch argument, *filespec*, can contain only file-name, extension, and version; any can be a wild card. Up to ten filespecs may be included, enclosed in parentheses and separated by commas if more than one.

  If version is omitted from *filespec*, RMSRST applies the following defaults:

  - All versions of the file represented by *filespec* are restored if any of the following are true:
    - The file-name or extension in *filespec* is a wild card.
    - The OA switch is not specified for the container file.
    - The OA switch is specified for the container file, but contains wild cards.
  - The first physically occurring version of the file represented by *filespec* is restored if:
    - Neither the file-name or extension in *filespec* is a wild card.
    - The OA switch is specified and contains no wild cards.

  Default If you do not include the SE switch, RMSRST uses all contained files as input to the restoration process.

9.6.4.5 Command String Examples —

- *.*=MT0:ALPHA.DAT

  RMSRST searches the volume mounted on MT0: for the first physically occurring version of container file ALPHA.DAT. The utility restores the contained files to their original form into the current account on SY:. Since the SU switch was not specified, the current account on SY: must not contain a file with the same file-name, extension, and version as a file being restored: RMSRST does not restore duplicate files; instead, it prints the following error message and continues processing.

  ```
  RST -- FILE ALREADY EXISTS - filespec
  ```

- *.*/SU=MT1:*.*

  RMSRST searches the volume mounted on MT1: for all container files. For each such file, it restores the contained files into the current account on SY:. If the current account contains a file with the same file-name, extension, and version as a file being restored, the disk file is superseded.

- *.*/SU/SL=DK0:*.*/BD:30-OCT-78,MT0:*.*

  RMSRST reads all files under the current account on DK0: that were backed up on October 30, 1978, and restores them to the current account on SY:. Then the utility restores files within all container files on the tape

mounted on MT0:. Throughout the process, the utility supersedes any existing file with the same file-name, extension, and version as the file being restored. RMSRST also prints a summary of activity at your terminal.

- [100,10]*.*/RC/SU=MT0:*.*/SL:[100,10]REST.LST

  RMSRST restores all files within all container files on the tape on MT0: into account [100,10] on SY:, superseding disk files with the same file-names, extensions, and versions. The utility checks after writing each restored file. Since Query mode is enabled by default, diagnostic messages may appear at your terminal as a result of these checks. The utility also creates a file named REST.LST in the output account and writes a summary of its activity into it.

- *.*=MT2:MASTER.BKP/SE:(PARTS.DAT,CUST.FIL)

  RMSRST searches the volume mounted on MT2: for the first physically occurring instance of the container file MASTER.BKP. When this file is found, RMSRST searches MASTER.BKP for all versions of files PARTS.DAT and CUST.FIL. These files are restored to their original form in the default account on SY:.

- *.*/SU/FR=MT0:ACCT.BKP/SE:*.DAT */OA:[100,11]

  RMSRST restores all versions of all files with an extension of DAT and an original account of [100,11] into the default account on SY: from container file ACCT.BKP on MT0:, changing the protection code as it does. The utility supersedes files in the default account if necessary.

- DK2:[*,*]*.* *=MT1:*.*

  RMSRST replaces all files within all container files on MT1: into their original accounts on DK2:.

- [200,11]*.*/FR=MT3:A200.BAC/OA:[200,11]

  RMSRST rewrites all files with an original account of [200,11] in container file A200.BAC into account [200,11] on SY:. The tape on MT3: was created with the RMSBCK utility on the RSTS/E operating system, but the files are being restored with RSX-11M. RMSRST therefore changes the protection code on each file to the system default as the file is written onto the system device.

9.6.5 Cautions

- Files can be restored only on disk devices.
- A system disk cannot be restored as a bootable volume.
- A file contiguous when backed up will not be contiguous when restored. However, you can use RMSDEF and RMSCNV to make a contiguous file.
- Do not use either the infile or outfile specifications as an argument in the SL switch.

Appendix A
RMS–11 and the Operating Systems

RMS–11 is available as an interface between user application programs and data storage devices on the following DIGITAL operating systems.

A.1 IAS

A.1.1 RMS–11 Restrictions on IAS

None.

A.1.2 IAS Restrictions on RMS–11

None.

A.1.3 Compatibility with Other File Managers

In addition to RMS–11, File Control Services (FCS–11) is available on IAS systems. FCS–11 and RMS–11 can reside on the same computer system and one program can use both record access systems with caution. The structure of RMS–11 Sequential files is identical to that of files created by FCS–11. Therefore, you can use FCS–11 to access RMS–11 Sequential files and vice-versa.

However, if you use FCS–11 to open an RMS–11 Relative or Indexed file, FCS–11 will logically destroy the file when it closes the file.

Within the Sequential organization, the following RMS–11 record formats are compatible with FCS–11:

RMS–11	FCS–11
Fixed	FLR
Variable	VLR
VFC	SVLR (fixed control area must be equal to two bytes)

A.1.4 Asynchronous Operations

IAS allows tasks to perform operations asynchronously. You use this technique so that your task can perform functions during I/O operations, rather than wait until the I/O activity is over.

A.1.4.1 RMS-11 Synchronous Environment — By default, the Task Builder incorporates the synchronous RMS-11 routines into tasks. These routines do **not** enable or disable Asynchronous System Traps (ASTs). You can program routines that run at AST level; however, those routines must not:

- initiate an RMS-11 operation while another is still in progress. RMS-11 synchronous routines are vulnerable to this type of procedure: the results of the interrupted operation are unpredictable. Therefore, you should avoid using RMS-11 routines at program level and AST level at the same time. To enable simultaneous operations at the different levels, you must take special precautions to avoid interference between the operations.
- cause the task segment containing the RMS-11 routines to be overlaid while an RMS-11 operation is in progress. The results in such a situation are unpredictable.

A.1.4.2 RMS-11 Asynchronous Environment — You can task build with the RMS-11 Asynchronous Executive modules (see *Nonoverlaid RMS-11* in "Task Building with RMS-11 Routines," Section 8.1 for details), thereby creating the RMS-11 Asynchronous Environment. File operations must still be performed synchronously. In order to perform asynchronous record operations, you must also set up an asynchronous Record Access Block (RAB) and turn on the RB$ASY in the RAB ROP field. See the *RMS-11 MACRO-11 Reference Manual* for details.

When your task initiates an RMS-11 asynchronous record operation, RMS-11 disables Asynchronous System Traps (ASTs) while it is running. RMS-11 enables ASTs when it must wait and when it returns control to your program. While your program is in control during RMS-11 I/O operations, the program can disable ASTs for critical operations. However, the outstanding RMS-11 operation typically will not be able to complete until ASTs are enabled again.

The RMS-11 Asynchronous Environment restricts operations as follows:

- You can initiate asynchronous RMS-11 operations at AST level provided the associated Record Access Stream is not in use. However, operations begun at AST level cannot complete until the task reverts to program level. Therefore, any associated wait operation ($WAIT macro) must **not** be performed at AST level.
- Any completion routine associated with an asynchronous record operation is typically executed at AST level and is considered part of the record operation:
  - The completion routine cannot initiate a synchronous RMS-11 operation. RMS-11 returns the error code ER$AST.

- The completion routine cannot use the $WAIT macro. RMS-11 aborts the task.
- Asynchronous RMS-11 operations initiated by the completion routine cannot complete until control is returned to RMS-11. You do this by issuing the $RETURN macro as the end of the completion routine. RMS-11 resumes control and performs an exit from AST level. Control returns to your program, although an AST is still outstanding because of the asynchronous operation initiated by the completion routine.

- Be careful initiating synchronous RMS-11 operations from the AST level while outside a completion routine. You must ensure that no resource required by the operation is exclusively controlled by an operation initiated at program level. Since the operation performs at AST level, it cannot be interrupted: the operation can loop continuously, waiting for that resource to be freed.

  Example Your program initiates an asynchronous record operation on an Indexed file. When control returns to the program, it initiates a synchronous operation on that same Indexed file. If the asynchronous operation locks the file Prologue or a bucket required by the synchronous operation, the synchronous operation loops, repeating its request for the locked blocks. Since the asynchronous operation cannot resume to unlock the blocks, the synchronous operation will never break out of its loop.

A.2 RSTS/E

A.2.1 RMS-11 Restrictions on RSTS/E

RMS-11 does not allow switches on file specifications.

A.2.2 RSTS/E Restrictions on RMS-11

The following RMS-11 capabilities are not available on RSTS/E:

- Asynchronous I/O operations
- Magnetic tape volume handling after RMS-11 opens a file on the tape

  Example Data integrity switches on the RMSBCK and RMSRST utilities.

  Example The macros $REWIND (for magtape files only), $NXTVOL, and $SPACE.
- Multi-volume magnetic tape files
- Windowsize adjustment
- Extending contiguous files
- Placement control for file areas other than Area 0

A.2.3 Compatibility with Other File Managers

In addition to RMS-11, BASIC-PLUS performs file handling. A stream ASCII (terminal-format) file created by BASIC-PLUS can be read as an RMS-11 Sequential file with Stream record format as long as the file is null-filled from the end of the last record to the physical end-of-file.

A.3 RSX-11M

A.3.1 RMS-11 Restrictions on RSX-11M

None.

A.3.2 RSX-11M Restrictions on RMS-11

None.

A.3.3 Compatibility with Other File Managers

In addition to RMS-11, File Control Services (FCS-11) is also available on RSX-11M systems. FCS-11 and RMS-11 can reside on the same computer system and one program can use both record access systems with caution. The structure of RMS-11 Sequential files is identical to that of files created by FCS-11. Therefore, you can use FCS-11 to access RMS-11 Sequential files and vice-versa.

However, if you use FCS-11 to open an RMS-11 Relative or Indexed file, FCS-11 will logically destroy the file when it closes the file.

Within the Sequential organization, the following RMS-11 record formats are compatible with FCS-11:

RMS-11	FCS-11
Fixed	FLR
Variable	VLR
VFC	SVLR (fixed control area must be equal to two bytes)

A.3.4 Asynchronous Operations

RSX-11M allows tasks to perform operations asynchronously. You use this technique so that your task can perform functions during I/O operations, rather than wait until the I/O activity is over.

A.3.4.1 RMS-11 Synchronous Environment — By default, the Task Builder incorporates the synchronous RMS-11 routines into tasks. These routines do **not** enable or disable Asynchronous System Traps (ASTs). You can program routines that run at AST level; however, those routines must not:

- initiate an RMS-11 operation while another is still in progress. RMS-11 synchronous routines are vulnerable to this type of procedure: the results of

the interrupted operation are unpredictable. Therefore, you should avoid using RMS-11 routines at program level and AST level at the same time. To enable simultaneous operations at the different levels, you must take special precautions to avoid interference between the operations.

- cause the task segment containing the RMS-11 routines to be overlaid while an RMS-11 operation is in progress. The results in such a situation are unpredictable.

A.3.4.2 RMS-11 Asynchronous Environment — You can task build with the RMS-11 Asynchronous Executive modules (see *Nonoverlaid RMS-11* in "Task Building with RMS-11 Routines," Section 8.1 for details), thereby creating the RMS-11 Asynchronous Environment. File operations must still be performed synchronously. In order to perform asynchronous record operations, you must also set up an asynchronous Record Access Block (RAB) and turn on the RB$ASY in the RAB ROP field. See the *RMS-11 MACRO-11 Reference Manual* for details.

When your task initiates an RMS-11 asynchronous record operation, RMS-11 disables Asynchronous System Traps (ASTs) while it is running. RMS-11 enables ASTs when it must wait and when it returns control to your program. While your program is in control during RMS-11 I/O operations, the program can disable ASTs for critical operations. However, the outstanding RMS-11 operation typically will not be able to complete until ASTs are enabled again.

The RMS-11 Asynchronous Environment restricts operations as follows:

- You can initiate asynchronous RMS-11 operations at AST level provided the associated Record Access Stream is not in use. However, operations begun at AST level cannot complete until the task reverts to program level. Therefore, any associated wait operation ($WAIT macro) must **not** be performed at AST level.
- Any completion routine associated with an asynchronous record operation is typically executed at AST level and is considered part of the record operation:
  - The completion routine cannot initiate a synchronous RMS-11 operation. RMS-11 returns the error code ER$AST.
  - The completion routine cannot use the $WAIT macro. RMS-11 aborts the task.
  - Asynchronous RMS-11 operations initiated by the completion routine cannot complete until control is returned to RMS-11. You do this by issuing the $RETURN macro as the end of the completion routine. RMS-11 resumes control and performs an exit from AST level. Control returns to your program, although an AST is still outstanding because of the asynchronous operation initiated by the completion routine.

- Be careful initiating synchronous RMS-11 operations from the AST level while outside a completion routine. You must ensure that no resource required by the operation is exclusively controlled by an operation initiated at program level. Since the operation performs at AST level, it cannot be interrupted: the operation can loop continuously, waiting for that resource to be freed.

  Example Your program initiates an asynchronous record operation on an Indexed file. When control returns to the program, it initiates a synchronous operation on that same Indexed file. If the asynchronous operation locks the file Prologue or a bucket required by the synchronous operation, the synchronous operation loops, repeating its request for the locked blocks. Since the asynchronous operation cannot resume to unlock the blocks, the synchronous operation will never break out of its loop.

A.4 VAX/AME

A.4.1 RMS-11 Restrictions on VAX/AME

RMS-11 utilities do not support global wild card characters in file specifications; they do not follow subdirectories in all directories possibly covered by the wild card indication.

However, RMS-11 utilities do support wild card characters within named directories as well as within Master File Directories to the User File Directory level, but not to the subdirectory level.

A.4.2 VAX/AME Restrictions on RMS-11

The following RMS-11 capabilities are not available on VAX/AME:

- Write sharing. As long as programs open RMS-11 files with access no-write and allow no-write declarations, they can share the files. However, if the first program declares write access, other programs are denied access to the file; and if a reading program opens a file, no writing program is allowed to open it.
- RMS-11 Resident Library

A.4.3 Asynchronous Operations

VAX/AME allows tasks to perform operations asynchronously. You use this technique so that your task can perform functions during I/O operations, rather than wait until the I/O activity is over.

A.4.3.1 RMS-11 Synchronous Environment — By default, the Task Builder incorporates the synchronous RMS-11 routines into tasks. These routines do **not** enable or disable Asynchronous System Traps (ASTs). You can program routines that run at AST level; however, those routines must not:

- initiate an RMS-11 operation while another is still in progress. RMS-11 synchronous routines are vulnerable to this type of procedure: the results of

the interrupted operation are unpredictable. Therefore, you should avoid using RMS-11 routines at program level and AST level at the same time. To enable simultaneous operations at the different levels, you must take special precautions to avoid interference between the operations.

- cause the task segment containing the RMS-11 routines to be overlaid while an RMS-11 operation is in progress. The results in such a situation are unpredictable.

A.4.3.2 RMS-11 Asynchronous Environment — You can task build with the RMS-11 Asynchronous Executive modules (see *Nonoverlaid RMS-11* in "Task Building with RMS-11 Routines," Section 8.1 for details), thereby creating the RMS-11 Asynchronous Environment. File operations must still be performed synchronously. In order to perform asynchronous record operations, you must also set up an asynchronous Record Access Block (RAB) and turn on the RB$ASY in the RAB ROP field. See the *RMS-11 MACRO-11 Reference Manual* for details.

When your task initiates an RMS-11 asynchronous record operation, RMS-11 disables Asynchronous System Traps (ASTs) while it is running. RMS-11 enables ASTs when it must wait and when it returns control to your program. While your program is in control during RMS-11 I/O operations, the program can disable ASTs for critical operations. However, the outstanding RMS-11 operation typically will not be able to complete until ASTs are enabled again.

The RMS-11 Asynchronous Environment restricts operations as follows:

- You can initiate asynchronous RMS-11 operations at AST level provided the associated Record Access Stream is not in use. However, operations begun at AST level cannot complete until the task reverts to program level. Therefore, any associated wait operation ($WAIT macro) must **not** be performed at AST level.
- Any completion routine associated with an asynchronous record operation is typically executed at AST level and is considered part of the record operation:
  - The completion routine cannot initiate a synchronous RMS-11 operation. RMS-11 returns the error code ER$AST.
  - The completion routine cannot use the $WAIT macro. RMS-11 aborts the task.
  - Asynchronous RMS-11 operations initiated by the completion routine cannot complete until control is returned to RMS-11. You do this by issuing the $RETURN macro as the end of the completion routine. RMS-11 resumes control and performs an exit from AST level. Control returns to your program, although an AST is still outstanding because of the asynchronous operation initiated by the completion routine.

- Be careful initiating synchronous RMS-11 operations from the AST level while outside a completion routine. You must ensure that no resource required by the operation is exclusively controlled by an operation initiated at program level. Since the operation performs at AST level, it cannot be interrupted: the operation can loop continuously, waiting for that resource to be freed.

  Example Your program initiates an asynchronous record operation on an Indexed file. When control returns to the program, it initiates a synchronous operation on that same Indexed file. If the asynchronous operation locks the file Prologue or a bucket required by the synchronous operation, the synchronous operation loops, repeating its request for the locked blocks. Since the asynchronous operation cannot resume to unlock the blocks, the synchronous operation will never break out of its loop.

Appendix B
RMS–11 and the Programming Languages

B.1 Implementation in Languages

RMS–11 is the record management software for the following PDP–11 programming languages:

BASIC–PLUS–2
DIBOL
MACRO–11
PDP–11 COBOL
RPG II

All features of RMS–11 supported by an operating system (generally described in Table B–1) are available to the MACRO–11 programmer.

The other, higher level languages support a subset of RMS–11 features. The specific implementation of these features and the syntax used to invoke them are described in the language documentation.

B.2 Function Chart

Table B–1 identifies PDP–11 programming languages and the RMS–11 features they support. *It is not a definitive source for language capabilities.* See your latest language documentation for its current RMS–11 implementation.

The explicit information shown indicates features available through the language without the help of RMS–11 utilities or other software. However, the flag # indicates that a user can specify the feature via an RMS–11 utility (primarily RMSDEF) and then use it in the language.

Table B-1: RMS-11 Features Supported by Programming Languages

The notation S/R/I indicates that the feature is supported by the language for all file organizations, Sequential (S), Relative (R), and Indexed (I). Blanks are substituted for a letter when the language does not support the feature for an organization.

Features	MACRO-11	BASIC PLUS-2	COBOL	RPG II	DIBOL
File Organizations:					
SEQUENTIAL	YES	YES	YES	YES	YES
RELATIVE	YES	YES	YES	YES	YES
INDEXED	YES	YES	YES	YES	YES
File Operations:					
Create a file	$CREATE	OPEN	OPEN	Yes	OPEN[1]
Single area	Yes	Yes	Yes	Yes	Yes
Multiple areas	Yes	No#	No#	No#	No#
Initial allocation quantity	Yes	Yes	Yes		
Default extension quantity	Yes		Yes		
Contiguous allocation	Yes	Yes	Yes		
Fill number	Yes		Yes		
Open a file	$OPEN	OPEN	OPEN	Yes[2]	OPEN
By file specification	Yes	Yes	Yes	Yes	Yes
By file ID	Yes	No	No	No	No
Close a file	$CLOSE	CLOSE/END	CLOSE	Yes[3]	CLOSE
Delete a file	$ERASE	KILL		No	XCALL DELET
Examine an Indexed file's structure	$DISPLAY				
Implicit file extension	Yes	Yes	Yes	Yes	Yes
Explicit file extension	$EXTEND	No	No	No	No
Record Formats:					
FIXED	S/R/I	S/R/I	S/R/I	S/R/I	S/R/I
VARIABLE	S/R/I	S/R/I	S/R/I	S/R/I	S/R/I
VFC	S/R	No	No	No	S/R
STREAM	S	S	No	S	S[4]
UNDEFINED (BLOCK I/O)	S/R/I	S	No	No	No
Record Access Modes:					
SEQUENTIAL	S/R/I	S/R/I	S/R/I	Yes	S/R/I
RANDOM	R/I	R/I	R/I	R/I	R/I
RECORD FILE ADDRESS	S/R/I	S/R/I	No	No	No
DYNAMIC SHIFT BETWEEN MODES	Yes	Yes	Yes	Yes	Yes
Record Transfer Modes:					
MOVE	Yes	Yes	Yes	Yes	Yes
LOCATE	Yes	No	No	No	No

[1] A DIBOL program can create Sequential files only.

(continued on next page)

[2] Automatically before RPG II cycle execution.

[3] Automatically after RPG II cycle execution.

[4] Sequential files with stream format records are accessible for input only by a DIBOL task.

Table B-1: RMS-11 Features Supported by Programming Languages (Cont.)

Features	MACRO-11	BASIC PLUS-2	COBOL	RPG II	DIBOL
Record Operations:					
GET	$GET	GET	READ	Implicit	READ/READS
PUT	$PUT	PUT	WRITE	Implicit	WRITES STORE
Honor fill number	Yes	Yes	No	No	No
UPDATE	$UPDATE	UPDATE	REWRITE	Implicit	WRITE
Honor fill number	Yes	Yes	No	No	No
FIND	$FIND	FIND	START	CHAIN	No
DELETE	$DELETE	DELETE	DELETE	No	DELETE
Release locked bucket	$FREE	No	No	No	No
Set up Record Access Stream	$CONNECT	CONNECT	No	No	No
Terminate Record Access Stream	$DISCONNECT	Yes	No	No	No
Set pointer back to file beginning	$REWIND	RESTORE	No	No	No
Truncate Sequential disk file	$TRUNCATE	SCRATCH		No	No
Number of Record Access Streams per file:					
SINGLE	S/R/I	S/R/I	S/R/I	S/R/I	S/R/I
MULTIPLE	R/I	R/I	No	No	No
I/O Techniques:					
Synchronous I/O	Yes	Yes	Yes	Yes	Yes
Asynchronous I/O	Yes	No	No	No	No
Multiple Block Count (SEQ only)	Yes	Yes	Yes	No	Yes
Root Caching (IDX only)	Yes	Yes	Yes	No	No
Mass Insert	Yes	No	Yes	No	No
Retrieval Window Specification	Yes	Yes	Yes	No	No
Clustersize	Yes	Yes	Yes	No	No
Index Keys:					
MAXIMUM NUMBER	255	255	255	225[5]	255
SEGMENTED KEYS	Yes	No#	No	No	No
DATA TYPES					
String	Yes	Yes	Yes	Yes	Yes
15-bit Signed Integer	Yes	No	No	No	No
31-bit Signed Integer	Yes	No	No	No	No
16-bit Unsigned Binary	Yes	No	No	No	No
32-bit Unsigned Binary	Yes	No	No	No	No
Packed Decimal	Yes	No	No	No	No
DUPLICATES:					
Primary	Yes	Yes	No	No	Yes
Alternate	Yes	Yes	Yes	Yes	Yes
CHANGE VALUE DURING UPDATE:					
Primary	No	No	No	No	No
Alternate	Yes	Yes	Yes	Yes	Yes
NULL KEY VALUE	Yes	No#	No#	No#	No#
MATCH DURING RANDOM ACCESS:					
Exact	Yes	Yes	Yes	Yes	Yes
Approximate	Yes	Yes	Yes	Yes	Yes
Generic	Yes	Yes	No	Yes	Yes
Sharing:					
SEQUENTIAL	Multiple readers	Multiple readers	Multiple readers	Multiple readers	Multiple readers
RELATIVE	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write
INDEXED	Read/Write	Read/Write	Read/Write	Read/Write	Read/Write
I/O UNIT LOCKING	Automatic	Automatic	Automatic	Automatic	Automatic
I/O Buffer Handling:					
CENTRAL TO TASK	Yes	No	Yes	Yes	No
PRIVATE FOR EACH FILE	Yes	No	Yes	Yes	No
DYNAMIC ALLOCATION FROM ADDRESS SPACE	Yes	Yes	No	No	Yes
Magnetic Tape Operations:	Yes	Yes	Yes	Yes	Yes

[5] RPG II task default is buffers for four keys; however, each time a program is executed, it can use only one key.

B.3 ERROR CODE MAPPING

Each higher level language checks for RMS-11 completion codes after each file and record operation. If the code indicates anything other than unconditional success, the language Object- or Run-Time System *maps* the RMS-11 error code to a value compatible with the language's error reporting structure.

Table B-2 shows:

- the RMS-11 error codes available to the MACRO-11 programmer and the higher language OTS or RTS
- the equivalent higher level language error codes available to programmers using those languages
- a brief explanation of the cause of the error. The descriptions often include references to RMS-11 control structures familiar only to the MACRO-11 programmer.

When Table B-2 indicates that the Status Value (STV) field contains a file processor code, you should refer to the description of such codes in one of the following manuals:

- Error code appendix of the *IAS/RSX-11 I/O Operations Reference Manual*
- User recoverable error messages in an appendix of the *RSTS/E Programming Manual.* Note that the value returned in STV is the negative of the decimal value shown in the Programming Manual. That is, if STV contains "-20_{10}," look up "20_{10}."

When Table B-2 indicates that the Status Value field contains an FSS directive error code, you should refer to the error code appendix of the *RSTS/E System Directives Manual* for a description of the codes.

Table B-2: Language Error Code Mapping

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$ABO	177760_8/-16	ER$STK/ER$MAP	255[1]	30	-16	67	Operation aborted: Stack save area exhausted or memory-resident control structures corrupted.
ER$ACC	177740_8/-32	File processor error code	162	30	-32	67	File processor error: File processor could not access the file.
ER$ACT	177720_8/-48		255[1]	30	-48	67	File activity precludes operation. *Example* You attempted to close a file before an asynchronous record operation finished.
ER$AID	177700_8/-64	XAB address	255[1]	30	-64	67	The area indicated by the XAB does not exist in the file.
ER$ALN	177660_8/-80	XAB address	255[1]	30	-80	67	Illegal alignment value for Placement Control.
ER$ALQ	177640_8/-96	XAB address	255[1]	30	-96	67	Illegal allocation quantity: The quantity exceeds 65K blocks during file creation on a non-Large Files RSTS/E system or equals zero during an explicit file extension operation.
ER$ANI	177620_8/-112		128	30			
ER$AOP	177600_8/-128	XAB address	255[1]	30	-112	67	Records in a file on ANSI-labeled magnetic tape are variable-length, but not in ANSI-D format.
ER$AST	177560_8/-144		255[1]	30	-128	67	Invalid type of allocation.
					-144	67	Invalid operation at AST level: You attempted to issue a synchronous operation from an asynchronous record operation completion routine.
ER$ATR	177540_8/-160	File processor error code	252	30			
					-160	67	File processor error: Read failure on file attributes.
	177530_8/-168						Invalid File ID. See ER$FID.
ER$ATW	177520_8/-176	File processor error code	252	30	-176	67	File processor error: Write failure on file attributes.
ER$BKS	177500_8/-192		255[1]	30	-192	67	File bucket size exceeds maximum for operating system.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$BKZ	177460_8/-208	XAB address	255[1]	30	-208	67	Area bucket size exceeds maximum for operating system.
ER$BLN	177440_8/-224		255[1]	30	-224	67	Control block (FAB, RAB, or XAB) length is invalid.
ER$BOF	177430_8/-232		129	30	-232	67	Beginning of file detected during magnetic tape spacing operation.
ER$BPA	177420_8/-240		255[1]	30	-240	67	Invalid I/O buffer: Private buffer pool not on word boundary.
ER$BPS	177400_8/-256		255[1]	30	-256	67	Invalid I/O buffer: Private buffer pool size not a multiple of two bytes.
ER$BUG	177360_8/-272		255[1]	30	-272	67	RMS-11 aborts your task because it detected an internal error. Contact a DIGITAL Software Specialist.
ER$CCR	177340_8/-288		255[1]	30	-288	67	You attempted to connect more than one record access stream to a Sequential file.
ER$CHG	177320_8/-304		130	21	-304	100	During an update operation, you attempted to change a key field that does not allow changes.
ER$CHK	177300_8/-320		29	30	-320	67	Indexed file bucket corrupted. Do as many of the following steps as are necessary: 1. Move the disk pack containing the file to another device and try the process again. If it works, the error was caused by a hardware read failure. 2. Recreate the file using the RMSIFL or RMSCNV utility. If this works, the corrupted bucket was in an index bucket not used during sequential access by Primary Key. 3. Restore the file from your last backup copy.
ER$CLS	177260_8/-336	File processor error code	16	98	-336	32	File processor error: During RMS-11 file close operation.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$COD	177240_8/-352	XAB address	255[1]	30	-352	67	XAB type is invalid for the organization or operation.
ER$CRE	177220_8/-368	File processor error code	162/16	30	-368	67	File processor error: File processor could not create file.
ER$CUR	177200_8/-384		131	30	-384	113	No Current Record: Delete, truncate, or update operation was not immediately preceded by a successful get or find.
ER$DAC	177160_8/-400	File processor error code	252	98	-400	67	File processor error: File processor deaccess failure during RMS-11 file close operation.
ER$DAN	177140_8/-416	XAB address	255[1]	30	-416	67	Invalid area number specified in Key XAB DAN field.
ER$DEL	177120_8/-432		132	30	-432	67	Record accessed by RFA access mode has been deleted.
ER$DEV	177100_8/-448		133	30	-448	25	• Syntax error in device name • No such device • Inappropriate device for operation *Example* You attempted to create an Indexed file on magnetic tape.
ER$DFW	177070_8/-456	File processor error code	255[1]	30	-456	67	File processor error: File processor could not write bucket; RMS-11 deferred the I/O operation until it needed the I/O buffer for another bucket because the user program specified Deferred Write.
ER$DIR	177060_8/-464		1	30	-464	67	Syntax error in filespec directory name.
ER$DME	177040_8/-480		255[1]	30	-480	67	Dynamic memory exhausted: An RMS-11 buffer pool had insufficient free space.
ER$DNF	177020_8/-496		1	30	-496	26	Directory not found.
ER$DNR	177000_8/-512		14	30	-512	374	Device not ready.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$DPE	176770$_8$/-520	File processor error code	255[1]	30	-520	67	Device positioning error.
ER$DTP	176760$_8$/-528	XAB address	255[1]	30	-528	67	Invalid key data type.
ER$DUP	176740$_8$/-544		134	2/22[2]	-544	102	Invalid record operation: You attempted to insert a record that would cause duplicate values in a key field where duplicates are not allowed.
ER$ENT	176720$_8$/-560	File processor error code	162	30	-560	67	File processor error: File processor could not enter filespec in directory.
ER$ENV	176700$_8$/-576		135	30	-576	425	You attempted an operation when the RMS-11 routines were not in the task: In MACRO-11, the operation or file organization was not specified in ORG$ macro; in BASIC-PLUS-2, you didn't specify correct switches with BUILD command.
ER$EOF	176660$_8$/-592		11	10	-592	1	• For record processing: End of file. • For Block I/O: Invalid VBN.
ER$ESS	176640$_8$/-608		255[1]	30	-608	67	Expanded file-name string area in NAM block too short.
ER$EXP	176630$_8$/-616		255[1]	30	-616	67	File expiration date not reached.
ER$EXT	176620$_8$/-624	File processor error code	255[1]	30	-624	67	File processor error: During RMS-11 file extension operation.
ER$FAB	176600$_8$/-640		255[1]	30	-640	67	FAB BID field does not contain FB$BID.
ER$FAC	176560$_8$/-656		136	30	-656	67	Invalid record operation: Operation does not match access declaration made when file was created or opened.
ER$FEX	176540$_8$/-672		16	30	-672	67	You tried to create a file that exists.
ER$FID	177530$_8$/-168		255[1]	30	-168	67	Invalid file ID.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

[2] "2" means operation succeeded.
"22" means operation failed.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG	DIBOL	Description
ER$FLG	176520_8/-688	XAB address	137	30	-688	67	Invalid combination of key characteristics. *Example* No duplicate key values, but key values can change during update operations.
ER$FLK	176500_8/-704		138	91	-704	62	File locked by another user: You cannot access the file because your sharing specifications cannot be met.
ER$FND	176460_8/-720	File processor error code	162	30	-720	67	File processor error: File processor could not find filespec in specified directory.
ER$FNF	176440_8/-736		5	97	-736	26	File not found during file open operation.
ER$FNM	176420_8/-752		2	30	-752	25	Syntax error in file-name.
ER$FOP	176400_8/-768		139	30	-768	67	Invalid file access option specified in FAB FOP field.
ER$FSS	176370_8/-776	FSS directive error code	255[1]	30	-776	67	RSTS/E monitor error: the File-name String Scan routine is unable to parse the file-name string supplied by RMS-11.
ER$FUL	176360_8/-784		4	24/34 /95[3]	-784	36	Device full: RMS-11 cannot create or extend file.
ER$IAN	176340_8/-800	XAB address	255[1]	30	-800	67	Invalid area number specified in Key XAB IAN field.
ER$IDX	176320_8/-816		140	30	-816	67	Specified index was not created. This code can only occur in the STV field when STS contains ER$RNF.
ER$IFI	176300_8/-832		255[1]	30	-832	67	FAB IFI field contains invalid value.
ER$IMX	176260_8/-848	XAB address	255[1]	30	-848	67	More than 254 keys and/or areas defined or multiple Summary, Protection, or Date/Time XABs present during operation.
ER$INI	176240_8/-864		255[1]	30	-864	67	$INIT or $INITIF macro call never issued.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

[3] "24" means boundary violation during WRITE operation.
"34" means file couldn't be extended during a sequential WRITE operation.

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$IOP	176220_8/-880		141	30	-880	67	Illegal operation. *Example* You attempted to truncate a nonSequential file. *Example* You attempted to delete or extend a magnetic tape file. *Example* You issued a Block I/O operation to a stream not connected for block operations. *Example* You issued a record operation to a stream connected for Block I/O operations.
ER$IRC	176200_8/-896		142	30	-896	67	Illegal record encountered in Sequential file: Record-length field is invalid.
ER$ISI	176160_8/-912		255[1]	30	-912	67	RAB ISI field is invalid. You may have altered it or failed to connect the stream.
ER$KBF	176140_8/-928		143	30	-928	67	No key specified during random record operation: RAB KBF field equals 0.
ER$KEY	176120_8/-944		143	23/24[4]	-944	42	Negative Relative Record Number during random operation or bad format in packed decimal key value.
ER$KRF	176100_8/-960		144	30	-960	67	Invalid key of reference: • During random get or find operation. • During connect or rewind operation: Error code is returned for the first sequential get or find operation following the connect or rewind.
ER$KSZ	176060_8/-976		145	30	-976	67	Invalid key size
ER$LAN	176040_8/-992	XAB address	255[1]	30	-992	67	Invalid area number specified in Key XAB LAN field.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.
[4] "23" means record was not found during random READ of a Relative file.
"24" means boundary violation during WRITE operation.

(continued on next page)

Table B–2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$LBL	176020_8/-1008		146	30	-1008	67	Invalid medium: Magnetic tape is not labeled in accordance with ANSI standards.
ER$LBY	176000_8/-1024		7	30	-1024	67	Logical channel busy: You attempted to create or open a file using a logical channel in use; that is, you already opened a file on that channel.
ER$LCH	175760_8/-1040		46	30	-1040	67	Invalid logical channel or unit number.
ER$LEX	175750_8/-1048	XAB address	255[1]	30	-1048	67	You attempted to extend an area containing an unused extent.
ER$LOC	175740_8/-1056	XAB address	255[1]	30	-1056	67	Invalid location during Placement Control.
ER$MAP	175720_8/-1072		255[1]	30	-1072	67	Memory-resident data structures, such as I/O buffers, corrupted. This code can occur in the STV field when Status Code contains ER$ABO.
ER$MKD	175700_8/-1088	File processor error code	252	30	-1088	67	File processor error: File processor could not mark file for deletion.
ER$MRN	175660_8/-1104		147	30	-1104	67	• Maximum Record Number field contains a negative value during creation of Relative file. • Relative Record Number for random operation to Relative file exceeds Maximum Record Number specified when file was created.
ER$MRS	175640_8/-1120		148	30	-1120	67	Maximum Record Size is zero during file creation and one of the following is true: • Record format is fixed • File organization is Relative.
ER$NAM	175620_8/-1136		255[1]	30	-1136	67	Odd address in FAB NAM field on file open, creation, or erase operation.
ER$NEF	175600_8/-1152		149	30	-1152	67	You attempted a put operation to a Sequential file when stream is not positioned to end-of-file.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$NID	175560_8/-1168		255[1]	30	-1168	67	Dynamic memory exhausted: Not enough buffer area to open an Indexed file.
ER$NPK	175540_8/-1184		150	30	-1184	67	You attempted to create an Indexed file without defining a Primary Key.
ER$OPN	175520_8/-1200	FIB error code	21	30	-1200	67	File processor error: During RMS-11 file open operation.
ER$ORD	175500_8/-1216	XAB address	255[1]	30	-1216	67	XABs not ordered properly.
ER$ORG	175460_8/-1232		255[1]	30	-1232	67	Invalid file organization.
ER$PLG	175440_8/-1248		29	30	-1248	67	A Prologue block may be corrupted. Do as many of the following steps as are necessary: 1. Move the disk pack containing the file to another device and try the process again. If it works, the error was caused by a hardware read failure. 2. Recreate the file using the RMSIFL or RMSCNV utility. 3. Restore the file from your last backup copy.
ER$POS	175420_8/-1264	XAB address	151	30	-1264	67	You specified a key position beyond the end of the record.
ER$PRM	175400_8/-1280	XAB address	255[1]	30	-1280	67	File directory entry contains date and time information not semantically correct. The file may be corrupted. Recreate field using RMSIFL or RMSCNV utility.
ER$PRV	175360_8/-1296		10	30	-1296	114	Privilege violation: access to the file denied by the operating system.
ER$RAB	175340_8/-1312		255[1]	30	-1312	67	RAB BID field does not contain RB$BID.
ER$RAC	175320_8/-1328		152	30	-1328	67	Invalid or illogical record access option specified in RAB RAC field.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$RAT	175300$_8$/-1344		255[1]	30	-1344	67	You specified both Carriage Return control and FORTRAN forms control.
ER$RBF	175260$_8$/-1360		255[1]	30	-1360	67	RAB RBF field contains an odd address (Block I/O access only).
ER$RER	175240$_8$/-1376	File processor error code	252	30	-1376	32	File processor error: • In record processing: Read failure on file block. • In Block I/O, VBN = 0, an illegal value.
ER$REX	175220$_8$/-1392		153	22	-1392	113	Record exists: During a put operation in random mode to a Relative file, you tried to insert a record into a cell containing a record.
ER$RFA	175200$_8$/-1408		173	30	-1408	67	Invalid RFA during RFA access.
ER$RFM	175160$_8$/-1424		167	30	-1424	67	Invalid record format.
ER$RLK	175140$_8$/-1440		154	92	-1440	62	Target bucket locked by another task or another stream in the same program.
ER$RMV	175120$_8$/-1456	File processor error code	252	30	-1456	67	File processor error: File processor could not remove filespec from directory.
ER$RNF	175100$_8$/-1472	ER$IDX	155	23	-1472	144	Record specified during random get or find operation does not exist in Relative or Indexed file. For Indexed files only, STV may contain ER$IDX.
ER$RNL	175060$_8$/-1488		255[1]	30	-1488	NO ERROR	You initiated a free operation, but no bucket was locked.
ER$ROP	175040$_8$/-1504		255[1]	30	-1504	67	Invalid record processing option or illogical combination of values specified in RAB ROP field.
ER$RPL	175020$_8$/-1520	File processor error code	255[1]	30	-1520	67	File processor error: Read failure on file Prologue.
ER$RRV	175000$_8$/-1536		29	30	-1536	67	Invalid RRV record encountered in Indexed file. File may be corrupted. Recreate field using RMSIFL or RMSCNV utility.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$RSA	174760_8/-1552		255[1]	30	-1552	67	Record stream active: In asynchronous environment, you attempted a record operation on a stream that is performing an operation.
ER$RSZ	174740_8/-1568		156	30	-1568	33	• Record size exceeds one of the following: - MRS for variable-length or VFC records. - magnetic tape block size. - data bucket size of Indexed file. - 51010 bytes and block spanning not allowed. - end of I/O buffer during Locate Mode put operation on sequential file. • Record is too short to contain Primary Key of Indexed file. • Record size is zero during Block I/O. • Record size is not equal to size of Current Record for update operation on a disk Sequential file or on an Indexed file where Primary Key duplicates are allowed. • Record size does not equal MRS for fixed-length records.
ER$RTB	174720_8/-1584	Actual record size	157	30	-1584	33	Record too big for user buffer: RMS-11 could not move entire record retrieved by get operation to user buffer. This error does not destroy the context of the stream. Rather, the stream's context is updated as if the operation were completely successful and as much of the record as possible is moved to the user buffer.
ER$RVU	174710_8/-1592		255[1]	30	-1592	67	During a put or update operation, RMS-11 moved the specified record to the file successfully, but could not update one or more Record Reference Vector (RRV). **The file is corrupted,** but you can retrieve every record via the Primary index. Therefore, you should create a new Indexed file and populate it from the bad file using either the RMSIFL or the RMSCNV utility.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$SEQ	174700_8/-1600		158	21	-1600	67	Invalid record operation: During a sequential put operation to an Indexed file, Primary Key of record to be written is not equal to or greater than key of previous record.
ER$SHR	174660_8/-1616		168	30	-1616	56	You specified allow write declaration for Sequential file.
ER$SIZ	174640_8/-1632	XAB address	159	30	-1632	67	Invalid key size. *Example* Key is bigger than Maximum Record Size.
ER$STK	174620_8/-1648		255[1]	30	-1648	67	During asynchronous record operation, RMS-11 found the stack is too big to save. This code can only occur in the STV field when Status Code contains ER$ABO.
ER$SYS	174600_8/-1664	Directive or QIO status error code	255[1]	30	-1664	67	The interface between RMS-11 and the operating system has changed. Report this error on an SPR to DIGITAL.
ER$TRE	174560_8/-1680		29	30	-1680	67	Index in Indexed file is corrupted. Recreate file using RMSIFL or RMSCNV utility.
ER$TYP	174540_8/-1696		2	30	-1696	67	Syntax error in extension, such as more than three characters specified.
ER$UBF	174520_8/-1712		255[1]	30	-1712	67	User buffer improperly specified: address is either zero or buffer is not word-aligned (for Block I/O access only).
ER$USZ	174500_8/-1728		255[1]	30	-1728	67	User buffer length equals zero.
ER$VER	174460_8/-1744		2	30	-1744	67	Syntax error in file version number.
ER$VOL	174440_8/-1760	XAB address	255[1]	30	-1760	67	Nonzero relative volume number.
ER$WCD	174430_8/-1768		255[1]	30	-1768	67	Explicit or default file specification field contains wild card character.
ER$WER	174420_8/-1776	File processor error code	252	30	-1776	67	File processor error: Write failure on file block.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

(continued on next page)

Table B-2: Language Error Code Mapping (Cont.)

Symbolic Value	MACRO-11 Status Code	MACRO-11 Status Value	BASIC-PLUS-2	PDP-11 COBOL	RPG II	DIBOL	Description
ER$WLK	174410_8/-1784		14	30	-1784	67	File processor error: Device is write locked.
ER$WPL	174400_8/-1792	File processor error code	252	30	-1792	67	File processor error: Error while writing Prologue.
ER$XAB	174360_8/-1808	XAB address	255[1]	30	-1808	67	FAB XAB field or XAB NXT field contains an odd address.
ER$XTR	174340_8/-1824		255[1]	30	-1824	67	Explicit or default file specification contains extraneous field.
SU$DUP	2/2		172	NO ERROR	2	67	Conditional success: A record inserted into an Indexed file by a put or update operation contains at least one key value present in another record.
SU$IDX	3/3		169	NO ERROR	3	67	A put or update operation on an Indexed file ended successfully, but RMS-11 could not optimize the index structure during the operation. Therefore, RMS-11 will require extra I/O operations to retrieve the record. With this success code, RMS-11 may include an RMS-11 error code to indicate why the index structure was not updated in the STV field of the control block.
SU$RRV	4/4		169	NO ERROR	4	67	No longer a valid completion code. See ER$RVU.

[1] On IAS/RSX-11M, value of ERR is same as MACRO-11 Status Code.

Appendix C
RMS–11 Disk–Resident Overlays

C.1 Overlay Structures

Figure C–1 is a graphic illustration of the overlay structures defined by the standard RMS–11 ODL files, RMS11S.ODL and RMS11X.ODL. From this figure, you can get a feeling for the extent and complexity of the RMS–11 overlay structures. You can also see what overlay segments are loaded into memory before the segment containing the called routine is loaded.

Example Your program initiates an update operation and causes a call to an RMS–11 routine in the overlay segment labeled with the factor name RMIO3U. Before the operating system can execute this routine, it must ensure that the following segments are in memory:

```
RMSALL
RMSREC
RM0X36
```

The following conventions were used in creating Figure C–1:

- The parenthetical comment *map name* indicates a cosmetic term used on the Memory Allocation Map produced by the Task Builder.

Example The standard RMS-11 ODL file RMS11S.ODL contains the factor:

```
RMSFIL: .FCTR    RMSFAB-LB:[1,1]RMSLIB/LB:R0IFLF:R0FSET:R0FSEI-RMSFL
```

RMS11S.ODL also contains the ODL statement:

```
.NAME    RMSFAB
```

Finding this combination of statements, the Task Builder describes the overlay segment defined in the RMSFIL factor with the name RMSFAB. *FAB* is the acronym for *File Access Block*, the internal control structure RMS-11 and the user program use to describe a file. Since the RMSFIL factor starts the series of overlay segments that contain file operation routines, the name *RMSFAB* is appropriate.

If RMSFAB were left out of the factor, the Task Builder would use the name of the first module or factor in the line; in this case, the utility would use *R0IFLF*, a less meaningful name.

- Factor names are in bold print.
- Each indentation of one space represents a concatenated factor or module within a factor. Each separate series of indented names represents an overlay segment.
- Factors or modules equally indented either overlay each other or the top one is only a factor or map name.

Example The following portion of the RMS11S.ODL chart shows:

- The factor RM0X0X concatenated with the module RMSSYM.
- The fact that RM0X0X is only the name for a factor that starts with another factor name (RMSIO). The two factor names are equally indented.

  Factor names such as RM0X0X are used because the Task Builder limits the length of input lines. Such names essentially extend factors.
- The factor RMSCBL is concatenated with the factor RMSIO. The intervening module names comprise RMSIO.

RMSSYM
RM0X0X
RMSIO
R0CACH
R0RLCH
R0MAPC
R0RWBF
R0UNLK
RMSCBL

Figure C-1: RMS11S.ODL and RMS11X.ODL Overlay Structures

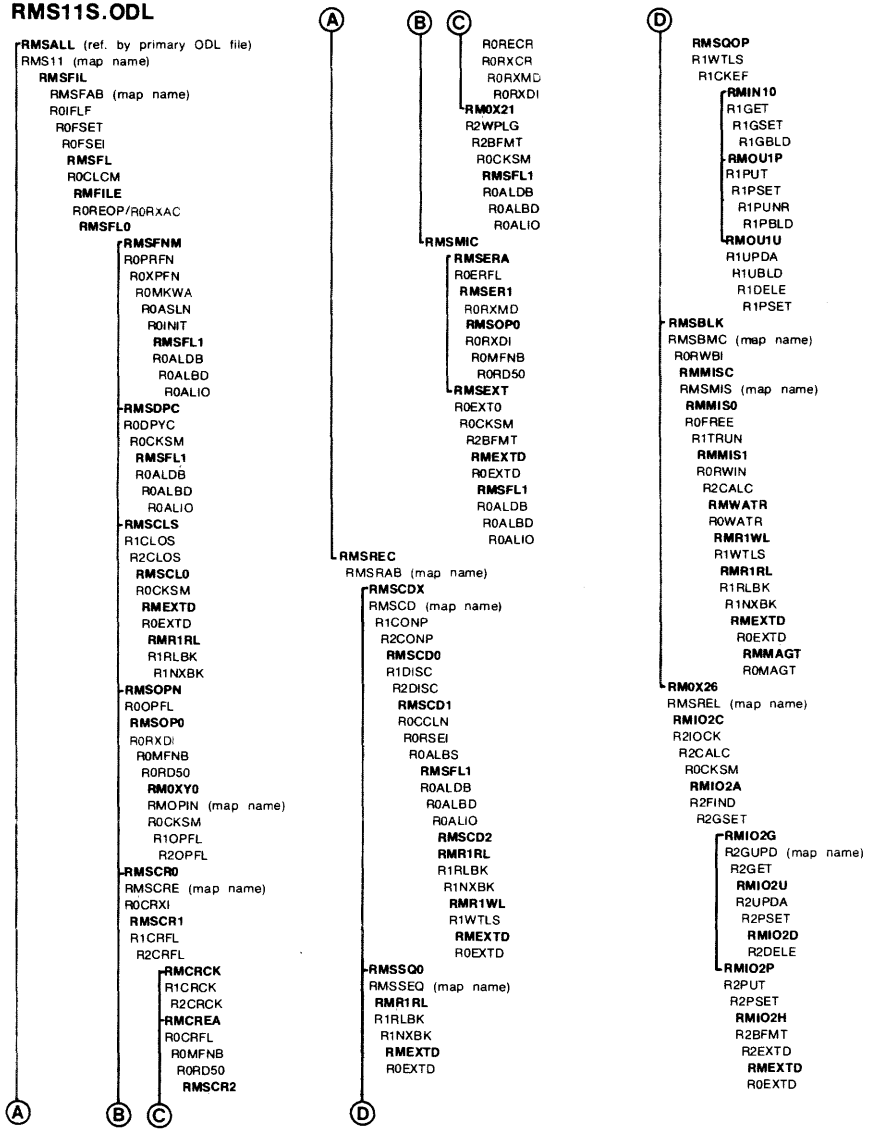
RMS11S.ODL

RMSROT (ref. by primary ODL file)
RMSSYM
RM0X0X
RMSIO
R0CACH
R0RLCH
R0MAPC
R0RWBF
R0UNLK
RMSCBL
R0RTCB
R0ACBB
R0RTDB
R0MDAT
RMSCOM
R0RMSE
R0SAVR
R0IDPB
RMSCM1
R0AUTO
R0IMPA
RMSEXC
R0EXSY
R0RSES
R0WTBS
R0RMSA

RMS11X.ODL

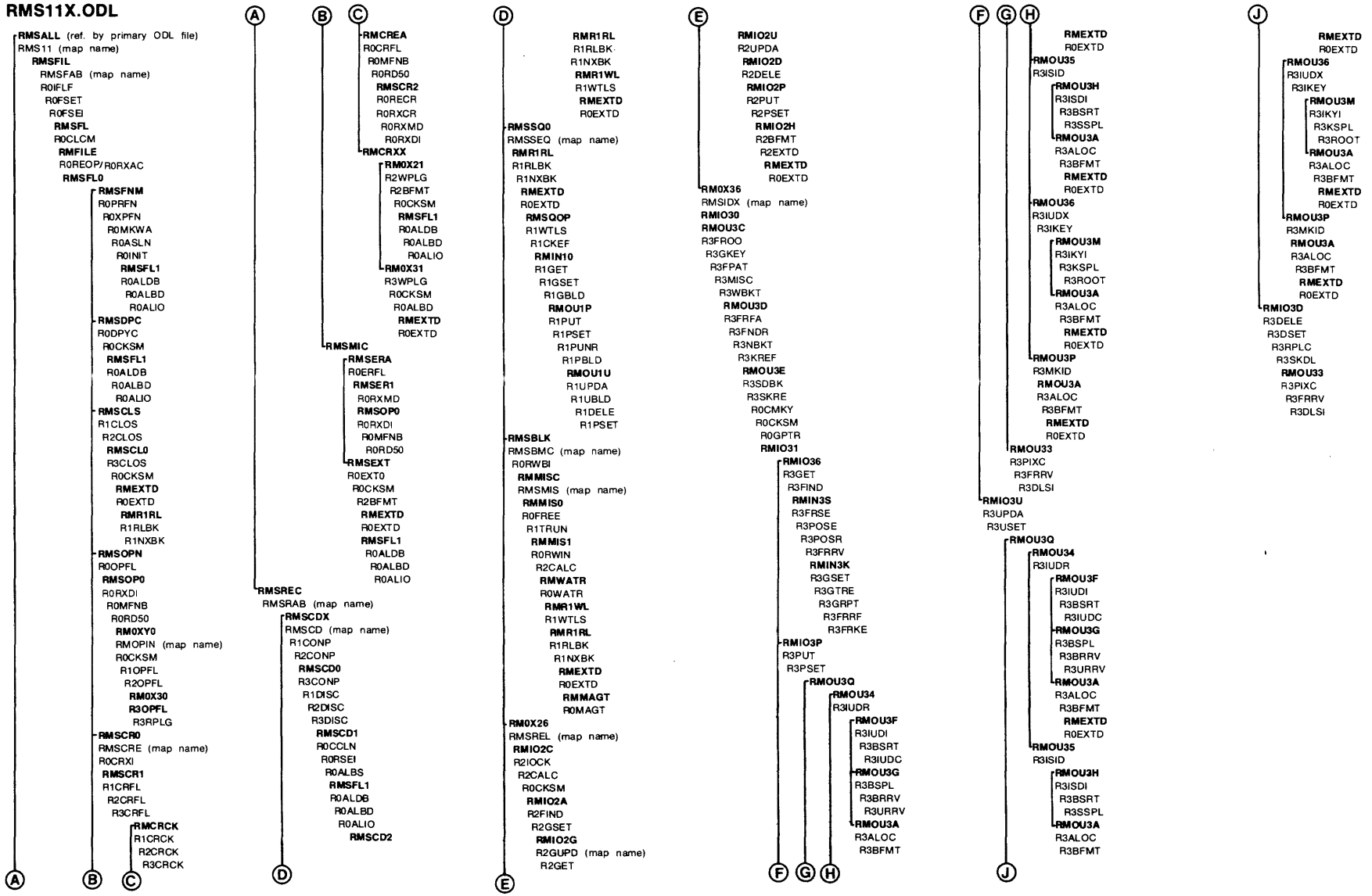
RMSROT (ref. by primary ODL file)
RMSSYM
RM0X3X
RMSIO
R0CACH
R0RLCH
R0MAPC
R0RWBF
R0UNLK
RMSCBL
R0RTCB
R0ACBB
R0RTDB
R0MDAT
RMSCOM
R0RMSE
R0SAVR
R0IDPB
RMSCM1
R0AUTO
R0IMPA
RMSEXC
R0EXSY
R0RSES
R0WTBS
R0RMSA
RMSIX0
R0RLSB
R0GETB

Figure C-1: RMS11S.ODL and RMS11X.ODL Overlay Structures (Cont).



(continued on next page)

Figure C–1: RMS11S.ODL and RMS11X.ODL Overlay Structure (Cont.)



C.2 RMS–11 ODL FILES

C.2.1 RMS16X.ODL

```
RMSCM1:  .FCTR   LB:[1,1]RMSLIB/LB:R0AUTO:R0IMPA
RMSROT:  .FCTR   LB:[1,1]RMSLIB/LB:RMSSYM-RMXXXX
RMSIX0:  .FCTR   LB:[1,1]RMSLIB/LB:R0RLSB:R0GETB
         .NAME   RMS11
         .NAME   RMSFAB
         .NAME   RMSRAB
         .NAME   RMSCRE
         .NAME   RMSCD
         .NAME   RMOPIN
         .NAME   RMSSEQ
         .NAME   RMSREL
         .NAME   R2GUPD
         .NAME   RMSIDX
         .NAME   RMSBMC
         .NAME   RMSMIS
         .NAME   RMDEXT
         .NAME   RMDRLB
         .NAME   RMDWTL
RMSFL:   .FCTR   LB:[1,1]RMSLIB/LB:R0CLCM-RMFILE-RMSFL0
RMSFNM:  .FCTR   LB:[1,1]RMSLIB/LB:R0INIT:R0PRFN:R0XPFN:R0MKWA:R0ASLN-RMSFL1
RMSDPC:  .FCTR   LB:[1,1]RMSLIB/LB:R0DPYC
RMSFL1:  .FCTR   LB:[1,1]RMSLIB/LB:R0ALDB:R0ALBD:R0ALIO
RMSCLS:  .FCTR   LB:[1,1]RMSLIB/LB:R1CLOS:R2CLOS-RMSCL0
RMSCL0:  .FCTR   LB:[1,1]RMSLIB/LB:R3CLOS:R0CKSM-RMEXTD-RMR1RL
RMSOPN:  .FCTR   LB:[1,1]RMSLIB/LB:R0OPFL-RMSOP0-RM0XY0
RMCRCK:  .FCTR   LB:[1,1]RMSLIB/LB:R1CRCK:R2CRCK:R3CRCK
RMCREA:  .FCTR   LB:[1,1]RMSLIB/LB:R0CRFL:R0MFNB:R0RD50-RMSCR2
RMSCR0:  .FCTR   RMSCRE-LB:[1,1]RMSLIB/LB:R0CRXI-RMSCR1
RMSALL:  .FCTR   RMS11-*(RMSFIL,RMSREC)
RMSFL0:  .FCTR   RMSFNM-(RMSOPN-RMSDPC-RMSCDX,RMSCR0,RMSCLS-RMSMIC-RMSCD0)
RMSMIC:  .FCTR   RMSERA-RMSEXT
RMSERA:  .FCTR   LB:[1,1]RMSLIB/LB:R0ERFL-RMSER1
RMSEXT:  .FCTR   LB:[1,1]RMSLIB/LB:R0EXT0:R2BFMT
RM0XY0:  .FCTR   RMOPIN-LB:[1,1]RMSLIB/LB:R0CKSM:R1OPFL:R2OPFL-RM0X30
RM0X30:  .FCTR   LB:[1,1]RMSLIB/LB:R3OPFL:R3RPLG
RMSCR1:  .FCTR   LB:[1,1]RMSLIB/LB:R1CRFL:R2CRFL:R3CRFL-RMCRCK-RMCREA-RMCRXX
RMCRXX:  .FCTR   RM0X21-RM0X31
RM0X21:  .FCTR   LB:[1,1]RMSLIB/LB:R2WPLG:R2BFMT:R0CKSM
RM0X31:  .FCTR   LB:[1,1]RMSLIB/LB:R3WPLG-RMEXTD
RMSCDX:  .FCTR   LB:[1,1]RMSLIB/LB:R1CONP:R2CONP:R3CONP:R0RSEI-RMSCD1
RMSCD1:  .FCTR   LB:[1,1]RMSLIB/LB:R0CCLN:R0ALBS
RMSCD0:  .FCTR   LB:[1,1]RMSLIB/LB:R1DISC:R2DISC:R3DISC-RMSCD2
RMSCD2:  .FCTR   LB:[1,1]RMSLIB/LB:R0CCLN-RMR1WL
RMSREC:  .FCTR   RMSRAB-RMEXTD-(RM0X26-RMSBLK-RMSSQ0,RM0X36)
RM0X26:  .FCTR   RMSREL-RMI02C-RMI02A
RMI02C:  .FCTR   LB:[1,1]RMSLIB/LB:R2IOCK:R2CALC:R0CKSM


RMI02A:  .FCTR   LB:[1,1]RMSLIB/LB:R2FIND:R2GSET-RMI02G-RMI02P
RMI02G:  .FCTR   LB:[1,1]RMSLIB/LB:R2GET-RMI02U-RMI02D
RMI02P:  .FCTR   LB:[1,1]RMSLIB/LB:R2PUT:R2PSET-RMI02H
RMI02H:  .FCTR   LB:[1,1]RMSLIB/LB:R2BFMT:R2EXTD
RMI02U:  .FCTR   LB:[1,1]RMSLIB/LB:R2UPDA
RMI02D:  .FCTR   LB:[1,1]RMSLIB/LB:R2DELE
RM0X36:  .FCTR   RMSIDX-RMI030-RMI031
RMI030:  .FCTR   RMOU3C-RMOU3D-RMOU3E-RMI03P-RMI03U-RMOU3A
RMI031:  .FCTR   (RMI03G-RMI03D-RMOU33,RMOU3Q)
RMI03G:  .FCTR   LB:[1,1]RMSLIB/LB:R3GET:R3FIND-RMIN3S-RMIN3K
RMIN3K:  .FCTR   LB:[1,1]RMSLIB/LB:R3GSET:R3GTRE:R3GRPT:R3FRRF:R3FRKE
RMIN3S:  .FCTR   LB:[1,1]RMSLIB/LB:R3FRSE:R3POSE:R3POSR:R3FRRV
```

```
RMIO3P:  .FCTR   LB:[1,1]RMSLIB/LB:R3PUT:R3PSET
RMOU3Q:  .FCTR   (RMOU34,RMOU35-RMOU36-RMOU3P)
RMIO3U:  .FCTR   LB:[1,1]RMSLIB/LB:R3UPDA:R3USET
RMIO3D:  .FCTR   LB:[1,1]RMSLIB/LB:R3DELE:R3DSET:R3RPLC:R3SKDL
RMOU33:  .FCTR   LB:[1,1]RMSLIB/LB:R3PIXC:R3DLSI
RMOU34:  .FCTR   LB:[1,1]RMSLIB/LB:R3IUDR-RMOU3J-RMOU3F-RMOU3K-RMOU3G
RMOU35:  .FCTR   LB:[1,1]RMSLIB/LB:R3ISID-RMOU3H
RMOU36:  .FCTR   LB:[1,1]RMSLIB/LB:R3UIDX:R3IKEY-RMOU3M-RMOU3O
RMOU3A:  .FCTR   LB:[1,1]RMSLIB/LB:R3ALOC:R3BFMT
RMOU3C:  .FCTR   LB:[1,1]RMSLIB/LB:R3FROO:R3GKEY:R3FPAT:R3MISC:R3WBKT
RMOU3D:  .FCTR   LB:[1,1]RMSLIB/LB:R3FRFA:R3FNDR:R3NBKT:R3KREF
RMOU3E:  .FCTR   LB:[1,1]RMSLIB/LB:R3SDBK:R3SKRE:R0CMKY:R0CKSM:R0GPTR
RMOU3F:  .FCTR   LB:[1,1]RMSLIB/LB:R3IUDI:R3BSRT
RMOU3G:  .FCTR   LB:[1,1]RMSLIB/LB:R3BRRV:R3URRV
RMOU3H:  .FCTR   LB:[1,1]RMSLIB/LB:R3ISDI:R3BSRT-RMOU3L
RMOU3J:  .FCTR   LB:[1,1]RMSLIB/LB:R3IUDC
RMOU3K:  .FCTR   LB:[1,1]RMSLIB/LB:R3BSPL
RMOU3L:  .FCTR   LB:[1,1]RMSLIB/LB:R3SSPL
RMOU3M:  .FCTR   LB:[1,1]RMSLIB/LB:R3IKYI
RMOU3N:  .FCTR   LB:[1,1]RMSLIB/LB:R3KSPL
RMOU3O:  .FCTR   LB:[1,1]RMSLIB/LB:R3ROOT-RMOU3N
RMOU3P:  .FCTR   LB:[1,1]RMSLIB/LB:R3MKID
RMSBLK:  .FCTR   RMSBMC-LB:[1,1]RMSLIB/LB:R0RWBI-RMMISC
RMMISC:  .FCTR   RMSMIS-RMMIS0
RMMIS0:  .FCTR   LB:[1,1]RMSLIB/LB:R0FREE:R1TRUN-RMMIS1
RMMIS1:  .FCTR   LB:[1,1]RMSLIB/LB:R0RWIN-RMWATR-RMMAGT
RMWATR:  .FCTR   LB:[1,1]RMSLIB/LB:R0WATR
RMMAGT:  .FCTR   LB:[1,1]RMSLIB/LB:R0MAGT
```

C.2.2 RMS20X.ODL

```
RMSCM1:  .FCTR   LB:[1,1]RMSLIB/LB:R0AUTO:R0IMPA
RMSROT:  .FCTR   LB:[1,1]RMSLIB/LB:RMSSYM-RMXXXX
RMSIX0:  .FCTR   LB:[1,1]RMSLIB/LB:R0RLSB:R0GETB
         .NAME   RMS11
         .NAME   RMSFAB
         .NAME   RMSRAB
         .NAME   RMSCRE
         .NAME   RMSCD
         .NAME   RMOPIN
         .NAME   RMSSEQ
         .NAME   RMSREL
         .NAME   RMSIDX
         .NAME   RMSBMC
         .NAME   RMSMIS
         .NAME   RMDEXT
         .NAME   RMDRLB
         .NAME   RMDWTL
RMSFL:   .FCTR   LB:[1,1]RMSLIB/LB:R0CLCM-RMFILE-RMSFL0
RMSFNM:  .FCTR   LB:[1,1]RMSLIB/LB:R0INIT:R0PRFN:R0XPFN:R0MKWA:R0ASLN-RMSFL1
RMSDPC:  .FCTR   LB:[1,1]RMSLIB/LB:R0DPYC
RMSFL1:  .FCTR   LB:[1,1]RMSLIB/LB:R0ALDB:R0ALBD:R0ALIO
RMSCLS:  .FCTR   LB:[1,1]RMSLIB/LB:R1CLOS:R2CLOS-RMSCL0
RMSCL0:  .FCTR   LB:[1,1]RMSLIB/LB:R3CLOS:R0CKSM-RMEXTD-RMR1RL
RMSOPN:  .FCTR   LB:[1,1]RMSLIB/LB:R0OPFL-RMSOP0-RM0XY0
RMCRCK:  .FCTR   LB:[1,1]RMSLIB/LB:R1CRCK:R2CRCK:R3CRCK
RMCREA:  .FCTR   LB:[1,1]RMSLIB/LB:R0CRFL-RMSCR2
RMSCR0:  .FCTR   RMSCRE-LB:[1,1]RMSLIB/LB:R0CRXI-RMSCR1
RMSALL:  .FCTR   RMS11-*(RMSFIL,RMSREC)
RMSFL0:  .FCTR   RMSFNM-(RMSOPN-RMSDPC-RMSCDX-RMSCR0,RMSCLS-RMSMIC-RMSCD0)
```

```
RMSMIC:  .FCTR   RMSERA-RMSEXT
RMSERA:  .FCTR   LB:[1,1]RMSLIB/LB:R0ERFL-RMSER1
RMSEXT:  .FCTR   LB:[1,1]RMSLIB/LB:R0EXTO:R2BFMT
RM0XY0:  .FCTR   RMOPIN-LB:[1,1]RMSLIB/LB:R0CKSM:R1OPFL:R2OPFL-RM0X30
RM0X30:  .FCTR   LB:[1,1]RMSLIB/LB:R3OPFL:R3RPLG
RMSCR1:  .FCTR   LB:[1,1]RMSLIB/LB:R1CRFL:R2CRFL:R3CRFL-RMCRCK-RMCREA-RMCRXX
RMCRXX:  .FCTR   RM0X21-RM0X31
RM0X21:  .FCTR   LB:[1,1]RMSLIB/LB:R2WPLG:R2BFMT
RM0X31:  .FCTR   LB:[1,1]RMSLIB/LB:R3WPLG
RMSCDX:  .FCTR   LB:[1,1]RMSLIB/LB:R1CONP:R2CONP:R3CONP:R0RSEI-RMSCD1
RMSCD1:  .FCTR   LB:[1,1]RMSLIB/LB:R0CCLN:R0ALBS-RMEXTD
RMSCD0:  .FCTR   LB:[1,1]RMSLIB/LB:R1DISC:R2DISC:R3DISC-RMSCD2
RMSCD2:  .FCTR   LB:[1,1]RMSLIB/LB:R0CCLN-RMR1WL
RMSREC:  .FCTR   RMSRAB-RMEXTD-(RMSBLK-RM0X26,RMSSQ0-RM0X36)
RM0X26:  .FCTR   RMSREL-RMI02C-RMI02A
RMI02C:  .FCTR   LB:[1,1]RMSLIB/LB:R2IOCK:R2CALC:R0CKSM
RMI02A:  .FCTR   LB:[1,1]RMSLIB/LB:R2FIND:R2GSET-RMI02G-RMI02P
RMI02G:  .FCTR   LB:[1,1]RMSLIB/LB:R2GET-RMI02U-RMI02D
RMI02P:  .FCTR   LB:[1,1]RMSLIB/LB:R2PUT:R2PSET-RMI02H
RMI02H:  .FCTR   LB:[1,1]RMSLIB/LB:R2BFMT:R2EXTD
RMI02U:  .FCTR   LB:[1,1]RMSLIB/LB:R2UPDA
RMI02D:  .FCTR   LB:[1,1]RMSLIB/LB:R2DELE
RM0X36:  .FCTR   RMSIDX-RMI030-RMI03G-RMI03P-RMI03U-(RMOU3Q)
RMI030:  .FCTR   RMOU3C-RMOU3D-RMOU3E-RMOU3A
RMI03G:  .FCTR   LB:[1,1]RMSLIB/LB:R3GET:R3FIND-RMIN3S-RMIN3K
RMIN3K:  .FCTR   LB:[1,1]RMSLIB/LB:R3GSET:R3GTRE:R3GRPT:R3FRRF:R3FRKE
RMIN3S:  .FCTR   LB:[1,1]RMSLIB/LB:R3FRSE:R3POSE:R3POSR:R3FRRV
RMI03P:  .FCTR   LB:[1,1]RMSLIB/LB:R3PUT:R3PSET
RMOU3Q:  .FCTR   (RMOU33-RMI03D-RMOU3P,RMOU34,RMOU35-RMOU36)
RMI03U:  .FCTR   LB:[1,1]RMSLIB/LB:R3UPDA:R3USET
RMI03D:  .FCTR   LB:[1,1]RMSLIB/LB:R3DELE:R3DSET:R3RPLC:R3SKDL
RMOU33:  .FCTR   LB:[1,1]RMSLIB/LB:R3PIXC:R3DLSI
RMOU34:  .FCTR   LB:[1,1]RMSLIB/LB:R3IUDR-RMOU3J-RMOU3F-RMOU3K-RMOU3G
RMOU35:  .FCTR   LB:[1,1]RMSLIB/LB:R3ISID-RMOU3H
RMOU36:  .FCTR   LB:[1,1]RMSLIB/LB:R3UIDX:R3IKEY-RMOU3M-RMOU3O
RMOU3A:  .FCTR   LB:[1,1]RMSLIB/LB:R3ALOC:R3BFMT
RMOU3C:  .FCTR   LB:[1,1]RMSLIB/LB:R3FROO:R3GKEY:R3FPAT:R3MISC:R3WBKT
RMOU3D:  .FCTR   LB:[1,1]RMSLIB/LB:R3FRFA:R3FNDR:R3NBKT:R3KREF
RMOU3E:  .FCTR   LB:[1,1]RMSLIB/LB:R3SDBK:R3SKRE:R0CMKY:R0CKSM:R0GPTR
RMOU3F:  .FCTR   LB:[1,1]RMSLIB/LB:R3IUDI:R3BSRT
RMOU3G:  .FCTR   LB:[1,1]RMSLIB/LB:R3BRRV:R3URRV
RMOU3H:  .FCTR   LB:[1,1]RMSLIB/LB:R3ISDI:R3BSRT-RMOU3L
RMOU3J:  .FCTR   LB:[1,1]RMSLIB/LB:R3IUDC
RMOU3K:  .FCTR   LB:[1,1]RMSLIB/LB:R3BSPL
RMOU3L:  .FCTR   LB:[1,1]RMSLIB/LB:R3SSPL
RMOU3M:  .FCTR   LB:[1,1]RMSLIB/LB:R3IKYI
RMOU3N:  .FCTR   LB:[1,1]RMSLIB/LB:R3KSPL
RMOU3O:  .FCTR   LB:[1,1]RMSLIB/LB:R3ROOT-RMOU3N
RMOU3P:  .FCTR   LB:[1,1]RMSLIB/LB:R3MKID
RMSBLK:  .FCTR   RMSBMC-LB:[1,1]RMSLIB/LB:R0RWBI-RMMISC
RMMISC:  .FCTR   RMSMIS-RMMIS0
RMMIS0:  .FCTR   LB:[1,1]RMSLIB/LB:R0FREE:R1TRUN-RMMIS1
RMMIS1:  .FCTR   LB:[1,1]RMSLIB/LB:R0RWIN-RMWATR-RMMAGT
RMWATR:  .FCTR   LB:[1,1]RMSLIB/LB:R0WATR-RMR1WL-RMR1RL
RMMAGT:  .FCTR   LB:[1,1]RMSLIB/LB:R0MAGT
```

Appendix D
RMSDFN Command Utility

You should use the RMSDEF interactive utility (described in Section 9.3), rather than the RMSDFN command utility, to create RMS-11 files:

- RMSDEF enables you to define all attributes of the file.
- RMSDEF interacts with you as you define the file. The utility requests information according to your entries and provides HELP messages to guide your choices.
- RMSDEF is actively supported.

D.1 Purpose

The RMSDFN command utility creates RMS-11 files. You define or default file attributes in the command string invoking the utility. RMSDFN then stores the attributes within the newly created file.

NOTE

The conventions discussed in Sections 9.0.1 and 9.0.2 also apply to RMSDFN.

D.2 Effect

To create a file with RMSDFN, you need specify only the name and attributes of the file. Optionally, you can specify an initial allocation quantity. However, RMSDFN does not write records into the file. You can employ either an application program or the RMSIFL or RMSCNV utility to populate the file after RMSDFN creates it.

The attributes you can specify when creating a file vary with the file organization selected. However, all file definitions include the following information:

- file organization
- record format
- allocation quantity

- a contiguous or noncontiguous allocation
- system protection code

Indexed files also require a Primary Key. Optionally, you can define up to nine Alternate Key definitions per file, including whether the keys can be changed during update or may contain the same value in multiple records.

D.3 Call and Termination

D.3.1 Permanently Installed Utility

DFN [*command string*]

If you do not specify a command string, RMSDFN assumes control of the user interface and prints the prompt:

```
DFN>
```

You can type a command string or CTRL/Z to terminate the utility. When RMSDFN has executed a command string, it reprints the prompt.

If you include a command string, the utility attempts to execute it and then returns control to the operating system.

D.3.2 Uninstalled Utility

RUN $RMSDF

RMSDFN assumes control of the user interface and displays the prompt:

```
DFN>
```

You can type a command string or CTRL/Z to terminate the utility. When RMSDFN has executed a command string, it reprints the prompt.

D.4 Command String

D.4.1 General Form

outfile[/*switch*]...]

where:

outfile is the filespec of the file to be created. Wild card characters cannot appear in any field of the specification. The file specified should not exist in the account indicated.

switch can be a code shown in Table D-1 and described under "Switches," Section D.4.2.

Table D–1: RMSDFN Utility Switches

Type	Switch	Description	Default
Global	/ID	Identify current version of RMSDFN utility.	No id.
Outfile	/AL:*nnn*	Allocation quantity.	AL:0
	/BK:*nn*	Number of blocks per bucket.	BK:1
	/CO	Contiguous allocation.	NOCO
	/FO:*val*	File organization.	FO:SEQ
	/KY:*n:m*[:(/OP:*val*[:*val*])]	Key definition.	No keys.
	/PR:*options*	Protection specification.	/SY:RWED /OW:RWED /GR:RWED /WO:RWED
	/PR:*nnn*	Octal protection code.	See /PR:*options*
	/PR:<*nnn*>	Protection specification.	PR:<60>
	/RF:*val*[:*nnnn*]	Record format and record size.	RF:VAR:0

D.4.2 Switches

- ID causes RMSDFN to print its current version numbers, in the form:

  ```
  DFN -- VERSION 1.5
  ```

  This switch can appear alone as a command string.

- AL:*nnn* specifies the initial allocation quantity for the file, where *nnn* is a number of blocks.

- BK:*nn* specifies the number of blocks per bucket for Relative and Indexed files. The default is one block per bucket; the maximum value for *nn* is 32 or 15.

- CO indicates that RMSDFN should attempt the initial file allocation as physically contiguous blocks.

- FO:*val* indicates the file organization being created. You can specify one of the following values:

  SEQ - Sequential file (default)
  REL - Relative file
  IDX - Indexed file

  The value set for file organization determines what additional switches can or must appear in the command string. For example, if FO:IDX is specified, then at least one key definition (KY) must also appear.

- KY:*n:m*[:(/OP:*val*[:*val*])] defines an Indexed file key. You can specify a maximum of ten key definitions for a single Indexed file. RMSDFN interprets the first one in the command string (scanning from left to right) as the

Primary Key definition, the next one as the First Alternate Key, the next as the Second Alternate Key, and so on.

You must specify at least one key definition, that is, the Primary Key definitions for Indexed files.

The minimum general form of a key definition is:

/KY:*n*:*m*

where:

n is the location of the first byte of the key field within the record; 0 represents the first byte of the record, 1 the second, and so on.

m is the key field length in bytes; the maximum length for a key field is 255 bytes.

Example /KY:9:16

This switch defines a key field that begins in the tenth byte of each record and is sixteen characters long.

You can also specify key characteristics with the KY switch using the following general form:

/KY:*n*:*m*:(/OP:*val*[:*val*])

where:

val is one of the following:

DUP if duplicate key values are allowed to occur among records in the file. DUP is the default for Alternate Keys.

CHG if Alternate Key fields are allowed to change values during an update operation. CHG is the default for Alternate Keys.

NODUP if duplicate key values cannot occur among records in the file. NODUP is the default for Primary Keys.

NOCHG if key fields cannot change during an update operation. NOCHG is the default and only acceptable value for Primary Keys.

RMS-11 does not allow certain combinations of characteristics, depending on the type of key (Primary or Alternate) being defined. Table D-2 summarizes these combinations.

Table D-2: Key Characteristic Combinations

Key Type	CHG + DUP	CHG + NODUP	NOCHG + DUP	NOCHG + NODUP
Primary	Error	Error	Allowed	Default
Alternate	Default	Error	Allowed	Allowed

When specifying the OP switch, you must use the parentheses shown in the general form.

Example /KY:0:8:(/OP:DUP:CHG)

This switch defines an Alternate Key. In each record of the file, the key field begins in the first byte and is eight bytes long. Duplicate key values can occur among records of the file and key fields are allowed to change during update operations.

- PR:*options* specifies the system protection code for the file being created. Protection is available for the four IAS/RSX-11M categories:

  System - access by the system's accounts
  Owner - access by the owner
  Group - access given other members in the owner's project
  World - access given all other accounts

  You can specify whether the accounts in each category can read, write, extend, or delete the file. Therefore, the full format of the PR switch is:

  /PR[/SY[:RWED][/OW[:RWED]][/GR[:RWED]][/WO[:RWED]]

  where:

  /SY, /OW, /GR, and /WO — are the subswitches that specify protection categories for the file. If no subswitch is specified for a category, that category is allowed complete access to the file.

  R, W, E, and D — in any order, represent the Read, Write, Extend, and Delete privileges granted to a category. If a subswitch is specified, but no privileges are indicated, none are allowed for the category.

  Protection can also be specified as an octal value in the PR switch form:

  /PR:nnn

  where:

  nnn is the octal representation of the protection to be assigned to the file. Refer to the *RSX-11M Utilities Procedures Manual* for a description of the format of the octal protection word.

- PR:<*nnn*> specifies the protection status of the file. See the *RSTS/E User's Guide* for a discussion of the values of *nnn.*

- RF:*val*[:*n*] specifies the record format and size for the file,

  where:

  val is the record format expressed as one of the following:

  FIX - fixed-length
  VAR - variable-length
  VFC - variable-with-fixed-control; fixed control area defaults to two bytes; restricted to Sequential and Relative files
  STM - stream; restricted to disk Sequential files

n specifies one of the following:

- when FIX is specified, the actual record size; a nonzero value is required: there is no default length.
- when VAR, VFC, and STM are specified, the maximum record size in bytes. If you enter a nonzero value, RMS–11 checks records written into the file against this maximum length, rejecting those records that are too long. However, if 0 or no value is specified, RMS–11 does not check record length. A nonzero value is required for Relative files; that is, RMSDFN rejects the default value of 0 with the following message if file organization is REL.

```
?DFN -- ILLEGAL OPTION VALUE - /RF
```

If the format is VFC, the length does not include the fixed control area.

D.4.3 Examples

- DK1:ALPHA.DAT/FO:REL/CO/RF:FIX:128

  RMSDFN attempts to create a Relative file, with a minimum initial allocation (one block) made contiguously. The records are 128 bytes long. The bucket size defaults to one block.

- TEST.INP

  RMSDFN attempts to create a Sequential file named TEST.INP. The records are variable in format with no specified maximum length. The initial allocation is zero blocks.

- MASTER.FIL/FO:IDX/KY:0:20

  RMSDFN attempts to create this minimally specified Indexed file. The records are variable in format with no specified maximum length. Bucket size defaults to one block. The Primary Key begins in the first byte of each record and is 20 bytes long.

- TRANS.DAT/FO:IDX/BK:4/RF:FIX:112/KY:0:12-
  /KY:19:6/KY:29:4(/OP:NOCHG:DUP)

  RMSDFN attempts to create the Index file TRANS.DAT. Each bucket contains four blocks and the fixed-length records are 112 bytes long. Each record contains three key fields:

Primary Key	begins in the first byte of each record and is 12 bytes long. By default, duplicate key values and changes in key values during updates are not allowed.
First Alternate Key	begins in the twentieth byte of each record and is six bytes long. By default, duplicate key values and changes in key values during updates are allowed.

Second Alternate Key	begins in the thirtieth byte of each record and is four bytes long. By specification, the key is duplicatable, but not changeable.

D.5 Cautions

- RMSDFN creates Relative and Indexed files only on disk devices.
- RMSDFN does not limit the record size specified in the RF switch. A file therefore can be created with attributes indicating record sizes greater than the maximums shown in Chapters 3, 4, and 6. However, processing such a file causes unpredictable results.

Appendix E
Utility Error Messages

E.1 RMSDEF Interactive Utility

```
BAD UIC, TRY AGAIN
```

Description

The account number in the file specification is not valid for your computer system for one of the following reasons:

- Either one or both parts of the account number exceeds the maximum value allowed by your operating system:

Operating System	Range
IAS/RSX-11M	1 through 377 (octal)
RSTS/E	1 through 254 (decimal)

- The specified account has not been set up by the system manager.

Suggested Action

Type the file specification with a valid account number.

```
THE DEFAULT EXTENSION QUANTITY MUST BE A MULTIPLE OF THE BUCKET SIZE.
```

Description

You typed a decimal number of blocks for Default Extension Quantity, but that number is not an integral multiple of the bucket size you specified.

Suggested Action

Type the next higher multiple of bucket size for Default Extension Quantity.

```
** DEVICE IS FULL **
```

Description

RMSDEF attempted to create the file you defined, but could not allocate the number of blocks you specified in the way you specified.

Suggested Action

If you specified a large contiguous initial allocation for the file, redefine the file requesting either a smaller initial allocation or a noncontiguous allocation.

If you did not specify a large initial allocation, either contiguous or noncontiguous, then the specified disk is full. You should either:

- delete unnecessary files from the disk and reorganize empty blocks in a single contiguous part of the disk using the DSC utility
- use a different disk pack on the device

```
** DEVICE NOT AVAILABLE **
```

Description

RMSDEF attempted to create the file you defined, but could not access the specified device because it was not ready or on-line.

Suggested Action

Make the specified device ready or specify another device; and try again.

Description

RMSDEF attempted to create the file you defined, but the file processor indicates that the specified account has not been set up.

Suggested Action

Specify another account number or request the system manager to create the specified account; and try again.

```
ERROR IN LUN ASSIGNMENT
```

Description

RMSDEF assigned a logical channel erroneously.

Suggested Action

Recall the utility and try the dialog again. If this message occurs again, submit a Software Performance Report as described in Section 9.0.2.6.

```
ERROR WITH INDIRECT FILE
```

Description

You directed RMSDEF to read an indirect file, but the utility detected an error in the file that is not covered by other error messages.

Suggested Action

Recall the utility and specify the indirect file. If this message occurs call your local DIGITAL representative.

```
FILE SPECIFICATION IS TOO LONG
```

Description

You directed RMSDEF to read an indirect file, but the specified file contains either a file specification longer than 120 bytes or a line (record) longer than 80 bytes.

Suggested Action

Edit the indirect file, shortening the file specification or line. Recall the utility and specify the indirect file.

```
I-O ERROR IN PROCESSING INDIRECT FILE
```

Description

You directed RMSDEF to read an indirect file, but the utility detected an I/O error during its processing.

Suggested Action

Recall the utility and specify the indirect file. If this error occurs again, recreate the indirect file. If the new file fails, call your local DIGITAL representative.

```
ILLEGAL DEVICE, TRY AGAIN
```

Description

RMSDEF attempted to create the file you defined, but encountered an error: either the specified device is not on your computer system or the utility has erroneously assigned its logical channel to another device.

Suggested Action

If the device is not on your system, specify another device and try again. If the error occurs for a device recognized by your system, submit a Software Performance Report as described in Section 9.0.2.6.

```
ILLEGAL FILE SPECIFICATION, TRY AGAIN
```

Description

RMSDEF detected a syntax error in the file specification.

Suggested Action

Retype the file specification correctly.

```
ILLEGAL KEY SIZE, TYPE ? FOR HELP.
```

Description

You typed a key size that is not valid for the key data type you specified.

Suggested Action

Type a question mark (?) to print a HELP message showing the size restrictions for the different data types. Then, type a valid key size.

```
ILLEGAL VALUE, TYPE ? FOR HELP.
```

Description

You typed a value that is not a proper reply to the current question or request.

Suggested Action

Type a question mark (?) to print a HELP message that should help you determine the proper values. Then type a proper value for the question or request.

```
INVALID DATA IN COMMAND FILE
```

Description

You directed RMSDEF to read an indirect file, but the a line in that file did not satisfy the utility's data requirements. If you specified the indirect file at command level, RMSDEF returns to that level. However, if you specified the indirect file after the utility printed its prompt, RMSDEF continues its dialog with you rather than the indirect file.

Suggested Action

Correct or recreate the indirect file and try again, or continue the interactive dialog.

```
KEY EXCEEDS MAXIMUM RECORD SIZE.
```

Description

You typed a size for the key being defined which extends the key beyond the end of the record you defined.

Suggested Action

Type a smaller size or terminate the current session with a CTRL/Z and start again, defining a larger Maximum Record Size.

```
MAXIMUM NESTING DEPTH EXCEEDED IN INDIRECT FILE
```

Description

An indirect command file contains an indirect command. RMSDEF does not permit nesting of indirect commands.

Suggested Action

Type the contents of one or both of the indirect command files as a sequence of individual command strings.

```
OPEN ERROR IN INDIRECT FILE
```

Description

You directed RMSDEF to read an indirect file, but the utility could not open the file. The file may not exist as specified; it may be locked; or there is some other problem with the file.

Suggested Action

Correct the problem and try again.

```
** PRIVILEGE VIOLATION **
```

Description

RMSDEF attempted to create the file you defined, but the operating system denied the utility access.

Suggested Action

Correct the problem and try again.

```
WARNING - BE AWARE THAT THE FILENAME HAS BEEN TRUNCATED.
```

Description

In the file specification you just typed, you specified a file-name longer than nine or six characters. RMSDEF continues the interactive session, but the utility uses only the maximum number of characters allowed in the file-name when it tries to create the file.

Example On a RSTS/E system, you typed the filespec:

```
PAYROLL.DAT
```

RMSDEF uses only "PAYROL" when it tries to create the file.

Suggested Action

Continue the session unless you cannot accept the file-name truncation.

```
WARNING - BE AWARE THAT THE FILE TYPE HAS BEEN TRUNCATED.
```

Description

In the file specification you just typed, you specified a file extension longer than three characters. RMSDEF continues the interactive session, but the utility uses only the first three characters in the file extension when it tries to create the file.

Example You typed the filespec:

```
PAYDAY.FILE
```

RMSDEF uses only "FIL" when it tries to create the file.

Suggested Action

Continue the session unless you cannot accept the shorter extension.

```
YOU ENTERED A DIFFERENT NUMBER OF SEGMENTS IN THE KEY POSITION
QUESTION.
```

Description

You are defining a segmented key. However, the series of key sizes you typed does not contain the same number of segments as the series of key positions you typed for the last question.

Suggested Action

Type the key size series again, specifying the same number of segments that you used in the key position series.

```
YOU HAVE SPECIFIED A VIRTUAL BLOCK NUMBER WHICH IS NOT WITHIN
THE BOUNDARIES OF THE FILE.
```

Description

You are using Virtual Block Number (VBN) alignment to place a file area other than Area 0. However, the number you typed in response to the LOCATION: request is not a valid VBN for the file you have defined to this point.

Suggested Action

Recalculate the extent of the file you have defined to this point by adding the initial allocation quantities for all the areas defined, starting with Area 0. Use the sum to help you type a valid VBN.

```
YOU ORIGINALLY GAVE A BAD FILE SPECIFICATION WHICH WAS OUTPUT TO
YOUR COMMAND FILE. YOU MUST EDIT THE CHANGE IN YOURSELF.
```

Description

When you typed a filespec for the file you were defining, you made an error. RMSDEF allowed you to retype the filespec at the end of the interactive session, after it failed to create the file. However, you also directed the utility to create an indirect file during the session. That indirect file now has an incorrect file specification at its beginning.

Suggested Action

Before you can use the indirect file, you must change that invalid file specification with a file editor.

E.2 Command Utilities

The format of an error message indicates the severity of the condition encountered:

- A question mark (?) appearing as the first character of the message indicates a fatal error condition. Fatal error conditions cause the utility to terminate processing at the point the condition was encountered.
- The absense of a question mark as the first character of the message indicates a nonfatal error condition. When a nonfatal error condition occurs, the utility terminates the processing of only the file causing the error. The utility continues processing any additional files specified in the command string.

In both types of error messages, the three character name of the utility encountering the error precedes the main text of the message.

Italicized lowercase letters in the error messages listed in this section indicate that the utility substitutes a value in the message printed at your terminal. The following conventions indicate these substitutions:

CONVENTIONS

fipcode A numeric value representing a file processor error code. When an error message contains a code, refer to the description of such codes in the:

- Error code appendix of the *IAS/RSX-11 I/O Operations Reference Manual*
- User recoverable error messages in an appendix of the *RSTS/E Programming Manual.* Note that the code is the negative of the decimal value shown in the Programming Manual. That is, if the code is -20_{10}, look up 20_{10}.

rmscode A numeric value representing an RMS-11 error code. When an error message contains a code, refer to the description of such codes in Appendix B.2.

date A date you specified within the command line.

device A device specification.

account A User File Directory (UFD) or User Identification Code (UIC) specification.

filespec A file specification.

switch A switch appearing within the command string.

utl The three-character name of the utility encountering the error.

value A switch value appearing within the command string.

vbn A virtual block number.

utl -- CHECK AFTER WRITE ERROR ON OUTPUT FILE - *filespec*,
VBN *vbn1* TO *vbn2*, CONTINUE (Y,N)?

Description

This diagnostic error message prins on your terminal if Query Mode is enabled and the RC switch (check after writing) is specified. During data integrity checking, the utility found that the contents of virtual blocks *vbn1* through *vbn2* in the indicated file are not identical to the associated virtual blocks of an input file.

Suggested Action

Type "Y" to continue processing despite the error. Type "N" to terminate processing the indicated output file and bypass the processing of any remaining files.

?utl -- CLOSE ERROR ON FILE *filespec*, ERROR CODE= *fipcode*

Description

The specified input or output file could not be properly closed.

Suggested Action

Type the command string again. If the error recurs, run a validity check of the file structure using the VFY/CLEAN utility on the volume to determine if it is corrupted.

?utl -- COMMAND FILE NESTING DEPTH EXCEEDED

Description

An indirect command file contains an indirect command. RMS-11 does not permit nesting of indirect commands.

Suggested Action

Type the contents of one or both of the indirect command files as a sequence of command strings.

?utl -- COMMAND LINE TOO LONG

Description

The command string you typed is longer than the utility's command string buffer (158 characters). Therefore, you have typed too many continuation lines.

Suggested Action

Use several command strings to perform the functions of the original command string. If the condition occurs repeatedly, submit an SPR to DIGITAL.

?utl -- CONFLICTING OPTION - *switch*

Description

The indicated switch contradicts a previous switch within the same command string (scanning from left to right).

Suggested Action

Type the command string with the correct switch.

?utl -- CREATE ERROR ON *filespec*, ERROR CODE= *fipcode*

Description

The specified file could not be created by the utility.

Suggested Action

Refer to the appropriate manual.

?utl -- DEVICE NOT READY - *device*

Description

The message is self-explanatory.

Suggested Action

Correct the situation and type the command again.

?utl -- DEVICE OFF LINE - *device*

Description

The indicated device is on the system, but access is prohibited for one of the following reasons:

- The device is not ready.
- No volume is mounted on the device.
- The device is currently reserved by another job.
- The device requires privileges for use and you are not privileged.
- The device is disabled.

Suggested Action

Correct the problem and type the command string again.

?utl -- DEVICE WRITE PROTECTED - *device*

Description

The utility cannot access the device for write operations.

Suggested Action

Write enable the device.

utl -- DEVICE/FILE IS FULL - *device/filespec*

Description

The utility cannot create or extend the file on the device because of insufficient space.

Suggested Action

Type the command using another device for output files or copy the indicated file to another device and retry the command. You can also delete unneeded files on the indicated device and retype the command.

?utl -- DIRECTORY NOT FOUND - *filespec*

Description

The account does not exist on the specified device.

Suggested Action

Type the command with the correct account specification.

?utl -- DIRECTORY SYNTAX ERROR - *filespec*

Description

The account portion of the file specification does not conform to the syntax rules.

Suggested Action

Type the command string with the correct syntax.

utl -- DUP RCD= *record*

Description

The utility encountered an input record that would cause duplicate values in an output Indexed file key field that does not allow duplicates. The utility did not insert the record into the output file, but wrote the first 72 characters into the summary listing file.

Suggested Action

If the exception records are valid, redefine the output file to allow duplicates in the necessary key fields, then repeat the command. You could also insert the exception records into the current output file, changing the key values causing the duplication. You can do this by creating a Sequential file containing the records and using RMSCNV, or you can use RMSCNV with your terminal as the input file (*outfile*=TI:).

utl -- EMPTY FILE INDEX AREA

Description

The utility attempted to read records from an Indexed file using an Alternate index without entries. This condition can occur:

- when records written into the file are too short to contain the Alternate Key
- when occurrences of the Alternate Key field contain its null key value

Suggested Action

Type the command string again, specifying the Primary index or another Alternate index.

utl -- ENTER FILE IN DIRECTORY ERROR ON *filespec*, ERROR CODE= *fipcode*

Description

The specified file-name could not be entered in the account.

Suggested Action

Refer to the appropriate manual.

utl -- ERROR IN FILE PROLOGUE - *filespec*

Description

The utility encountered an error in the Prologue of the indicated file. The utility bypasses the file and continues processing.

Suggested Action

Move the disk pack to another device and try the command string again. If the utility processes the file normally, there was a read error on the first attempt. However, if the utility prints this error message again, you must recreate the indicated file:

1. Use RMSDEF to define and create a file, then use RMSIFL or RMSCNV to populate the new file from the corrupted one.
2. Use the last back-up copy of the file with the RMSRST utility.

?utl -- ERROR IN TEMPORARY FILE

Description

The wild card processor called by the utility encountered an error while creating a temporary file for resolution of wild cards in a file specification.

Suggested Action

Type successive command strings to achieve the result wild cards would produce.

?utl -- ERROR INSERT TOO BIG, ERROR CODE= *rmscode*

Description

The utility cannot print the text of an error message on your terminal.

Suggested Action

Submit an SPR to DIGITAL documenting the conditions under which the error occurred.

?utl -- ERROR WITH LOGICAL UNIT

Description

The utility assigned a logical channel erroneously.

Suggested Action

Recall the utility and try the command string again. If this message recurs, submit a Software Performance Report as described in Section 9.0.2.6.

?utl -- ERROR WITH WILDCARDS

Description

The wild card processor returned an error to the utility during resolution of wild cards in a file specification.

Suggested Action

Type the command string again. If the condition recurs, use successive command strings to achieve the result wild cards would produce.

?utl -- EXTEND ERROR ON OUTPUT FILE

Description

The output file you provided was not large enough to contain the input records. The utility explicitly extended the file, but the operation failed.

Suggested Action

Determine why the file could not be extended (such as, disk full, contiguous on RSTS/E, and so on). Allocate a larger output file and type the command string again.

?utl -- FAILURE TO CREATE CONTIGUOUS FILE - *filespec*

Description

You specified the CO switch, but there is insufficient contiguous space on the disk to allocate the specified file.

Suggested Action

Type the command without the CO switch. You could also delete unneeded files from the disk and reorganize the empty blocks into a contiguous space using the DSC utility.

?utl -- FAILURE TO OPEN COMMAND FILE

Description

An I/O or file system error, such as a privilege violation, an attempted write to a locked unit, parity error, and so on, occurred when the utility attempted to open an indirect command file.

Suggested Action

Correct the problem and retry the command.

?utl -- FATAL RMS ERROR - PC= *nnnnnn*, STS= *rmscode*, STV= *rmscode*, FILE= *filespec*

Description

A fatal error occurred in the utility's interface with RMS-11 file and record handling routines.

Suggested Action

Appendix B.2 describes the STS and STV codes.

?utl -- FILE ACCESS ERROR ON *filespec*, ERRORN CODE= *fipcode*

Description

The specified file could not be accessed.

Suggested Action

Refer to the appropriate manual.

?utl -- FILE ALREADY EXISTS - *filespec*

Description

The utility attempted to create a file that already exists.

Suggested Action

Type the command line using a new or corrected filespec or delete the existing file and type the original command string again.

?utl -- FILE, *filespec*, CORRUPTED

Description

The internal structure of an RMS-11 file is corrupted, and RMS-11 cannot read or write records in the file.

Suggested Action

Move the disk pack to another device and try the command string again. If the utility processes the file normally, there was a read error on the first attempt. However, if the utility prints this error message again, you must recreate the indicated file:

1. Use RMSDEF to define and create a file, then use RMSIFL or RMSCNV to populate the new file from the corrupted one.
2. Use the last back-up copy of the file with the RMSRST utility.

?utl -- FILE LOCKED - *filespec*

Description

The specified file was improperly closed the last time it was accessed and remains in a locked state.

Suggested Action

Use the UN switch with PIP to unlock the file.

?utl -- FILE NAME SYNTAX ERROR - *filespec*

Description

The name portion of the file specification does not conform to the syntax rules.

Suggested Action

Type the command string with the correct syntax.

?utl -- FILE NOT AVAILABLE - *filespec*

Description

The indicated file is being accessed for exclusive use by another job.

Suggested Action

Retry the command until the file is released.

[*?*]*utl* -- FILE NOT FOUND - *filespec*

Description

The indicated file was not found in the specified account and device.

Suggested Action

Verify the file specification and type the command string again.

?utl -- FILE POSITION LOST - *filespec*

Description

The utility lost its position within a container file on magnetic tape while rewinding or backspacing. The error may be caused by hardware failure.

Suggested Action

Determine from the output account or summary listing file the extent of the processing that was completed prior to the error. Type the command string eliminating file specifications of files successfully processed. Use a new tape volume and/or a different tape drive.

?utl -- FILE READ ERROR

Description

The utility encountered a hardware read error on an input or output device.

Suggested Action

Use CTRL/Z to terminate the utility. Check input and output devices for hardware problems.

?utl -- FILE NAME TOO LONG

Description

You typed a file specification longer than 120 characters.

Suggested Action

Type the command string with a shorter filespec.

?utl -- FILE TYPE SYNTAX ERROR - *filespec*

Description

The extension of the file specification does not conform to the syntax rules.

Suggested Action

Type the command string with the correct syntax.

?utl -- FILE VERSION SYNTAX ERROR - *filespec*

Description

The version portion of the file specification does not conform to the syntax rules.

Suggested Action

Type the command string with the correct syntax.

utl -- FIND FILE ERROR ON *filespec*, ERROR CODE= *fipcode*

Description

The indicated file could not be found.

Suggested Action

Refer to the appropriate manual.

?utl -- ILLEGAL DEVICE - *device*

Description

The device does not exist.

Suggested Action

Type the command string with a correct device specification.

?utl -- ILLEGAL FILE SPECIFICATION - *filespec*

Description

The file specification does not conform to the syntax rules.

Suggested Action

Type the command string with the correct syntax.

?utl -- ILLEGAL INPUT OPTION - *switch*

Description

The indicated switch is not a legal infile switch for the utility.

Suggested Action

Type the command string with the correct switch or without the indicated switch.

?utl -- ILLEGAL MAXIMUM RECORD SIZE - *filespec*

Description

The Maximum Record Size attribute stored in the indicated file exceeds the RMS–11 maximum. The file is corrupted.

Suggested Action

Move the disk pack to another device and try the command string again. If the utility processes the file normally, there was a read error on the first attempt. However, if the utility prints this error message again, you must recreate the indicated file:

1. Use RMSDEF to define and create a file, then use RMSIFL or RMSCNV to populate the new file from the corrupted one.
2. Use the last back-up copy of the file with the RMSRST utility.

?utl -- ILLEGAL NUMBER OF INPUT FILES

Description

You specified multiple input files for a utility that accepts only a single input file.

Suggested Action

Type the command specifying a single input file.

?utl -- ILLEGAL OPTION VALUE - *switch*

Description

The value specified in the switch is illegal.

Suggested Action

Type the command string with the correct switch value.

?utl -- ILLEGAL OUTPUT OPTION - *switch*

Description

The switch is not a legal outfile switch.

Suggested Action

Type the command string with the correct switch or without the indicated switch.

utl -- ILLEGAL RMS FILE ORGANIZATION - *value*

Description

The switch value does not represent a legal RMS-11 file organization.

Suggested Action

Type the command with the correct switch value.

?utl -- ILLEGAL RMS RECORD FORMAT - *value*

Description

One of the following conditions exists:

- The indicated switch value does not represent a legal RMS-11 record format.

- The indicated record format is not permitted with the file organization.

Suggested Action

Type the command with the correct record format.

?utl -- ILLEGAL RMS RECORD SIZE

Description

One of the following conditions exists:

- You specified a record size greater than 16,383 bytes.
- You failed to specify a Maximum Record Size greater than zero when creating a Relative file.
- The specified record size is too large for the buckets of the file.

Suggested Action

Type the command with the correct record size.

?utl -- ILLEGAL USE OF /AP, /BL, /MA OR /SU WITH FILE ORGANIZATION

Description

You combined one of the switches illogically with a file organization, as follows:

- AP or SU with a non-Sequential file
- BL with a disk file
- MA with a non-Indexed file

Suggested Action

Type the command string without the switch.

?utl -- ILLEGAL USE OF /LO WITH FILE ORGANIZATION

Description

You specified the LO switch in an RMSCNV command string. However, the output file is not an Indexed file.

Suggested Action

Type the command string without the LO switch.

?utl -- ILL USE OF /PD WITH OUTPUT FILE'S RECORD FORMAT

Description

You specified the PD switch in an RMSCNV command string, but the output file contains variable-length records.

Suggested Action

Type the command string without the PD switch.

?utl -- ILLEGAL USE OF /WF WITH RECORD FORMAT

Description

You specified the WF switch in an RMSCNV command string, but neither the input nor the output file contains variable-with-fixed-control (VFC) records.

Suggested Action

Type the command string without the WF switch.

?utl -- ILLEGAL USE OF WILD CARDS IN SUMMARY LISTING FILE

Description

You used asterisks in a file specification as the argument of an SL switch. RMS-11 does not permit wild cards in this context.

Suggested Action

Type the command string without wild cards in the summary listing file specification.

?utl -- ILLEGAL USE OF WILD CARDS ON INPUT FILE

Description

The utility does not permit asterisks in an infile specification.

Suggested Action

Type the command string without wild cards in the input file specification.

?utl -- ILLEGAL USE OF WILD CARDS ON OUTPUT FILE

Description

The utility does not permit asterisks in the outfile specification.

Suggested Action

Type the command string without wild cards in the outfile specification.

?utl -- ILLOGICAL DEVICE - *device*

Description

The indicated device is not permitted in the context of the command string.

Suggested Action

Type the command string with an appropriate device specification.

?utl -- ILLOGICAL USE OF OPTION - *switch*

Description

The indicated switch is illogical in the context of the command string.

Suggested Action

Type the command string with appropriate switches.

utl -- INCOMPLETE OUTPUT FILE REMAINS - *filespec*

Description

The error condition described by a prior error message caused the termination of processing before the indicated output file could be completed and properly closed.

Suggested Action

None.

?utl -- INCORRECT FILE ORGANIZATION - *filespec*

Description

The organization of the file does not match the organization you specified.

Suggested Action

Type the command line with the correct file organization.

?utl -- INDEXED OUTPUT FILE NOT EMPTY

Description

The output file you specified contains records.

Suggested Action

Specify or create another, empty output file.

?utl -- INPUT AND OUTPUT FIXED CONTROL HEADER SIZES DO NOT CORRESPOND

Description

You attempted to write records from one file to another. Both files contain variable-with-fixed-control (VFC) records, but the sizes of the fixed areas are different.

Suggested Action

Redefine the output file and type the command again.

?utl -- INPUT AND OUTPUT RECORD SIZES DO NOT CORRESPOND

Description

You attempted to write records from one file to another. However, one of the following conditions exists:

- Both files have fixed-length records, but the record sizes differ.
- Both files have variable-length records, but the Maximum Record Size of the input file is greater than the Maximum Record Size of the output file.

Suggested Action

If you are using RMSCNV, use the TR or PD switch or both (see Section 9.2.4.3). Otherwise, redefine the output file and retry the command.

utl -- INPUT FILE IS NOT BACK-UP FILE - *filespec*

Description

The utility requires the input file to be a back-up file. You have specified a file not in back-up format.

Suggested Action

Type the command string with the correct file specification.

utl -- INPUT RECORDS NOT IN ASCENDING ORDER

Description

You specified the MA switch to RMSCNV, but your input file was not sorted in ascending order by the output file's Primary Key value.

Suggested Action

Sort the input file or type the command again without the MA switch.

utl -- INTEGRITY CHECK TABLE FULL, CONTINUE (Y,N)?

Description

The table allocated internally by the utility to monitor input read errors exceeded its capacity.

Suggested Action

Type "Y" to continue processing. Type "N" to terminate processing.

?*utl* -- INVALID /KR VALUE

Description

You specified a key of reference greater than 9. The utility cannot process an Indexed file with more than ten defined keys.

Suggested Action

To process an Indexed file with more than ten keys, you must write an application program.

utl -- I/O ERROR ENCOUNTERED ON INPUT FILE *filespec*,
VBN *vbn1* TO *vbn2*, CONTINUE (Y,N)?

Description

The virtual blocks cannot be read from the file. If the file is input to RMSBCK, the utility cannot back up the data records within the blocks. If the file is input to RMSRST, the utility cannot restore the data within the blocks.

Suggested Action

Type "Y" if you want the utility to continue despite the error. Type "N" if you want the utility to terminate.

utl -- I/O ERROR ENCOUNTERED ON OUTPUT FILE - *filespec*

Description

One of the following conditions exists:

- The device is not on-line.
- The device is not mounted.
- The hardware failed.
- The volume is full.

Suggested Action

Correct the problem and type the command string again.

?utl -- I/O ERROR ON COMMAND LINE INPUT

Description

The utility cannot read:

- the command string you typed at the terminal
- a line within an indirect file

Suggested Action

Type the command line again or recreate the indirect file and retry the command.

?utl -- KEY OPTION COMBINATION CHG-NODUP ILLEGAL

Description

You attempted to create an Indexed file with an illegal combination of characteristics for one or more keys.

Suggested Action

Type the command with legal key characteristics.

?utl -- /KR NOT ALLOWED FOR SEQUENTIAL OR RELATIVE FILE

Description

The KR switch is permitted for Indexed files only. You either specified the wrong file or used the KR switch in the wrong context.

Suggested Action

Type the command with the correct file specification or without the KR switch.

?utl -- LABEL ERROR - *device*

Description

The magnetic tape volume does not have ANSI-standard labels.

Suggested Action

Mount the appropriate ANSI-labeled volume.

?utl -- MAXIMUM RECORD EXCEEDED - *filespec*

Description

No more records can be written into the indicated Relative file because the file's Maximum Record Number has been reached.

Suggested Action

Create a Relative file with the RMSDEF utility or an application program: specify an appropriate Maximum Record Number. Rerun the utility.

utl -- MISSING OPTION VALUE - *switch*

Description

You did not supply a value with the indicated switch.

Suggested Action

Type the command specifying a value for the indicated switch.

?utl -- NO DEFAULTS ALLOWED

Description

You did not specify a value for a quantity the utility cannot supply.

Suggested Action

Type the command string specifying all required values.

?utl -- NO KEY DECLARED FOR INDEXED FILE

Description

You attempted to create an Indexed file without defining a Primary Key.

Suggested Action

Type the command string with an appropriate Primary Key definition.

?utl -- NO RENAME ALLOWED

Description

The RMSBCK and RMSRST utilities require output file-names and extensions be the same as the corresponding input file-names and extensions. Only magnetic tape container files can have names that do not correspond to an input name.

Suggested Action

Type the command string specifying wild cards for the name and extension components of the output file specification.

utl -- NO SUCH FILE

Description

No file in the account you specified fits the wild cards in a file specification.

Suggested Action

List the files in the account. Type the command string again with a valid file specification.

?utl -- NO SUCH KEY FOR FILE - *value*

Description

The key of reference value you specified represents a key not defined in the specified Indexed file.

Suggested Action

Type the command with a valid key of reference value.

?utl -- NOT A DIRECTORY DEVICE - *device*

Description

You issued an account-oriented command for a device (such as a printer) that does not have directories (accounts).

Suggested Action

Type the command string without an account.

?*utl* -- NOT A SHARABLE DEVICE - *device*

Description

The command string you typed requires the sharing of a nonsharable device.

Suggested Action

Type the command string with a correct device specification.

?*utl* -- NOT ENOUGH MEMORY AVAILABLE

Description

The memory required by the utility is greater than the job's memory size maximum.

Suggested Action

Submit an SPR to DIGITAL documenting the conditions under which the condition occurred.

?*utl* -- ONLY [*,*] OR [X,Y] IS LEGAL AS DESTINATION

Description

You typed an outfile specification with an invalid account number. The only acceptable forms are:

- Wild cards as [*,*]
- Two numbers separated by commas as [*x*,*y*]

Suggested Action

Type the command string with an account specification of [*,*].

?*utl* -- OUTPUT FILE MISSING

Description

You did not specify an output file in the command string.

Suggested Action

Type the command string with an output file specification.

?*utl* -- OUTPUT FILE MUST BE AN INDEXED FILE

Description

You specified a Sequential or Relative file as an outfile.

Suggested Action

Specify or create another, Indexed output file.

?*utl* -- OUTPUT VOLUME CORRUPT - *filespec*

Description

While accessing the indicated file, the utility determined that the output volume is corrupted.

Suggested Action

Mount another volume or use a different device and type the command with appropriate device specifications.

?*utl* -- PRIMARY KEY CAN NOT CHANGE

Description

You specified that the Primary Key of an Indexed file can change. RMS-11 does not permit Primary Keys to change.

Suggested Action

Type the command without specifying change for the Primary Key.

?*utl* -- PRIVILEGE VIOLATION - *filespec*

Description

You do not have the privileges necessary to access the indicated file.

Suggested Action

Ask the owner of the file to change the protection code.

?*utl* -- PROLOGUE VERSION NUMBER TOO HIGH

Description

The utility opened an RMS-11 Indexed file and checked the version number of the file's prologue. That number indicates that the file was created with RMS-11 attributes that the utility cannot handle.

Suggested Action

Use a newer version of the utility or write an application program using the latest RMS–11 software to perform the processing.

utl -- READ AFTER WRITE ERROR ON OUTPUT FILE *filespec*
VBN *vbn1* TO *vbn2*, CONTINUE (Y,N)?

Description

This diagnostic error message prins on your terminal if Query Mode is enabled and the RA switch (read after writing) is specified. During data integrity checking, the utility encountered a read error while attempting to read virtual blocks *vbn1* through *vbn2* of the indicated file.

Suggested Action

Type "Y" if you want the utility to stop processing the indicated file, but process the rest of the command string. Type "N" if you want the utility to stop processing the command string.

utl -- READ ERROR, INTEGRITY CHECK TABLE AND REWRITE DATA MAY HAVE BEEN LOST

Description

The utility encountered an error while attempting to read formatting information in a back-up or container file.

Suggested Action

Retry the command specifying the RC switch . If the same error occurs, determine from the summary listing produced when the back-up or container file was created, which files cannot be completely restored. If you have other back-up copies of these files, use them with the command.

utl -- READ ERROR ON FILE ATTRIBUTES - *filespec*

Description

The volume is corrupted or you do not have the necessary privileges to access the indicated file.

Suggested Action

Use RMSDSP to determine the file's protection:

- If you are forbidden access to the file by its protection code, request that its protection be changed.
- If you are allowed access to the file by its protection code, then use the VFY /CLEAN utility to check the volume for corruption.

utl -- READ ERROR ON FILE PROLOGUE - *filespec*

Description

The utility cannot read the Prologue of the indicated file. The utility bypasses the file and continues processing.

Suggested Action

Type the command specifying only the indicated file. If the same error occurs, move the disk pack to another device and try the command string again. If the utility processes the file normally, a read error occurred on the first two attempts. However, if the utility prints this error message again, you must recreate the indicated file:

1. Use RMSDEF to define and create a file, then use RMSIFL or RMSCNV to populate the new file from the corrupted one.
2. Use the last back-up copy of the file with the RMSRST utility.

utl -- READ ERROR OR INCONSISTENT DATA, MAY HAVE LOST FILES.

Description

The utility encountered an error while reading a back-up or container file.

Suggested Action

Retry the command, specifying the RC switch. If the same error occurs, determine, from the summary listing produced when the back-up or container file was created, which files cannot be completely restored. If you have other back-up copies of these files, use them with the command.

?*utl* -- RECORD TOO BIG - *filespec*

Description

A record from the input file exceeds the Maximum Record Size specified for the output file.

Suggested Action

Use the RMSDEF utility to create a file with an appropriate Maximum Record Size.

?*utl* -- RELATIVE OUTPUT FILE NOT EMPTY

Description

The utility requires that a Relative output file not contain any records. However, the utility found one or more records in the output file.

Suggested Action

Read "Outfile Switches," Section 9.2.4.3, for the use of the TR and PD switches. Then, type the command again using one or both of the switches.

?utl -- SYNTAX ERROR *string*

Description

The format of the command string does not conform to the syntax rules. The utility prints the command string from the point of the error to the end of the line.

Suggested Action

Refer to the appropriate utility description in Chapter 9. Type the command string with the correct syntax.

?utl -- TI LUN ASSIGNMENT ERROR

Description

The utility cannot communicate with your terminal.

Suggested Action

Consult with a DIGITAL Software Support specialist.

?utl -- TOO MANY KEYS - *filespec*

Description

The specified Indexed file has more than ten keys defined. The utility cannot process an Indexed file with more than ten keys.

Suggested Action

You must write an application program to process an Indexed file with more than ten keys.

?utl -- TOO MANY SELECTIVE FILE SPECIFICATIONS

Description

More than ten file specifications appear as switch values of the SE switch.

Suggested Action

Type the command line specifying no more than ten file specifications as switch values of the SE switch. Use multiple command strings to select additional files.

utl -- UNABLE TO RESTORE SPECIAL ATTRIBUTES - *filespec*

Description

The utility cannot restore the file with one or more of the following attributes from the original file:

- one or more of its original date attributes
- its original protection specification
- contiguity
- placement control
- areas

Suggested Action

Use the RMSDSP utility to determine which attributes of the file were not restored.

utl -- UNCORRECTABLE ERROR, VBN *vbn1* TO *vbn2*
REASON= *message*

where:

message is one of the following:

READ CHECK ERROR

WRITE CHECK ERROR

INPUT ERROR

OUTPUT UNCHECKABLE

Description

This diagnostic error message prints on your terminal if Query Mode is enabled. During automatic retry of errors found during data integrity checking (RA or RC switch) or while reading a file, the utility encountered an uncorrectable error.

Suggested Action

Record the ranges of erroneous virtual blocks in the file.

?*utl* -- UTILITY INPUT ERROR

Description

A severe, unrecoverable error occurred during the execution of the indicated utility.

Suggested Action

Type the command again. If the same message appears, contact a DIGITAL Software Support specialist.

?utl -- WRITE ERROR ON ATTRIBUTES OF FILE - *filespec*

Description

The volume is corrupted or you do not have the necessary privileges to write the file.

Suggested Action

Verify access to file.

?utl -- WRITE ERROR ON CREATE OF OUTPUT FILE - *filespec*

Description

One of the following conditions exists:

- The device is not on-line.
- The device is not mounted.
- The hardware failed.
- The volume is full.

Suggested Action

Rectify the condition and type the command string again.

?utl -- WRITE ERROR ON INTEGRITY CHECK TABLE ON OUTPUT FILE - *filespec*

Description

The utility cannot write internal data integrity checking tables in the output back-up file.

Suggested Action

If the output medium is magnetic tape, use a different tape volume and retry the command. If the output medium is disk, rename the output file so that the utility will not attempt to use the space and retry the command.

Appendix F
Magnetic Tape Handling

RMS-11 supports only the magnetic tape structure defined by the American National Standards D (ANSI-D) format. This standard is described "Magnetic Tape Labels and File Structure for Information Interchange," ANSI X3.27-1969.

RMS-11 enforces only one part of the ANSI-D standard; the operating system's file processor enforces the rest, primarily the tape labeling formats. As a result, magnetic tape handling is largely transparent to RMS-11 and the RMS-11 user.

This appendix contains the few RMS-11 processing options that differ with magnetic tape.

For documentation about using magnetic tapes on your system, including creating ANSI-standard tapes and tape label formats, see:

- For IAS, the *IAS System Management Guide* and the *IAS/RSX-11 I/O Operations Reference Manual*
- For RSTS/E, the *RSTS/E System User's Guide*, *RSTS/E System Manager's Guide*, and *RSTS/E Programming Manual*
- For RSX-11M, the *RSX-11M Operator's Manual* and the *IAS/RSX-11 I/O Operations Reference Manual*

F.1 General Magnetic Tape File Processing

Magnetic tape is a sequential access, single-directory medium. Only one user can access a given tape reel, called a *volume*, at a time. No more than one file in a volume can be open at a time.

You can write files on tape volumes in the following combinations:

- Single file on a single volume
- Single file on more than one volume
- Multiple files on a single volume
- Multiple files on more than one volume

NOTE

When you have more than one tape volume that are processed together, the related volumes are called a *volume set*. And when a file is stored on more than one volume, the part of the file stored on each volume is called a *file section*.

The file processor distinguishes between volumes and between files with *labels*, short sections of tape written to a defined format. The following labels are used on ANSI-standard tapes.

- Volume label (VOL1)
- End-of-volume labels (EOV1) and (EOV2).
- File header labels (HDR1) and (HDR2).
- File trailer labels (EOF1) and (EOF2).
- User labels (not supported by RMS–11 or the file processors)

The following graphics illustrate volume structures supported by PDP–11 operating systems according to the ANSI–D standard.

NOTE

These conventions are used in the graphics:

- An asterisk indicates a tape mark.
- BOT indicates Beginning-of-Tape.
- A comma indicates a physical record delimiter.

Single File on a Single Volume

```
BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**
```

Single File on Multiple Volumes

```
BOT,VOL1,HDR1,HDR2*---DATA---*EOV1,EOV2**

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**
```

Multiple Files on a Single Volume

```
BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2*HDR1,HDR2*---DATA--*  EOF1,EOF2**
```

Multiple Files on Multiple Volumes

```
BOT,VOL1,HDR1,HDR2*--DATA--*EOF1,EOF2*HDR1,HDR2*---DATA--*EOV1,EOV2**
```

```
BOT,VOL1,HDR1,HDR2*--DATA--*EOF1,EOF2*HDR1,HDR2*--DATA--*EOF1,EOF2**
```

F.2 RMS–11 Magnetic Tape File Processing

RMS–11 requires the unused characters at the end of a block on magnetic tape to be circumflexes (^):

- While reading a tape, RMS–11 returns the error code ER$ANI if it encounters any character other than a circumflex between the last record in a block and the end of the block.
- When writing a tape, RMS–11 monitors the end of each tape block, whose length is set when you create the file. RMS–11 checks each put operation to see if the specified record fits in the tape block. If it does not, RMS–11 writes circumflexes to the end of the block and then starts a new block with the specified record. Records do not span blocks in magnetic tape files.

F.2.1 Rewinding Tape Volumes

You can rewind a magnetic tape volume when a file on it is created, opened, or closed. You specify one of the following options:

- Rewind-on-open as input to a file creation operation.
- Rewind-on-open as input to a file open operation.
- Rewind-on-close as input to a file creation operation.
- Rewind-on-close as input to a file open operation.
- Rewind-on-close as input to a file close operation.

NOTE

If you specify the rewind-on-close option during file creation or open, you cannot reset that requirement before you close the file. However, if you do not request rewind-on-close during the create or open, you can select the option before initiating the close operation and the file processor rewinds the tape after it closes the file.

Glossary

15–bit Signed Integer (Key)

Same as Two-Byte Signed Integer.

16–bit Unsigned Binary (Key)

Same as Two-Byte Unsigned Binary.

31–bit Signed Integer (Key)

Same as Four-Byte Signed Integer.

32–bit Unsigned Binary (Key)

Same as Four-Byte Unsigned Binary.

Access

1. The ability to read or write data in a file.
2. To read or write data in a file.

Access by Record's File Address

A Record Access Mode where a program can randomly access a record by its Record's File Address.

Access Declaration

Included in the information a program provides when it opens a file, the access declaration indicates the record operations the program will perform on the file. Contrast with Allow Declaration.

Access Path

The sequence of steps RMS-11 performs to transform the parameters in the user's record operation request into the requested record. See also Index, Key.

Access Time

The period between the initiation of a record operation and its completion. During this period, RMS-11 reads, writes, and/or modifies portions of a file as required by the record operation.

Active Page Register

A hardware register used by the operating system to map the virtual address space used by a task onto physical memory.

Allocation

The disk blocks associated with a file; normally, the size of a file established when the file is created.

Allow Declaration

Included in the information a program provides when it opens a file, the allow declaration indicates the type of record operations the program allows other programs to perform on the file. Contrast with Access Declaration.

Alternate Index

A structure providing a secondary logical access path to records stored in an RMS-11 Indexed file. Each Alternate index is based on a related Alternate Key field in the user data record. See also Primary Index.

Alternate Key

A series of bytes in a data record that can be used to identify the record for access. Alternate Key value does not affect the position of the user data record in the file. Contrast with Primary Key. See also Segmented Key.

Application Program

A program that processes data for an end user of a computer system. Normally, any program not part of the operating system. See also Program, Utilities.

APR

Same as Active Page Register

Area

A portion of an Indexed file treated independently by RMS-11 for initial allocation, extensions, placement, and bucket sizes.

Area Descriptors

Data used by RMS-11 to maintain areas in an Indexed file. RMS-11 stores this data in the file's Prologue. See also Key Descriptors.

Assembler

Software that converts assembly-language mnemonic instructions to object code. The assembly language for the PDP-11 is called MACRO-11. See also Compiler.

Asynchronous Record Operation

A record operation where RMS-11 may return control to your program before the operation is finished. The program continues processing while the physical transfer(s) of data between disk and memory is carried out. Contrast with Synchronous Operations.

Back up

1. The process of copying data to a medium that is preserved in case the original data is lost or destroyed.
2. Result of that process.
3. RMSBCK utility.

See also Restore.

Binary Search Technique

A searching technique applicable only to items ordered by the value in the search argument field. A binary search proceeds as follows:

1. Locate an item at the middle of the items searched.
2. Divide the items into three parts: low part, high part, and the item found.
3. If the item found does not satisfy the search, repeat this procedure on the low or high part depending on a comparison of the search value with the item found.

Bit

The smallest storage location recognized by PDP-11 hardware. A bit is a hardware location (in physical memory, on a disk or tape surface, and so on) that can assume one of two recognized values, conventionally designated "0" and "1". See also Byte, Block, Track, Cylinder.

Block

1. A logical unit of disk storage, containing 512 bytes.
2. A logical unit of magnetic tape storage, containing from eight through 8192 characters. See also Virtual Block, Logical Block Number, Sector.

Block I/O

The mode of file access that bypasses RMS-11 record processing. Contrast with Record Processing. See also Undefined Record Format.

Block Spanning

You must specify whether you allow records to span blocks when you create Sequential files. If you allow block spanning, records can cross block boundaries and are not restricted by block size. In Relative and Indexed files, records can automatically span blocks if a bucket contains more than one block; however, records cannot span buckets.

Bootable Volume

A disk containing a bootstrap loader program in Logical Block 0. Since this boot block is not a part of any file, RMSBCK and RMSRST do not read and write it.

Bucket

The I/O and disk storage unit for Relative and Indexed files.

Bucket Header

Fifteen bytes of control information that RMS-11 uses in each Indexed file bucket. See also Record Header, Record Format Overhead.

Bucket Locking

A process administered by the operating system between files and by RMS-11 between Record Access Streams.

RMS-11 activates system-level bucket locking for a Relative or Indexed file when the first program to open it allows write sharing. During record operations, RMS-11 requests the operating system to lock specific virtual blocks comprising the buckets being used.

RMS-11 maintains its own locked-block list when a program connects more than one Record Access Stream to a Relative or Indexed file.

When one Record Access Stream locks a bucket, no other program or stream can access it. See also Sharing Specifications in Accessing Programs.

Bucket Pointer

In this manual, part of an index record that indicates the bucket whose High-Key Value is contained in the rest of the index record.

Bucket Size

The number of blocks in a bucket. Bucket size is an attribute of a Relative file or an Indexed file area and cannot be changed for the life of the file.

Bucket Splitting

The process of formatting a new bucket in the file and moving the high portion of the target bucket into the new bucket. Bucket splitting occurs during put and update operations when both of the following are true:

- RMS–11 must insert a record into the target bucket to preserve ascending key value sequence or expand a record already in the target bucket.
- The record will not fit in the target bucket because there is not enough free space.

Buffer

A part of the virtual memory dedicated to a task. A buffer is a holding area for data moved to and from other storage areas. See also I/O Buffer, User Buffer.

Byte

A hierarchical unit of disk and memory structure, containing eight bits. See also Sector, Track, Cylinder.

Cache, Caching

The process of storing blocks in memory against future need; used to minimize physical transfer of data between mass storage devices and memory.

Cache Cluster

A single- or multi-block unit of the RSTS/E data cache. The caching software imposes a grid of cache clusters on the logical structure of a disk. An I/O operation that reads blocks to be cached is broken into disk access requests by cache cluster. This mechanism requires careful alignment of cache, pack, and file clusters.

Carriage Control

A type of forms control carried as a file attribute. When records from such a file are written directly to a unit record device, the device driver puts a line feed character before the record and a carriage return character after the record before passing it to the device.

Cell

Same as Record Storage Cell.

Change Key (Characteristic)

A flag associated with an Alternate Key. The flag indicates whether you allow or do not allow the key field to change values during an update record operation. See also Duplicate Key, Null Key.

Check-After-Write

An optional data integrity check performed by the RMSBCK and RMSRST utilities. See also Read-After-Write.

Close (File Operation)

RMS-11 terminates access to a file and releases control structures associated with the file. Contrast with Open.

Cluster

A sequence of logically contiguous blocks treated as a unit. See also Cache Cluster, Device Cluster, File Cluster.

Clustersize

The number of blocks in a cluster.

Command File

Same as Indirect File.

Command String

A series of characters, terminated with the RETURN key, with which you instruct a utility to perform processing. A command string normally includes one or more file specifications and optional switches.

Compiler

Software that converts higher-level-language instructions to object code. See also Assembler.

Concatenate

To string separate entities together end-to-end. Especially, overlay segments. See also Contiguity.

Connect (Record Operation)

RMS-11 makes the records of a file available to your program for a stream of operations. Contrast with Disconnect.

Container File

A file on magnetic tape containing one or more RMS-11 files in back-up format. When the output medium is tape, each command string to RMSBCK produces one container file.

Context

The position of a Record Access Stream within a file, consisting of Current Record and Next Record.

Contiguity

The property of having all parts touching. A contiguous file contains a continuous series of logical blocks; all logical blocks between the first and the last ones, inclusive, belong to the file.

Continuation Bucket

An Indexed file bucket containing part of a series of data records with the same key value. Duplicate series can exist only in Level 0 buckets. When a duplicate series grows beyond a single bucket, RMS-11 continues it in a separate Continuation Bucket that is not represented by an entry in Level 1. All records in a Continuation Bucket contain the same key value.

Control Structure

A part of virtual memory used by RMS-11 routines to communicate with the program and with each other. See also I/O Buffer, User Buffer.

Convention

A method or structure of presentation that enables easier communication.

Convert

1. To use the RMSCNV utility to copy data from one file to another. Since the files may have different attributes, the data must be converted from its input form to the proper output form.
2. RMSCNV utility.

Crash Dump

The output from an RMS-11 utility when it cannot recover from an error condition.

Create (File Operation)

RMS-11 passes all necessary information to the file processor, then requests it to create a file as specified. Contrast with Erase. See also Open.

Creation Time

That time when you actually instruct the file processor to create a file.

Current Record

Part of the context of a Record Access Stream. The Current Record is:

- established by a successful find or get operation
- the target of a delete, get (if immediately preceded by a find), truncate, or update operation.

Since only get or find operations set the Current Record, one of these operations must precede an update or delete operation. Other operations leave the stream without a Current Record. RMS-11 rejects any update or delete operation attempted without a Current Record.

See also Next Record.

Cylinder

The tracks at a single radius on all platters of a disk. See also Bit, Byte, Sector, Track.

Data Bucket

A bucket in an Indexed file that contains either user data records or SIDRs. See also Index Bucket.

Data Integrity Check

An optional process performed by RMSBCK or RMSRST to validate the data that has been transferred from one volume to another. See also Check-After-Write, Read-After-Write.

Data Level

The lowest level of an index, containing data buckets. See also Index Level, Lowest Index Level.

Data Record

Same as User Data Record. Also a SIDR in Alternate indexes. See also Index Record.

DCN

Same as Device Cluster Number.

Default Extension Quantity

The number of blocks that RMS–11 requests the file processor to add to a file when RMS–11 must extend the file automatically to complete a record operation. See also Allocation, Extend.

Default Value

The value supplied for an operation parameter when you do not provide an explicit value.

Deferred Write

An RMS–11 I/O technique. RMS–11 does not write the data in an I/O buffer to disk until it must use that buffer for other data.

Delete (Record Operation)

RMS–11 marks a record in a file, indicating that the record is no longer valid.

Depth

The number of the Root level in an index.

DEQ

Same as Default Extension Quantity.

Design

The process of considering the goals of an application and its files and the environment RMS-11 provides, and then planning both for optimal performance.

Device Cluster

A single- or multi-block unit of disk storage assignment whose size is characteristic of a type of device.

Device Driver

Software written for a specific type of hardware device that instructs devices of that type during data transfer and other operations. See also File Processor.

Directory

A file on a mass storage device that describes the layout of the data on that device in terms of file-names, lengths, locations, last date of access, protection codes, and so on.

Directory Caching

A RSTS/E technique. Directory blocks are cached in memory.

Disconnect (Record Operation)

RMS-11 terminates a stream of record operations, making the buffers assigned to the stream available for other operations. Contrast with Connect.

Disk Head

The electro-mechanical device that reads and writes information on disk platters.

Disk-Resident Overlay (Structure)

1. A segment of a task that resides on disk until required by the memory-resident portion of the task.
2. The tree-like structure that relates the segments. Contrast with Memory-Resident Overlay.

Display

1. To print the attributes and structural data of a file on a terminal or to a file with the RMSDSP utility.
2. To make file attributes and structural data available to a program with the RMS-11 $DISPLAY file operation.
3. RMSDSP utility.

Duplicate Key (Characteristic)

A flag associated with a Primary or Alternate Key. The flag indicates whether you allow or do not allow more than one record in a file to contain the same value in a specific key field. See also Change Key, Null Key.

Duplicate Pointer Array

A series of record pointers stored in each SIDR for an Alternate Key where duplicates are allowed. Each pointer indicates a user data record containing the key value represented by the SIDR.

Dynamic Access

The ability to change Record Access Mode with each record operation. In PDP-11 COBOL, you must declare dynamic access when you open a file. RMS-11 does not make this requirement.

End-of-File (Attribute)

The location of the end of useful data stored in a file. The location indicated is not necessarily the last block in the file.

Equal Access Times

All records in an RMS-11 Indexed file can be accessed by Primary Key with the same number of I/O operations, regardless of how long they have been in the file.

Erase (File Operation)

RMS-11 requests the file processor to delete the file from the device directory and release its blocks for re-use. Contrast with Create.

Exclusive Access to a File

A type of file access that prohibits concurrent access to a file by any other task. Not available through RMS-11.

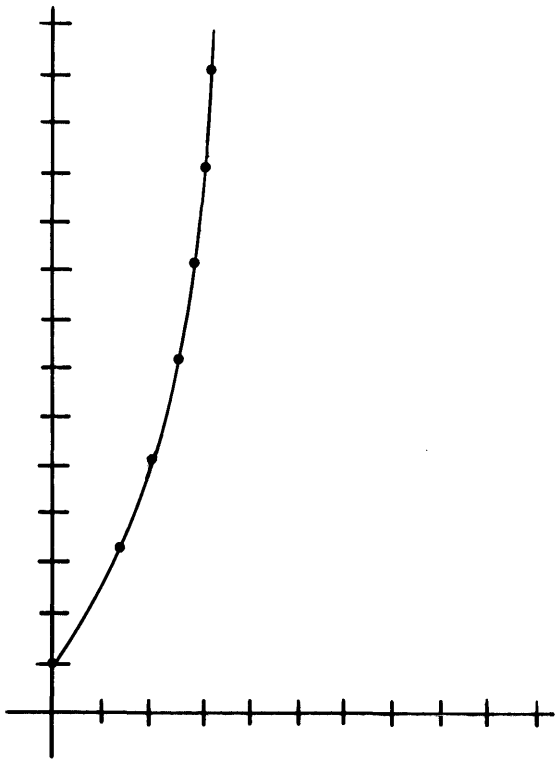
Executable Task

Same as Task.

Executive

In IAS and RSX-11M, the part of an operating system that coordinates the other components. See also Monitor.

Exponential Variance



Extend (File Operation)

RMS-11 requests the file processor to add blocks to a file's allocation. See also Default Extension Quantity.

Extended Buffer Pool

In RSTS/E, a portion of a computer's physical memory that is reserved for use by the operating system. Message send/receive, the DECnet/E package, the RSTS/2780 package, the FIP buffering module, and the data caching module use this reserved memory; if none is available, they use small buffers.

Extended Diagnostic Messages

A feature of RMSBCK and RMSRST: the utility prints a detailed error message on your terminal, allowing you to decide whether to continue processing. Also called Query mode.

Extent

A portion of a file containing contiguous blocks that is not recognized by the file processor as being contiguous with other portions of the file. See also Allocation.

F11ACP

Same as FILES-11 Ancillary Control Processor.

FAB

Same as File Access Block.

FCB

Same as File Control Block.

FCS–11

Same as File Control Services.

Fifo

Same as First-In, First-Out.

File

A logical container of data, individually maintained by the file processor without regard to the nature of its contents.

File Access Block

An RMS–11 control structure that represents a file during file operations.

File Attributes

Characteristics of a file stored in the file directory and Prologue. They include file organization, record format, and so on.

File Cluster

A single- or multi-block unit of disk storage assignment. File cluster is the minimum unit used by the file processor during file allocation and extension. File clustersize is a file attribute.

File Control Block

In IAS and RSX–11M, a memory-resident structure used by the operating systems to coordinate access to a file opened by an active task.

File Control Services

A file management software product available on IAS and RSX–11M operating systems.

File Id

An identifying notation optionally returned when a file is created or opened by file specification; can be used to open file.

File Operation

A file-level function, including create, open, close, extend, and erase. See also Record Operation.

File Organization

Method of arranging records within a file. File organization establishes the availability of access methods, record operations, and other ways of using the file.

File Processor

1. A component of the operating system that maintains the structure and integrity of data storage on file-structured devices.
2. In RSTS/E, the file processor.

See also FILES-11 Ancillary Control Processor.

File Specification

A string of identifiers that specify a particular file or class of files.

FILES-11 Ancillary Control Processor

In IAS and RSX-11M, the file processor.

FILES-11 ODS-1

On-Disk Structure Level One.

FILES-11 ODS-2

On-Disk Structure Level Two.

Filespec

Same as File Specification.

Fill Number

The number of bytes in an Indexed file bucket that you want used to store records. Data level and index level fill numbers can be set for each index in the file, but are honored by RMS-11 only during operations that so request. By default, RMS-11 uses all free bytes in a bucket to store records.

Find (Record Operation)

RMS-11 locates the record specified, but does not move it from the I/O buffer to the user buffer.

FIP

Same as File Processor

First-In, First-Out

Pertaining to a technique of storing serial values. The first value stored is the first one found in a sequential access. RMS-11 uses this technique when it stores user data records containing duplicate Primary Key values and the SIDR pointers to duplicate Alternate Key values.

Fixed-Length (Record Format)

Records in the file are the same length. The length is a file attribute enforced by RMS-11 during write-type record operations. See also Stream, Variable-Length, Variable-with-Fixed-Control, Undefined.

Flush (Record Operation)

RMS-11 writes all I/O buffers to a file if they haven't already been written.

Follow the Index

The procedure that RMS-11 uses when it must start at the Root and read down an index to a specific record location.

Forms Control

Codes passed to a unit record device that control the printing or display of data on the device. See also Carriage Control, FORTRAN Forms Control.

FORTRAN Forms Control

A type of forms control carried as a file attribute. The device driver interprets the first byte of each record as a FORTRAN forms control character.

Four-Byte Signed Integer (Key)

A key data type that can represent the decimal integer values -2,147,483,648 through +2,147,483,647. See also Two-Byte Signed Integer, Two-Byte Unsigned Binary, Four-Byte Unsigned Binary.

Four-Byte Unsigned Binary (Key)

A key data type that can represent the decimal integer values 0 through +4,294,967,295. See also Two-Byte Unsigned Binary, Two-Byte Signed Integer, Four-Byte Signed Integer.

Free (Record Operation)

RMS-11 releases buckets locked by a Record Access Stream. See also Bucket Locking.

Get (Record Operation)

RMS-11 locates the specified record and normally moves it from the I/O buffer to the user buffer.

Hashing

A programming technique where a key value is converted to a relative record number used for random access.

HELP Message

A message printed by an RMS-11 utility in response to a question mark (?) or the word "HELP." The message helps you use the utility.

High-Key Value

The key value of the last record in an Indexed file bucket, where:

- The last record in a bucket has an equal or higher key value than any other record in the bucket.
- The last record in a bucket has a lower key value than the first record in the next bucket in the chain, neglecting Continuation Buckets if present.

Both data and index buckets have High-Key Values. The key value used in the Primary Level 0 is the Primary Key value.

High Portion

Those records in an Indexed file bucket with key values higher than the record being inserted or updated. During a bucket split, the high portion of the data is moved to a newly created bucket.

Higher Level Language

In this manual, BASIC-PLUS-2, DIBOL, PDP-11 COBOL, and RPG II.

Highest Index Level

Same as Root.

Highest Possible Key Value

A logical maximum whose form depends on key data type; RMS-11 uses the highest possible key value to mark the logical end of an index level.

I/O Buffer

A portion of a task's virtual address space used to store data meant for or arriving from peripheral devices. See also Control Structure, User Buffer.

I/O Operation

The process of requesting a transfer of data from a peripheral device to memory, or vice-versa, the actual transfer of the data, and the processing and overlaying activity to make both of those happen.

I/O Techniques

Programming techniques used to improve the performance of record operations. See also Deferred Write, Mass Insert.

I/O Unit

The data moved in and out of a task during an I/O operation. For disk Sequential files, the I/O unit is one or more blocks, depending on the MBC value; and for Relative and Indexed files, the I/O unit is the bucket.

Incremental Reorganization

The process of inserting each data record where it logically belongs in Level 0 and updating the upper levels of an index.

Index

The structure for the logical access path to a data record. An Indexed file contains one index for each key defined for the file. Each index contains index records that guide RMS-11 through the index levels to the data records in Primary Level 0.

Index Bucket

A bucket in an Indexed file that contains index records. See also Data Bucket.

Index Descriptor

A memory-resident copy of a Key Descriptor's fields that are required for normal operations.

Index Level

The higher levels (1+) of an index, containing index buckets. See also Data Level, Lowest Index Level.

Index Record

A record maintained by RMS-11 in Levels 1+ of all indexes. The record contains a High-Key Value for a bucket in the next lower level and a pointer to that bucket. See also Data Record.

Indexed File

An RMS-11 file created with the Indexed file organization.

Indexed File Load

An RMS-11 utility that bypasses normal access methods to load an Indexed file with records from a file you designate. RMSIFL optimizes the structure of all indexes in the file it populates. See also RMSCNV.

Indexed File Organization

The method of organizing records in a file so that the records are sorted in ascending Primary Key order and one or more indexes are built that provide access path(s) to those records. See also Sequential File Organization, Relative File Organization.

Indirect File

A file containing a sequence of commands that can be interpreted by a single task, usually a system task such as a utility. These commands appear in the indirect file exactly as they are entered from your terminal. As such, they can be command strings or answers to questions.

Infile

A convention for the term "input file specification."

Initialize (Bucket)

When RMS-11 allocates a bucket for a Relative file, during an explicit extend file operation, for instance, RMS-11 initializes the bucket by setting all bits to zero. However, RMS-11 does not format Indexed file buckets until it must use the bucket to store a record.

Input File

The source of records for processing by a utility. Contrast with Output File.

Key

Identifier for a record during random record operations, used especially for Indexed files. A key is the contents of one or more specific portions of each user data record; the combination of these portions is called a key field. For each key defined for a file, RMS-11 constructs an index in the file. See also Segmented Key.

Key Characteristics

Features of individual keys in an Indexed file, including duplicates, changes, and null key.

Key Data Type

Feature of a key indicating to RMS-11 how to interpret the data in the associated key field. Supported types include string, two- and four-byte integer and binary, and packed decimal.

Key Descriptors

Data used by RMS-11 to maintain keys in an Indexed file. RMS-11 stores this data in the file's Prologue. See also Area Descriptor.

Key Field

Concatenation of one or more portions of a user data record. The key field contains the value for the associated key. See also Segmented Key.

Key of Reference

A number indicating the index RMS–11 should follow during a random record operation. A key of reference of 0 indicates the Primary index, while a key of reference of 1, 2, and so on, indicates the First Alternate index, Second Alternate index, and so on.

Key Value

Value of the data in the field associated with the key.

Least Significant Byte

The last byte examined when RMS–11 performs key value comparisons. Key data type determines whether the lowest-addressed byte (binary and integer data types) or highest-addressed byte (string and packed decimal data types) is the least significant. Contrast with Most Significant Byte.

Level

A chain of buckets in an Indexed file. All buckets in a level have the same conceptual function: Level 0 buckets contain data records, either user data records or SIDRs; Level 1 buckets contain index records that point to Level 0 buckets and indicate their High-Key Values; Level 2 buckets contain index records that point to Level 1 buckets and indicate their High-Key Values; and so on to the single-bucket level called the Root.

Library

In this manual, same as RMS–11 Resident Library.

Locate Mode

A Record Transfer Mode where RMS–11 allows a user program to manipulate data in the I/O buffer instead of the user buffer. Contrast with Move Mode.

Logical Access Path

Same as Access Path.

Logical Block Number

The number the file processor assigns to each logical block on a disk, starting with 0. See also Logical Structure, Virtual Block.

Logical Structure

A method of making the file processor device-independent. Each disk is considered to be composed of a logically contiguous series of data units called blocks. The disk driver translates the Logical Block Number to a physical location.

Lowest Index Level

The chain of index buckets whose entries point to data buckets. Also called Level 1. See also Data Level, Index Level.

Mass Insert

RMS-11 I/O technique valid for Indexed files only. With Mass Insert, during sequential put operations, 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 gets full, RMS-11 creates a new one, beginning with the next record inserted, and notes its existence in a Primary Level 1 index bucket.

Mass Insert significantly improves file population performance for single-key Indexed files. The percentage of improvement lessens with each additional key defined in the file.

See also Deferred Write.

Maximum Key Value

Same as Highest Possible Key Value.

Maximum Record Size

An attribute for files containing VFC or variable-length records. If Maximum Record Size is not zero, RMS-11 checks each record specified in a put or update operation for size. If the record is longer than the maximum associated with the file, RMS-11 returns the error code ER$RSZ.

The record size for a file of fixed-length records is also stored as the Maximum Record Size attribute.

Maximum Record Number

RMS-11 will not put a record into a Relative file with a relative record number greater than the assigned Maximum Record Number (MRN)--unless MRN is zero. In that case, RMS-11 makes no check on relative record numbers during put operations. MRN is an attribute of Relative files.

MBC

Same as Multi-Block Count.

MBF

Same as Multiple Buffers.

Medium

A file storage device. Relative and Indexed files can be stored only on disks, while Sequential files can be accessed on disks, magnetic tapes, and unit record devices.

Memory-Resident Overlay

An overlay structure or segment maintained in physical memory. The overlay is executed by the task via Active Page Registers. See also Disk-Resident Overlay.

Monitor

In RSTS/E, the part of the operating system that coordinates and controls the other components. See also Executive.

Most Significant Byte

The first byte examined when RMS-11 performs key value comparisons. Key data type determines whether the lowest-addressed byte (string and packed decimal data types) or highest-addressed byte (binary and integer data types) is the most significant. Contrast with Least Significant Byte.

Move Mode

A Record Transfer Mode where RMS-11 always moves a record between the I/O buffer and the user buffer. The user program can manipulate the data only in the user buffer. Contrast with Locate Mode.

MRN

Same as Maximum Record Number.

Multi-Area File

An Indexed file with more than one area. See also Single-Area File.

Multi-Block Count

The number of blocks that RMS-11 moves in and out of the I/O buffer during each I/O operation for a Sequential file. See also I/O unit.

Multiple Buffers

An I/O technique where you assign to a Relative or Indexed file more than the minimum number of I/O buffers required by RMS-11. If the file is not being write-shared, RMS-11 uses the extra buffers to cache buckets in memory.

Multiple Record Access Streams

RMS-11 allows each program to use from one to 255 streams on a Relative or Indexed file.

Next Record

Part of the context of a Record Access Stream. The Next Record is the target of a sequential get not immediately preceded by a find, a find, or a put operation (put operations on Sequential and Relative files only). See also Current Record.

Nonoverlaid

Pertaining to a task that does not include overlay segments. All of a nonoverlaid task must fit in the available virtual address space (64KB) and is brought into memory when the task is run. Contrast with Overlay.

Null Key (Characteristic)

A flag associated with an Alternate Key. The flag indicates whether you defined a null key value for the key. For binary, integer, and packed decimal key data types, the null key value is zero expressed in that data type. For the string key data type, you must specify a byte value from 000(8) through 377(8) including ASCII codes.

During a put or update operation, RMS-11 makes no entry for a record in an Alternate index if:

- The null key flag is set, meaning that a null key value has been defined for the associated Alternate Key field.
- If the associated key field is filled with the defined null key value.

See also Change Key, Duplicate Key.

Object Code

An intermediate form of executable code where external references are not resolved. Assemblers and compilers normally produce modules of object code. Each module consists of relocatable machine code, relocation information, and a symbol table listing the use and definition of symbols within the module.

The Task Builder reads object modules, resolving such references between them and producing an executable task.

Object Module Library

A method of storing object modules so they can be used easily by the Task Builder. Specifically, the RMS-11 object module library named RMSLIB.OLB.

ODL

1. Same as Overlay Description Language. 2. File containing ODL statements. See ODL File.

ODL File

A file containing ODL statements that describes all or part of the overlay structure for a task. ODL files are control files for the Task Builder.

ODS-1

Same as FILES-11 ODS-1.

ODS-2

Same as FILES-11 ODS-2.

Open (File Operation)

RMS-11 initiates access to a file; also occurs as last step in the create file operation. Contrast with Close.

Operating System

The collection of tasks that administer the computer system by scheduling and controlling user and system tasks, performing I/O and various utility functions, and allocating resources for efficient use of the hardware. See also File Processor, Monitor, Executive, Device Driver.

Outfile

A convention for the term "output file specification."

Output File

The destination for records processed by a utility. Contrast with Input File.

Overlay

The technique of repeatedly using the same area of memory during different stages of a program. When one routine is no longer needed in memory, another routine can replace all or part of it.

Overlay Description Language

A set of statements and syntax that the Task Builder can interpret to build the overlay structure for a task.

Overlay Segment

A section of a task treated as a unit that can overlay and be overlaid by other overlay segments.

Overlay Structure

A system of overlays defined by one or more ODL files. The structure consists of a root segment and optionally, one or more overlay segments.

Pack Cluster

See Device Cluster

Packed Decimal (Key)

An Indexed file key data type where a half-byte represents a decimal digit. See also Two-Byte Signed Integer, Four-Byte Signed Integer, String, Two-Byte Unsigned Binary, Four-Byte Unsigned Binary.

Padding

Characters added to data to achieve a required length. In disk Sequential files, RMS-11 adds a null byte (all bits 0) to user data records that are an odd number of bytes long. Additionally, when you do not allow records to span disk blocks, RMS-11 writes null bytes in the gap between the last record in a block and the end of a block. However, on tape, RMS-11 uses circumflexes (^) to pad blocks. See also Word Aligned.

Patch Level

A number specifying the number of times an RMS-11 utility has been patched.

Physical Block

Same as Sector.

Physical I/O Operation

The actual transfer of data between memory and disk, involving the positioning of the disk heads and the electrical flow of data. See also I/O Operation.

Placement Control

The ability to specify a Logical Block, Device Cluster, or Virtual Block Number where (or near where, for VBN) the file processor should begin an area. On VAX/VMS, you can also specify a related file as a guide to placement.

Platter

One of the circular pieces of metal covered with a ferromagnetic substance that make up a disk. Data is stored on platters. See also Bit, Byte, Track, Cylinder, Sector.

Pointer Array

Same as Duplicate Pointer Array.

Population

In this manual, the process of inserting a large burst of records into a file after it has been created and before it is made available for normal processing. You can populate a file with RMSIFL, RMSCNV, or an application task.

Position of Key in Record

The number of the byte(s) where segment(s) of the key begin. Numbering begins with 0; therefore, the first byte in a record is referred to as byte 0. See also Key Field, Segmented Key.

Primary Index

The structure providing the logical access paths for the Primary Key. See also Alternate Index.

Primary Key

The portion of each user data record whose value determines the position of that record within Level 0 of the Primary index. RMS–11 constructs the Primary index from the Primary Key values of records inserted into the file. You must define a Primary Key for each Indexed file. Contrast with Alternate Key. See also Segmented Key.

Primary ODL File

The ODL file referenced by the indirect file you supply to the Task Builder. You can also supply the name of the primary ODL file directly in a Task Builder command line. See also Secondary ODL File.

Primary Root

The single bucket that is the entry point for processes following the Primary index. See also Root.

Program

1. The source file that must be processed by a compiler or assembler and the Task Builder before it can be run. Contrast with Task. 2. That part of a task written by a user, as opposed to the RMS–11 routines in another part of the task.

Prologue

The first blocks of a Relative or Indexed file where RMS–11 maintains file attributes that cannot fit in the file directory.

Prototype ODL File

A heavily commented version of the ODL file for the RMS–11 9KB overlay structure. You can modify this prototype to select only those RMS–11 functions required by your program and to optimize the overlay structure. See also Standard ODL File.

Pyramid

Figurative way of representing an index.

Query Mode

Same as Extended Diagnostic Messages.

Random Access Mode

A mode of record access where you, rather than the organization of the file, establish the order in which records are processed. You must specify a record identifier with each random access. Each record access is independent of the previous record used. Successive operations in the Random Access Mode can identify and access records anywhere in the file. Supported for Relative and Indexed files only. See also Sequential Access Mode, Access by Record's File Address.

Read Access

To open a file to perform find and/or get operations only. Contrast with Write Access.

Read-After-Write

An optional data integrity check performed by the RMSBCK and RMSRST utilities. See also Check-After-Write.

Read Sharing

More than one task opens a file for read access and allows other tasks to perform read-type operations.

Read-Type (Record Operation)

Find or get. See also Write-Type.

Record

A discrete group of data items whose form is repeated throughout the data. Each group represents an entity, and a file consists of a series of such entities. A record is normally the unit of data interchange between a user program and an RMS–11 file.

Record Access Mode

The method used to locate the position of a record within a file for purposes of inserting a record there or retrieving the record already there. See also Access By Record's File Address, Random Access Mode, Sequential Access Mode.

Record Access Stream

A channel between your program and a file through which record operations pass.

Record Format

The shape or form by which RMS–11 recognizes and processes each record in a file. Format dictates how RMS–11 determines a record's size. See also Fixed-Length, Stream, Variable-Length, Variable-with-Fixed-Control, Undefined.

Record Format Overhead

The control bytes added to your data that RMS–11 requires to support a record format. The overhead is zero for fixed-length records and equals two for variable-length and VFC records. See also Bucket Header, Record Header.

Record Header

Seven bytes of control data that RMS–11 appends to every user data record in an Indexed file. This header is converted to an RRV and left in the record's place when the record is moved from its original bucket during a bucket splitting operation; the record assumes a new header in its new location. See also Bucket Header, Record Format Overhead.

Record-Length Field

A unit of data appended to every VFC or variable-length record, regardless of file organization. The field contains the length of the record to which it is attached.

Record Operation

A processing technique involving a Record Access Stream, including:

Connect
Delete
Disconnect
Find
Flush
Get
Put
Rewind
Truncate
Update

Record Processing

Processing using records as the logical units of data interchange. Contrast with Block I/O.

Record Size

The number of bytes in a record. All fixed-length records have the same size, which is stored as the Maximum Record Size file attribute. The size of variable-length and VFC records is indicated by the attached record-length fields. Files containing variable-length and VFC records can have a Maximum Record Size as an attribute.

Record Storage Cell

A fixed-length logical division of a Relative file. Records are stored in cells. However, not all cells must contain records.

Record Transfer Mode

A method of performing record operations indicating whether the user program can manipulate record data in the I/O buffer or in the user buffer. See also Locate Mode, Move Mode.

Record's File Address

A unique identifier for every record within a disk file. This Record's File Address (RFA) remains valid for that record alone for the life of the file. If a record is deleted, its RFA is not reused. See also Access by Record's File Address.

Record's Reference Vector

A copy of a record header that is left in the record's original position when the record is moved during a bucket splitting operation. The RRV preserves Access by Record's File Address and facilitates Alternate Key access.

Relative File

An RMS-11 file created with the Relative file organization.

Relative File Organization

A method of organizing records in a file so that each record is stored in a fixed-length cell and can be accessed randomly. See also Sequential File Organization, Indexed File Organization.

Relative Record Number

Each record in a Relative file is stored in a cell. Each cell can be addressed randomly by a number relative to its distance from the beginning of the file. This number is called the relative record number. For instance, the sixteenth cell has relative record number 16. The cell's number is associated with the record in the cell.

Resident Library

A set of memory-resident routines which can be shared by multiple tasks, but are part of none of them. When a task uses a routine in the Library, the operating system maps the Library segment containing the routine into the task's address space with Active Page Registers.

Restore

1. The process of copying back-up data to recover the contents of a file.
2. RMSRST utility.

Retrieval Pointer

Data associated with a file that specifies blocks on disk. From the structure and content of a retrieval pointer, the file processor can equate virtual blocks to logical blocks. See also Window.

Rewind (Record Operation)

RMS-11 resets the context of a stream to the logical beginning of the file.

RFA

Same as Record's File Address.

RMSBCK

The RMS-11 back-up utility. See also RMSRST.

RMSCNV

The RMS-11 utility that converts one RMS-11 file into another. See also RMSIFL.

RMSDEF

The RMS-11 utility that helps you define an RMS-11 file during an interactive process and then creates that file.

RMSDSP

The RMS-11 utility that prints the attributes and structural data of any RMS-11 file.

RMSIFL

An RMS-11 utility that loads records into an RMS-11 Indexed file. RMSIFL bypasses normal RMS-11 record processing and optimizes the structure of all indexes. See also RMSCNV.

RMSRST

The RMS-11 utility that restores data backed up by RMSBCK.

Root

The highest level in an index. The Root is a single-bucket entry point to the index for random accesses using the associated key.

Root Caching

An I/O technique. You supply virtual address space for more than the two I/O buffers required by RMS-11 for an Indexed file. If the file is not being write shared, RMS-11 uses the extra buffers to cache the Root buckets of indexes it uses for random access operations. See also Multiple Buffers.

RRV

Same as Record Reference Vector.

Secondary Index Data Record

Records occupying Level 0 of each Alternate index. SIDRs contain an Alternate Key value and one or more pointers to user data records in the Primary Level 0 that contain the key value.

Secondary Key

Same as Alternate Key.

Secondary ODL File

An ODL file indirectly referenced by the primary ODL file. Especially, the RMS-11 standard ODL files and your modified version of the RMS-11 prototype ODL file.

Sector

A physical division of a disk, containing 512 bytes on most disks supplied by DIGITAL. See also Bit, Byte, Track, Cylinder.

Segmented Key

A Primary or Alternate string key that consists of separate sections, or segments, in different parts of the record. RMS-11 concatenates the specified segments before it performs any operations that involve key value comparisons. The concatenated version of the key is also stored in index records and SIDRs (for Alternate Keys).

Sequential Access Mode

A mode of record access where the organization of the file establishes the order where records are processed. Each record access depends on the previous record used. Successive operations in the Sequential Access Mode access records in their logical order according to the file organization. Supported for all file organizations. See also Access by Record's File Address, Random Access Mode.

Sequential File

An RMS-11 file created with the Sequential file organization.

Sequential File Organization

A method of organizing records in a file by their order of insertion. The records are virtually contiguous. See also Relative File Organization, Indexed File Organization.

Sequential Write Operations

Put operations in Sequential Access Mode must be performed within file organization restrictions.

Shared Access

More than one task, File Access Block, or Record Access Stream maintain simultaneous access paths to the same file. See also Exclusive Access to a File.

Sharing Specifications In Accessing Programs

When a program opens a file, it must declare the record operations it intends to perform on the file and the type of record operations it will allow other programs to perform on the file.

SIDR

Same as Secondary Index Data Record.

Signed Integer (Key)

See Two-Byte Signed Integer and Four-Byte Signed Integer. See also Packed Decimal, String, Two-Byte Unsigned Binary, Four-Byte Unsigned Binary.

Single-Area File

An Indexed file with one area, whether by default or definition. Sequential and Relative files also consist of one area for purposes of Placement Control. See also Multi-Area File.

Standard File Structure and Interface

RMS-11 operates on most PDP-11 operating systems as well as VAX/VMS. The files it creates are identical and can be used on any system within the confines of on-disk structures. The RMS-11 interface with user programs is also standard and identical on all systems (subject to system-specific limitations). VAX/RMS uses the same disk structure for files as RMS-11 and an interface that parallels RMS-11's.

Standard ODL File

Secondary ODL files provided by DIGITAL to define 8KB, 9KB, and 12KB RMS-11 overlay structures. See also Prototype ODL File.

Stream

Same as Record Access Stream.

Stream (Record Format)

Each record in the file consists of a series of contiguous characters. RMS-11 detects the end of a stream record only by the presence of specific terminators. See also Fixed-Length, Variable-Length, Variable-with-Fixed-Control, Undefined.

String (Key)

An Indexed file key data type where each byte is interpreted by its binary contents. Permissible values are not limited to valid ASCII codes. See also Packed Decimal, Two-Byte Signed Integer, Four-Byte Signed Integer, Two-Byte Unsigned Binary, Four-Byte Unsigned Binary.

Summary Listing

A series of messages that serve as an audit trail of the processing performed by an RMS-11 utility. Normally, a summary listing can be printed on your terminal or written into a file you specify.

Switch

A mnemonic, preceded by a slash (/), that qualifies the processing requested by a command string you supply to a utility.

Synchronous Operations

Any operation where RMS-11 returns control to your program only after all processing associated with the operation is finished. File operations are always synchronous. Record operations can be asynchronous.

System Protection (Code)

A file attribute that dictates how the operating system restricts access to a file based on account number and type of operation.

Target Bucket

The data bucket where RMS–11 determines it should perform the specified operation.

Task

The executable version of a program, especially while running in memory. See also Task Image.

Task Builder

A system utility that converts and combines object modules into a task image. The Task Builder also arranges task segments according to an overlay structure defined in an ODL file.

Task Image

The disk file containing the executable code that comprises the task. When you use RMS–11, a task image is the product of the Task Builder utility. When a task image is executed, it is called a task.

Track

The sectors at a single radius on one disk platter. See also Bit, Byte, Sector, Cylinder.

Truncate (Record Operation)

For Sequential files only, RMS–11 logically deletes the Current Record and all records following it in the file by establishing a new end-of-file position at the first byte of the Current Record.

Two-Byte Signed Integer (Key)

A key data type that can represent the decimal integer values –32,768 through +32,767. See also Four-Byte Signed Integer, Two-Byte Unsigned Binary, Four-Byte Unsigned Binary.

Two-Byte Unsigned Binary (Key)

A key data type that can represent the decimal integer values 0 through +65,535. See also Four-Byte Unsigned Binary, Two-Byte Signed Integer, Four-Byte Signed Integer.

Undefined (Record Format)

No records are defined for the file. RMS–11 reads only blocks, and your program must interpret the contents of each block. Used with Block I/O only. See also Fixed-Length, Stream, Variable-Length, Variable-with-Fixed-Control.

Unit Record Device

A peripheral device capable of handling one record at a time. Terminals and line printers are unit record devices. See also Medium, Forms Control.

Unsigned Binary

See Two-Byte Unsigned Binary and Four-Byte Unsigned Binary. See also Packed Decimal, Two-Byte Signed Integer, Four-Byte Signed Integer, String.

Update (Record Operation)

RMS-11 replaces a record in the I/O buffer with a revised version from the user buffer.

User Buffer

Memory allocated within your program's portion of a task to store one record. This buffer is available to your program and the data in it can be manipulated. RMS-11 can also access this buffer.

User Data Record

The record your program provides in its user buffer plus RMS-11 overhead: the logical unit of data actually stored in a file. See also Index Record.

Utilities

Tasks, provided by DIGITAL, that use RMS-11 to accomplish standard file-related jobs. See also Application Program, Operating System.

Value Match

A criterion used by RMS-11 during a random read-type operation on an Indexed file. Your program must specify whether the key value in a record must be greater than or equal to (or either) the value specified when the operation was initiated.

Variable-Length (Record Format)

The records in the file can be any length, up to the Maximum Record Size associated with the file. For each record, RMS-11 maintains a record-length field specifying the number of data bytes in the record. See also Fixed-Length, Stream, Variable-with-Fixed-Control, Undefined.

Variable-with-Fixed-Control (Record Format)

Each record in the file consists of a fixed control area whose length is the same for all records and a variable area that can vary from zero to the Maximum Record Size associated with the file. See also Fixed-Length, Stream, Variable-Length, Undefined.

Virtual Block (Number)

The file processor treats each file as a device containing virtually contiguous blocks, numbered serially from 1. See also Logical Block.

Virtual-to-Logical-Block Mapping

The file processor translates the Virtual Block Number supplied by RMS–11 to the Logical Block Number the processor must provide to the device driver during a disk access operation. See also Retrieval Pointer, Window.

VFC

Same as Variable-with-Fixed Control.

Window

The set of retrieval pointers the operating system maintains in memory for each open file for mapping Virtual Block Numbers to Logical Block Numbers.

Window Turning

The process of changing the retrieval pointers in memory until the window contains pointers covering the specified virtual blocks.

Word Aligned

Each record in a Sequential file starts and ends on a word boundary; that is, each record is stored as an even number of bytes. RMS–11 uses this convention to maintain structural compatibility with FCS–11 sequential files.

Write Access

To open a file to perform put, update, or delete operations, as well as get or find operations.

Write Sharing

More than one task opens a file for write access and allows other tasks to perform write-type operations.

Write-Type (Record Operation)

Delete, put, or update. See also Read-Type.

XBUF

Same as Extended Buffer Pool.

G

K

L

T

U

V

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 tl
United State

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: Commercial Engineering Publications MK1-2/2H3
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK N.H. 03054

— — — — Do Not Tear - Fold Here and Tape —