EY-3894E-SG-0001

RMS Structures and Utilities on VAX/VMS

Student Guide

Prepared by Educational Services
of
Digital Equipment Corporation

Copyright © 1986, Digital Equipment Corporation.
All Rights Reserved.

The reproduction of this material, in part or whole, is strictly prohibited. For copy information, contact the Educational Services Department, Digital Equipment Corporation, Bedford, Massachusetts 01730.

Book production was done by Educational Services Development and Publishing in Bedford, MA.

Printed in U.S.A.

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 not be used or copied except in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

The following are some of the trademarks of Digital Equipment Corporation.

digital™	DECtalk	Professional
ALL-IN-1	DECUS	RAINBOW
CTS-300	DECwriter	RSTS
DATATRIEVE	DIBOL	RSX
DEC	DSM-11	TOPS-10
DECdirect	FMS-11	TOPS-20
DECmail	KI	UNIBUS
DECmat	KL10	VAX
DECmate	MASSBUS	VAXcluster
DECnet	MicroPower/Pascal	VAXstation VMS
DECservice	MicroVAX II	VT
DECsystem-10	PDP	Work Processor
DECSYSTEM-20	P/OS	

CONTENTS

Course Description x

Course Organization x

Prerequisites x

Course Goals xi

Non-Goals xii

Resources xii

MODULE 1. OVERVIEW OF RMS DATA STRUCTURES AND SERVICES

Components of the VAX/VMS I/O System 1-1

VAX RMS User Control Blocks 1-2

- File Access Block (FAB) 1-4
- Record Access Block (RAB).....1-5
- Extended Attribute Blocks (XABs) 1-6
- Name Block (NAM) 1-8

RMS Naming Conventions 1-9

- Field Names 1-9
- Field Values 1-10

RMS Services/Procedures 1-12

MODULE 2. RMS UTILITIES

Part 1. Introduction 2-1

RMS Utilities 2-1

Creating FDL Files 2-2

- EDIT/FDL Scripts 2-3

Creating Data Files 2-10

Part 2. Evaluating/Utilizing 2-12

FDL -- Invoke Script 2-12
FDL Graphics Output 2-17
FDL -- TOUCHUP Script 2-19

MODULE 3. OVERVIEW OF FILES-11 ON-LINE DISK FILE STRUCTURE

Disk File Structure on VAX/VMS 3-1

The ODS-2 Reserved Files 3-1
Comparing the ODS-1 and ODS-2 Structures 3-2
ODS-2 Directories 3-2
Directory File Structure 3-3

File Characteristics 3-5

ODS-2 File Headers 3-5

Disk Physical Characteristics 3-6

Disk Organization 3-7

MODULE 4. OVERVIEW OF RMS FILE ORGANIZATIONS, RECORD FORMATS, AND ACCESS METHODS

RMS File Organization 4-1

Sequential Files 4-1
Relative Files 4-2
Relative Data Cell and Bucket Format 4-5

Record Format 4-6

Record Access Options 4-7

Random Access to Indexed Files 4-8
Access by Record's File Address 4-9

MODULE 5. INDEXED FILE ORGANIZATION --
INTERNAL STRUCTURE AND OVERHEAD

Part 1. Internals 5-1

Overall Tree Structure and Prolog 5-1

Indexed Sequential Files 5-3
Prolog Description 5-6

Key Descriptors 5-10

Area Descriptors 5-21

Data Bucket Structure 5-27

Data Record Format 5-32

Record Reference Vectors (RRVs) and Bucket Splits 5-34

Simple Bucket Splits 5-35
Multibucket Splits 5-37
Bucket Splits with Duplicate Records 5-38

Key and Data Compression (Prolog 3) 5-39

Primary Key Compression 5-39
Data Compression 5-41

Index Bucket 5-42

Index Record Format 5-43

Index Compression 5-46

Binary Versus Nonbinary Index Search 5-48

Secondary Index Buckets and Data Records (SIDRs) 5-49

Secondary Index Bucket Format 5-49
Secondary Index Data Record Format 5-49
Compression 5-53

Part 2. Simulated Data Example 5-54

MODULE 6. RMS UTILITIES

Part 3. Introduction.....6-1

Analyzing File Structure 6-1
Measuring Run-Time Performance 6-2

Part 4. Evaluating/Utilizing 6-8

MODULE 7. FILE SHARING AND RECORD/BUCKET LOCKING: SEQUENTIAL, RELATIVE, AND INDEXED FILES

File Sharing 7-1

Record Locking 7-4

Automatic Record Locking (RMS Default) 7-5
Manual Record Unlocking 7-6

Alternative Record Locking Controlling Options 7-8

MODULE 8. BUFFER MANAGEMENT: SEQUENTIAL, RELATIVE, AND INDEXED FILES

Interaction of RMS Options with Buffers 8-1

Read-Ahead/Write-Behind 8-2
Asynchronous Option with Read-Ahead/Write-Behind Interaction 8-2
Deferred Write 8-3
Asynchronous I/O 8-5

Local Buffers 8-6

Size and Number of Buffers 8-11

Sequential 8-11
Relative 8-11
Indexed 8-12
Choosing Data Bucket Size for Indexed Files 8-12
Number of Buffers 8-15

Global Buffers and Index Caching 8-17

Single Node 8-18

Read Only 8-18
Not Restricted to Read Only 8-19
Summary -- Global Buffer Performance (as of VAX/VMS Version 4.4) 8-28

VAXclusters 8-29

Lock Value Block 8-30
Performance Recommendations for VAXcluster Global Buffers 8-30

Calculating the Number of Buffers Needed to Cache Index 8-32

MODULE 9. RMS UTILITIES

Part 5. Optimizing and Redesigning Files 9-1

- FDL Optimizing Function 9-2
  Reorganizing Files 9-3

MODULE 10. OPTIMIZING FILE PERFORMANCE: DESIGN AND TUNING SUMMARY

Design -- File Creation Parameters 10-1

Tuning -- Run-Time Parameters 10-2

MODULE 11. PERMANENT FILE ATTRIBUTES VERSUS RUN-TIME FILE CHARACTERISTICS

Permanent File Attributes 11-1

Run-Time File Characteristics 11-3

File Open Options 11-3
Record Connect Options 11-3

Default Settings for RMS Control Blocks for Higher-Level Languages 11-7

FAB Default Settings 11-7
RAB Default Settings 11-11
XAB Defaults 11-14

DCL Commands for Implementing Run-Time Features 11-15

SET FILE 11-15
SHOW RMS_DEFAULT 11-16
SET RMS_DEFAULT 11-17

Specifying Run-Time Options 11-19

RMS Run-Time Options Available Through the FDL ADD Function 11-20

MODULE 12. CALLING RMS SERVICES DIRECTLY FROM MACRO AND HIGHER-LEVEL LANGUAGES

The VAX/VMS Procedure Calling Standard 12-1

Reporting Success or Failure of a Call 12-2

Calling as a Function 12-2
RMS Completion Status Codes 12-2
Testing Completion Status 12-7

Passing Arguments to Procedures 12-15

MODULE 13. ALTERNATE APPROACHES TO ACCESSING RMS CONTROL BLOCKS DIRECTLY -- LANGUAGE EXAMPLES

USEROPEN Function or Regular I/O 13-1

Entry Points to Pascal Utilities 13-25

FDL$PARSE Alternative 13-27

FDL$PARSE Routine 13-28

Fields in RAB Defining User Record Buffers 13-32

VAX Language I/O Operations and RMS Services 13-35

Current Record Context 13-41

MODULE 14. ADVANCED USE OF FILE SPECIFICATIONS

Search Lists and Wildcards 14-1

Defaults or Logical Names 14-1
Search List 14-1
Wildcards 14-1

RMS Default File-Parsing Activities 14-3

RMS File-Parsing Activities Not Done by Default 14-4

LIB$FIND_FILE -- Find File 14-9

MODULE 15. PROCESS QUOTAS AND LIMITS

Process and System Resources for File Applications 15-1

Memory Requirements 15-1
Process Record-Locking Quota 15-1
Other Limits 15-2

MODULE 16. RMS UTILITIES

Part 6. ANALYZE/RMS/INTERACTIVE 16-1

ANALYZE/RMS_FILE Interactive Commands 16-1

Sample Interactive Sessions.....16-4

MODULE 17. DATA RECOVERY FOR CORRUPTED INDEXED FILES

Detecting Problems 17-1

DUMP Utility 17-4

Guidelines for Recovering Data from
Corrupted Indexed Files 17-8

Introduction to the PATCH Utility 17-13

Qualifiers 17-13
Patch Commands 17-14

Data Recovery Examples 17-16

APPENDIX A A-1

APPENDIX B B-1

COURSE DESCRIPTION

This course is designed for application programmers who are responsible for the processing of data files using the Record Management Services (RMS). File optimization strategies are approached from two perspectives, with emphasis placed on indexed file structures.

1. Features that can be implemented at the DCL level

2. Features that can be implemented only within program control.

This course teaches students how to use the RMS utilities and how to call RMS services directly from their programming language (specifically in BASIC, COBOL, FORTRAN, PASCAL or MACRO).

COURSE ORGANIZATION

Length: 5 days

Format: Lecture/Lab (2/3 Lecture, 1/3 Lab)

PREREQUISITES

1. Completion of the VAX/VMS Utilities and Commands course or its equivalent.

2. At least three months of programming experience in one of the following languages: BASIC, COBOL, FORTRAN, PASCAL, or MACRO. This experience should include the use of regular file I/O for the programming language of the user's choice to read and write records to a file, and to update or delete records.

COURSE GOALS

This course is designed to prepare students to perform the following tasks.

- Use the RMS utilities (FDL, CONVERT, ANALYZE) and selected DCL commands (DUMP, SET FILE, SET RMS_DEFAULT).
- Interpret statistical output from ANALYZE/RMS/STATISTICS.
- Tune files on an on-going basis.
- Identify and implement run-time file options that might optimize file performance for a particular application.
- Perform benchmarks on file performance.
- Calculate and set the number of buffers needed for a particular file.
- Identify when global buffers should be enabled for a shared file.
- Access RMS control blocks (FAB, RAB) directly from the programming language of the user's choice (BASIC, COBOL, FORTRAN, PASCAL, or MACRO).
- Call RMS services directly from the programming language of the user's choice (BASIC, COBOL, FORTRAN, PASCAL, or MACRO).
- Enable RMS alternative locking options available within program control that control record locking and unlocking.
- Recover data from corrupted files.

NON-GOALS

This course is not designed for users who must:

- Write programs in VAX languages in which they have no prior experience (covered in the VAX generic language courses).
- Write programs that call system services or Run-Time Library routines (covered in the Utilizing VMS Features from VAX courses).
- Monitor and tune overall system file performance (covered in the Managing Performance on VAX/VMS course).
- Write programs that perform DECnet file operations (covered in the DECnet courses).

RESOURCES

For complete mastery of this course, the following resources from the VAX/VMS documentation set should be available to you.

Guide to VAX/VMS File Applications

VAX Record Management Services Reference Manual

VAX/VMS Analyze/RMS-File Utility Reference Manual

VAX/VMS Convert and Convert/Reclaim Utility Reference Manual

VAX/VMS File Definition Language Facility Reference Manual

MODULE 1
OVERVIEW OF RMS DATA STRUCTURES AND SERVICES

Major Topics

- Components of VAX/VMS I/O system
- VAX RMS user control blocks (FAB, RAB, XAB, NAM)
- RMS naming conventions
- RMS services/procedures

Source

RMS Reference Manual — Sections 1 and 2

MODULE 1
OVERVIEW OF RMS DATA STRUCTURES AND SERVICES

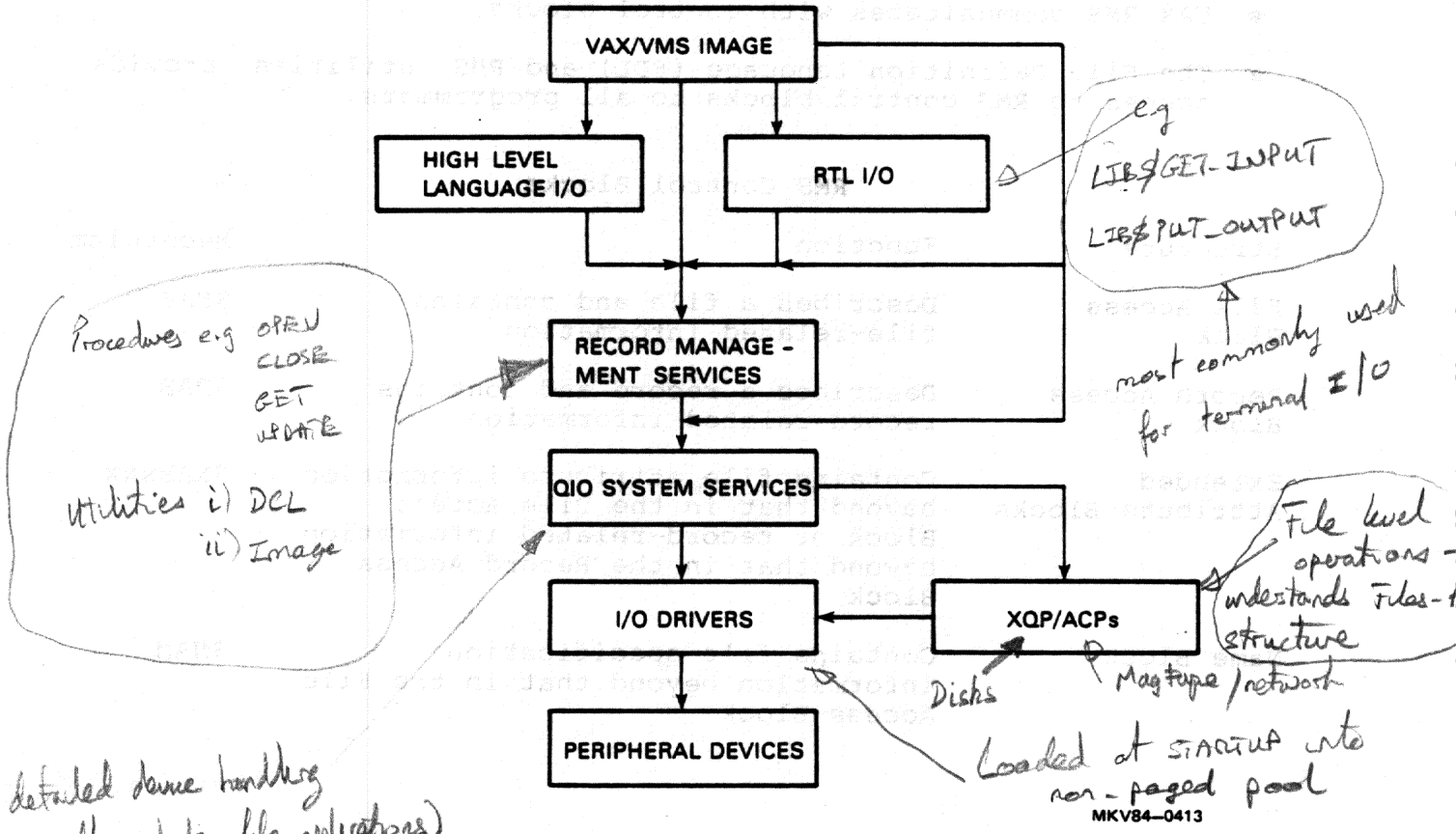
Major Topics

- Components of VAX/VMS I/O system
- VAX RMS user control blocks (FAB, RAB, XAB, NAM)
- RMS naming conventions
- RMS services/procedures

Source

RMS Reference Manual — Sections 1 and 2

COMPONENTS OF THE VAX/VMS I/O SYSTEM



- Extended QIO Procedures (XQPs) used to perform disk functions.
- Ancillary Control Processes (ACPs) used for:

  magnetic tape handling functions
  network functions

- I/O drivers that perform device-level operations.

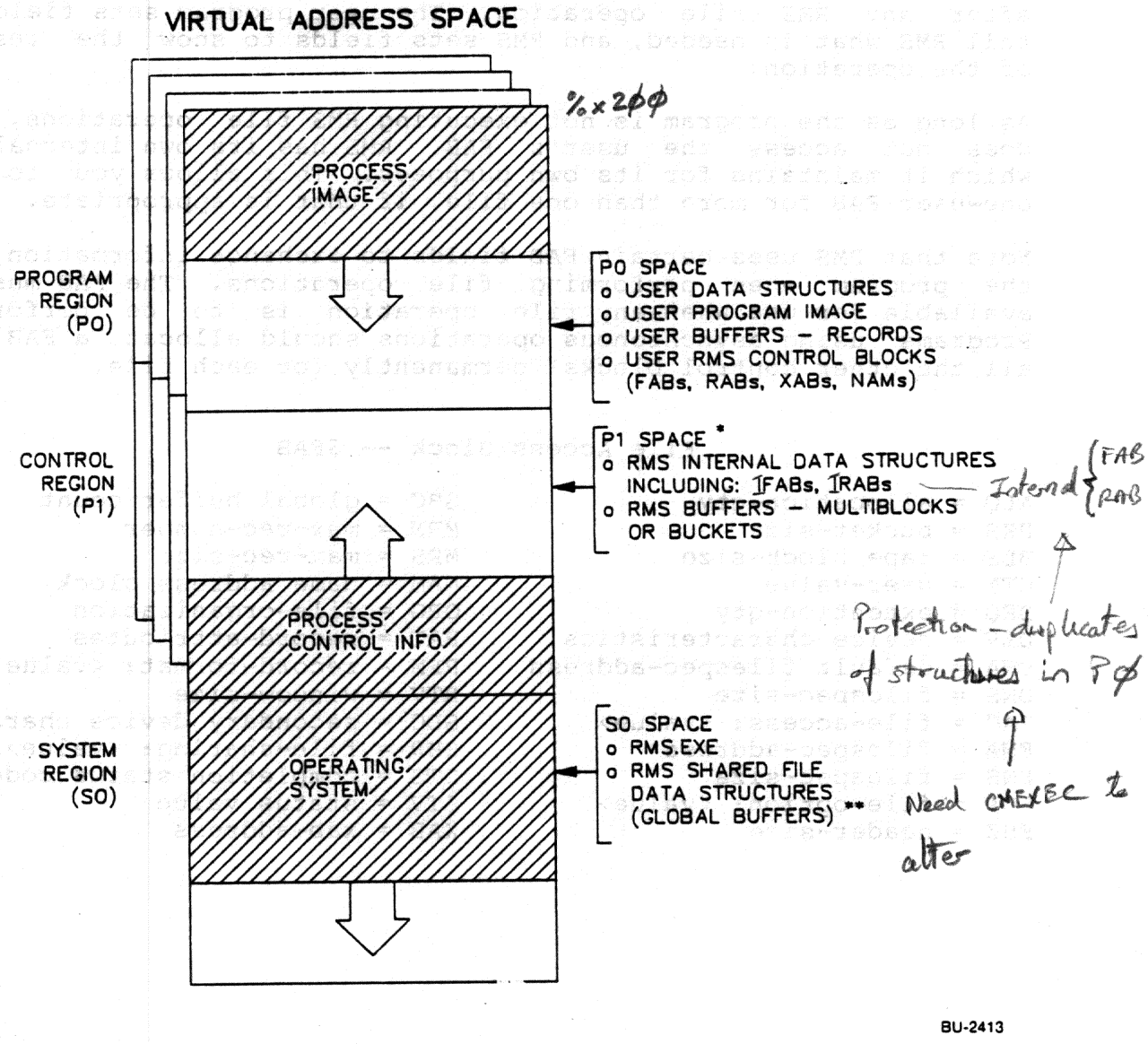
VAX RMS USER CONTROL BLOCKS

- VAX RMS communicates with control blocks.
- The File Definition Language (FDL) and RMS utilities provide access to RMS control blocks to all programmers.

RMS Control Blocks

Structure	Function	Mechanism
File Access Block	Describes a file and contains file-related information	$FAB
Record Access Block	Describes a record and contains record-related information	$RAB
Extended Attribute Blocks	Contains file attribute information beyond that in the File Access Block or record-related information beyond that in the Record Access Block	$XABxxx
Name Block	Contains file specification information beyond that in the File Access Block	$NAM

User Program and RMS Data Structures and Buffers



* Process permanent files have their RMS internal structures in P1. Normal (image) files begin in P1 but can overflow into P0, unless the image was linked using the option IOSEGMENT=NOP0BUFS. This latter option is rarely used.

** Global buffers are page file global sections. They appear to a process that is mapped to them to be in P0 or P1, although they are maintained in S0.

File Access Block (FAB)

The FAB is used for exchanging information with RMS before and after any RMS file operation. The user program sets fields to tell RMS what is needed, and RMS sets fields to show the results of the operation.

As long as the program is not executing RMS file operations, RMS does not access the user's FAB. RMS has its own internal FAB which it maintains for its own purposes. This allows you to use one-user FAB for more than one file, if that is appropriate.

Note that RMS uses certain FAB fields to exchange information with the program when performing file operations. The FAB must be available for use when any file operation is to be performed. Programs using asynchronous operations should allocate a FAB (and all the other control blocks) permanently for each file.

File Access Block -- $FAB

```
ALQ = allocation-qty                GBC = global buffer count
BKS = bucket-size                   MRN = max-rec-number
BLS = tape block-size               MRS = max-rec-size
CTX = user-value                    NAM = name address block
DEQ = extention-qty                 ORG = file-organization
DEV = device characteristics        RAT = record-attributes
DNA = default filespec-address      RFM = record-format: <value>
DNS = filespec-size                 RTV = window-size
FAC = file-access: <value>          SDC = secondary device characteristics
FNA = filespec-address              SHR = file-sharing: <value>
FNS = filespec-size                 STS = completion status code
FOP = file-option: <value>          STV = status value
FSZ = header-size                   XAB = xab-address
```

Record Access Block (RAB)

The Record Access Block (RAB) is used and maintained in the same way as the FAB, except that the RAB is involved in RMS record operations rather than file operations.

The RAB is associated with a record stream, so there could be more than one RAB concurrently associated with the same file. For this reason, the RAB contains a field pointing to the FAB, rather than the other way around (this pointer is for your use, not RMS).

RMS maintains an internal RAB and does not use the user RAB unless a record operation is executing. You will find it difficult to use one RAB for more than one file. There are usually many record operations in the course of a program run. If there is only one RAB, the program will continually need to restore the contents of the RAB from copies that it will have to maintain.

Record Access Block -- $RAB

BKT = bucket-code	RBF = record-buffer-address
CTX = user-value	RFA = record-file-address
FAB = fab-address	RHB = header-buffer-address
KBF = key-buffer-address	ROP = record-options: <value>
KRF = key reference number	RSZ = record-size
KSZ = key-size	STS = completion status code
MBC = multiblock count	STV = completion value
MBF = multibuffer count	TMO = seconds
PBF = prompt-buffer-address	UBF = user-buffer-address
PSZ = prompt-buffer-size	USZ = user-buffer-size
RAC = record-access-mode	XAB = next XAB address

Extended Attribute Blocks (XABs)

The Extended Attribute Blocks (XABs) are a family of related blocks that are linked to the FAB to communicate to VAX RMS any file attributes beyond those described in the FAB.

An XAB can both supersede and supplement the file characteristics specified in the FAB. Each type of XAB has a 6-letter mnemonic name consisting of the prefix XAB followed by three letters that are associated with the function the XAB provides. For instance, the XAB that provides the RMS Create service with file allocation information that supplements and supersedes the file allocation information in the FAB is called an allocation control XAB, or XABALL. Multiple XABs can be used for the same file.

The XABs are described in Chapters 8 through 15 of the VAX Record Management Services Reference Manual. The XABs are generally smaller and simpler than the FAB, RAB, and NAM blocks because each describes information about a single aspect of the file. They are all optional; you use only the ones that you need for any given call to an RMS service routine.

There are seven types of XABs provided by RMS for file operations.

1. Allocation control XAB (XABALL) allows greater control over disk file allocation and positioning during file allocation.

2. Date and time XAB (XABDAT) specifies backup, creation, expiration, and revision date-time values, and also the revision number.

3. File header characteristics XAB (XABFHC) receives the information contained in the header block of a file, which consists of certain file characteristics. This information is restricted to user output.

4. Key definition XAB (XABKEY) defines the key characteristics to be associated with an indexed file.

5. File protection XAB (XABPRO) defines file protection characteristics that specify what class of users or list of users can have certain specified access rights. In the case of an ANSI magnetic tape file with HDR1 labels, XABPRO defines the accessibility field character.

6. Revision date and time XAB (XABRDT) specifies the revision date-time value and revision number to be associated with a file.

7. Summary XAB (XABSUM) receives file characteristics associated with an indexed file, which are not returned by XABFHC. This information is restricted to user output.

The XABs used for any given RMS service call are connected to the FAB in a linked list. The head of the list is the FAB$L_XAB field in the FAB. This field contains the address of the first XAB to be used. Each successive XAB in the list links to the next using the XAB$L_NXT field. This field contains the address of the next XAB in the list.

One XAB type, XABTRM, is associated with the RAB rather than with the FAB. Its purpose is to allow extended control over terminal read operations via RMS, rather than by using the QIO system service.

Name Block (NAM)

The name block (NAM block) supplements the file specification information available in a FAB. A NAM block is useful for opening and locating files, especially if the file specification was entered by a terminal user, or if wildcards or a search list logical name may be present in a file specification, representing multiple files.

There is only one type of NAM block, and usually only one NAM block is associated with each file. To provide an extra level of defaults for a file specification, RMS will apply defaults using additional NAM blocks that contain the file specifications of related files.

RMS NAMING CONVENTIONS

Field Names

RMS uses mnemonic names to identify each field in a control block. For example, the mnemonic name for the field in the FAB that contains the allocation quantity is ALQ.

The mnemonic name (usually three characters in length) serves as a suffix to a symbolic name that identifies the location of each control block field. Use of the supplied symbolic names ensures that you will place values in the correct control block fields. RMS defines each symbolic name as a constant value equal to the offset, in bytes, from the beginning of that control block to the beginning of the field location. These symbolic names are called symbolic offsets. The general format of the symbolic offset is:

```
CCC$X_fff
```

The components of this format are summarized below.

Component	Length	Description
ccc	3 letters	Identifies the type of control block: FAB, NAM, XAB (for all XABs), and RAB
$	1 character	Separator character; always a dollar sign ($)
x	1 letter	Identifies the length of the field: • B for byte • W for word • L for longword • Q for quadword • T for text buffer address
_	1 character	Separator character; always an underscore (_)
fff	3 or more letters	Identifies the mnemonic name of the field, which is used in the VAX MACRO control block macro or higher-level language USEROPEN functions. Some mnemonics contain more than three letters; for example, symbolic offset XAB$B_PROLOG (from XABKEY).

Example

The FAB field whose mnemonic is ALQ has a length of one longword and is identified by the symbolic offset FAB$L_ALQ.

Field Values

Field values involve four different naming conventions.

1. xxx$C_fff

   The first kind of symbolic field values are simple symbolic field values. These are identified by the presence of a C_ immediately following the block prefix in their name. For example, the RAB$B_RAC field has three symbolic values, one each for sequential, keyed, and RFA access modes. The symbolic names for these values are RABC_SEQ, RABC_KEY, and RAB$C_RFA. These symbolic field values are used in simple assignment statements.

   The C symbol is used for any field that can have only one value (a constant).

2. xxx$M_fff

   The second kind of symbolic field value uses mask values to define bit offsets rather than explicit values. These are identified by the presence of M_ immediately following the block prefix in their name. For example, the FAB$L_FOP field is a longword field with the individual bits treated as flags. Each flag has a mask value for specifying a particular file processing option.

FAB$M_CBT	Contiguous 'best try'
FAB$M_CTG	Contiguous
FAB$M_TEF	Truncate at end of file

   The M symbol is used for any fields in which several options may be specified simultaneously. These options are identified by bits within the field.

   The masking value is an integer value that sets the appropriate bit(s).

3. xx.V_fff -- bit offset

4. xxx$S_fff -- size

   The third and fourth kinds of symbolic field values are also used to define flag fields within a larger named field. These are identified by the S_ and V_ values immediately following the block prefix in their names. The S_ form of the name defines the size of that flag field (usually the value 1 for single bit flag fields), and the V_ form defines the bit offset from the beginning of the larger field. These forms can be used with the symbolic bit manipulation functions to set or clear the fields without destroying the other flags.

The V symbol is an alternative to the M symbol to be used for any fields containing options identified by bits.

The RMS Reference Manual identifies field options by the V symbol. However, every V symbol has a corresponding M version.

For most of the FAB, RAB, NAM, and XAB fields that are not supplied using symbolic values, you will need to supply sizes or pointers. For the sizes, you can use ordinary numeric constants or other numeric scalar quantities.

RMS SERVICES/PROCEDURES

RMS services can be called from any VAX language using the VAX Procedure and Condition Handling standard. RMS services are system services and are identified by the entry point prefix SYS$ followed by three or more characters. In the corresponding VAX MACRO macro name, the SYS$ prefix is not used. For example, the Create service has an entry point of SYS$CREATE and a VAX MACRO macro name of $CREATE.

RMS Services

Service Name	Macro Name	Description
File Processing and File Naming		
SYS$CLOSE	$CLOSE	Terminates file processing and disconnects all record streams
SYS$CREATE	$CREATE	Creates and opens a new file of any organization
SYS$DISPLAY	$DISPLAY	Returns the attributes of an open file to the user program
SYS$ENTER*	$ENTER	Enters a file name into a directory
SYS$ERASE	$ERASE	Deletes a file and removes its directory entry
SYS$EXTEND	$EXTEND	Extends the allocated space of a file
SYS$OPEN	$OPEN	Opens an existing file and initiates file processing
SYS$PARSE	$PARSE	Parses a file specification
SYS$REMOVE*	$REMOVE	Removes a file name from a directory
SYS$RENAME	$RENAME	Assigns a new name to (renames) a file
SYS$SEARCH	$SEARCH	Searches a directory, or possibly multiple directories, for a file name

RMS Services (Cont.)

Service Name	Macro Name	Description
Record Processing		
SYS$CONNECT	$CONNECT	Establishes a record stream by associating a RAB with an open file
SYS$DELETE	$DELETE	Deletes a record from a relative or indexed file
SYS$DISCONNECT	$DISCONNECT	Terminates a record stream by disconnecting a RAB from an open file
SYS$FIND	$FIND	Locates and positions to a record and returns its RFA
SYS$FLUSH	$FLUSH	Writes (flushes) modified I/O buffers and file attributes
SYS$FREE	$FREE	Unlocks all records previously locked by the record stream
SYS$GET	$GET	Retrieves a record from a file
SYS$NXTVOL*	$NXTVOL	Causes processing of a magnetic tape file to continue to the next volume of a volume set
SYS$PUT	$PUT	Writes a new record to a file
SYS$RELEASE	$RELEASE	Unlocks a record pointed to by the contents of the RAB$W_RFA field
SYS$REWIND	$REWIND	Positions to the first record of a file
SYS$TRUNCATE	$TRUNCATE	Truncates a sequential file
SYS$UPDATE	$UPDATE	Rewrites (updates) an existing record in a file
SYS$WAIT	$WAIT	Awaits the completion of an asynchronous record operation

* This service is not supported for DECnet operations involving remote file access between two VAX/VMS systems.

RMS Services (Cont.)

Service Name	Macro Name	Description
Block I/O Processing		
SYS$READ	$READ	Retrieves a specified number of bytes from a file, beginning on block boundaries
SYS$SPACE	$SPACE	Positions forward or backward in a file to a block boundary
SYS$WRITE	$WRITE	Writes a specified number of bytes to a file, beginning on block boundaries

RMS Library Routines

Procedure	Operation
FDL$CREATE	Creates a file from an FDL specification and then closes the file.
FDL$GENERATE	Produces an FDL specification by interpreting a set of RMS control blocks. It then writes the FDL specification either to an FDL file or to a character string.
FDL$PARSE	Parses an FDL specification, allocates RMS control blocks, and then fills in the relevant fields.
FDL$RELEASE	Deallocates the virtual memory used by the RMS control blocks created by FDL$PARSE. Use FDL$PARSE to fill in (populate) the control blocks if you plan to release the memory with FDL$RELEASE later.

Procedure	Operation
FDLSCREATE	Creates a file from an FDL specification and then closes the file.
FDLSGENERATE	Produces an FDL specification by interpreting a set of RMS control blocks. It then writes the FDL specification either to an FDL file or to a character string.
FDLSPARSE	Parses an FDL specification, allocates RMS control blocks, and then fills in the relevant fields.
FDLSRELEASE	Deallocates the virtual memory used by the RMS control blocks created by FDLSPARSE. Use FDLSPARSE to fill in (populate) the control blocks if you plan to release the memory with FDLSPRIVATE later.

MODULE 2
RMS UTILITIES

Major Topics

Part 1. Introduction
- RMS Utilities
- Creating FDL Files
- Creating Data Files
  - CREATE/FDL
  - CONVERT/FDL

Part 2. Evaluating/utilizing
- FDL — INVOKE script
- FDL graphics output
- FDL — TOUCHUP script

Source

Guide to VAX/VMS File Applications, Chapter 1 (Section 1.5)
Chapter 4 (Sections 4.1, 4.2, 4.4)

VAX/VMS File Definition Language Facility Reference Manual

VAX/VMS Convert and Convert/Reclaim Utility Reference Manual

MODULE 2
RMS UTILITIES

Major Topics

- Part 1: Introduction
  - RMS Utilities
  - Creating FDL Files
  - Creating Data Files
    - CREATE/FDL
    - CONVERT/FDL

- Part 2: Evaluation/Editing
  - RDL -- INVOKE script
  - RDL graphics output
  - TDL -- TOUCH/"Q" script

Source

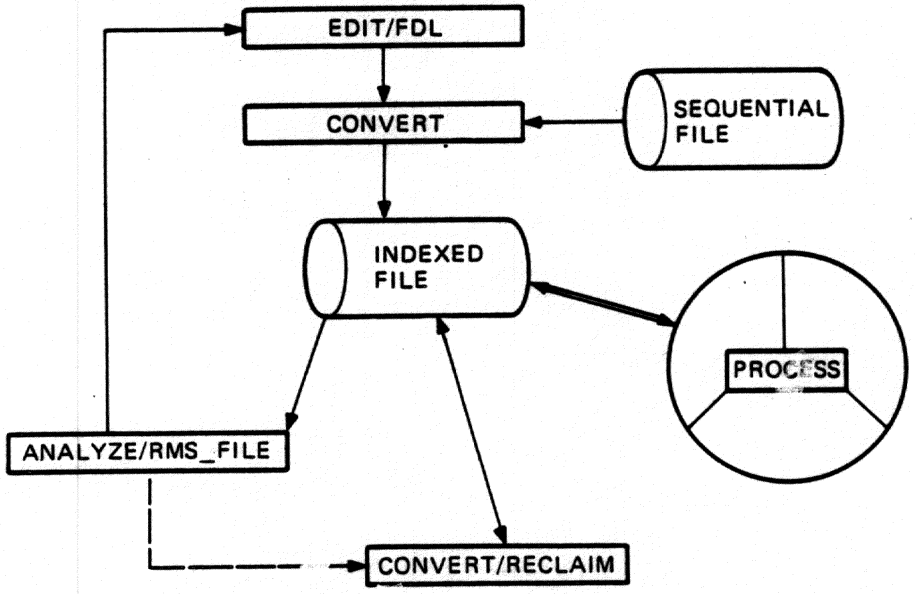
Guide to RMS/VMS File Applications (Chapter 1 (Section 1.5)
Chapter 3 (Sections 3.1, 3.2, 3.4)
RMS File Definition Language Facility Reference Manual
RMS Utilities Guide and Reference (Utility Reference Manual)

PART 1. INTRODUCTION

RMS UTILITIES

- VAX RMS provides the following set of tools to assist in designing and creating data files.
  - EDIT/FDL
  - CREATE/FDL
  - CONVERT/FDL
  - ANALYZE/RMS_FILE/FDL

Tuning Cycle

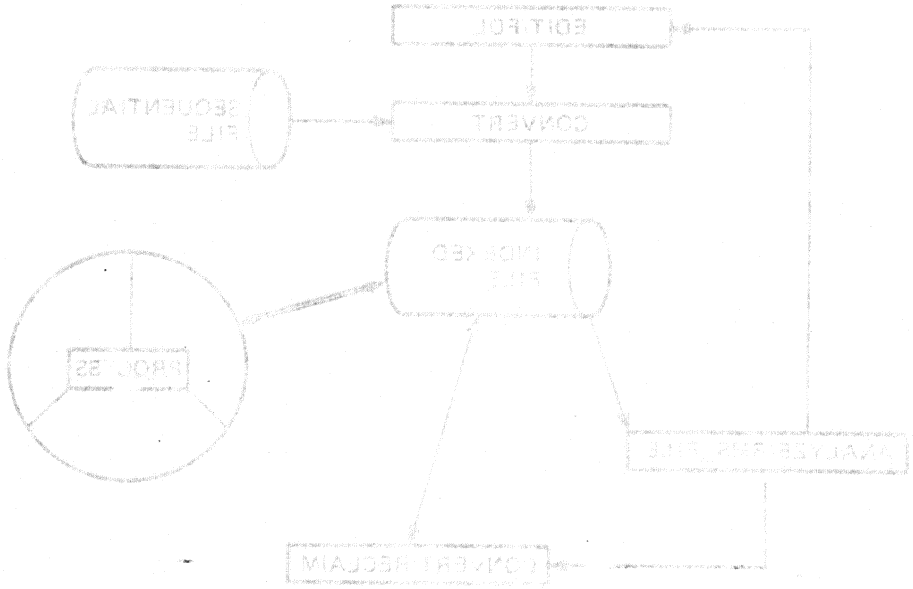


MKV84-1827

CREATING FDL FILES

- The File Definition Language (FDL) editor allows you to:
  - create and modify data file specifications
  - specify all create-time options
  - model data files
  - optimize FDL files
- The specifications are written in the FDL
- To generate an FDL file from an existing data file, type:

  $ ANALYZE/RMS_FILE/FDL file-spec



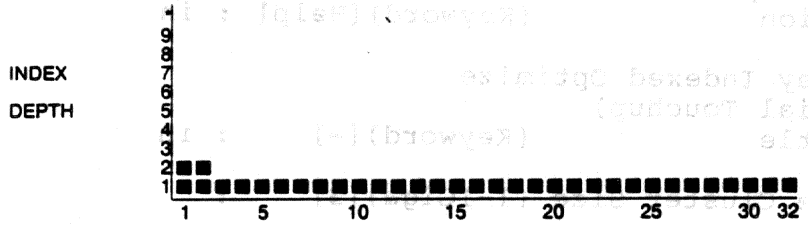
```
                    Parsing Definition File
                    Definition Parse Complete
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                (Keyword)[Help] : ?

                    VAX-11 FDL Editor

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


(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                (Keyword)[Help] : INV

(Add_Key Delete_Key Indexed Optimize
 Relative Sequential Touchup)
Editing Script Title                (Keyword)[-]    :

 You must provide an answer here (or ^Z for Main Menu).
                    Script Title Selection

Add_Key          modeling and addition of a new index's parameters
Delete_Key       removal of the highest index's parameters
Indexed          modeling of parameters for an entire Indexed file
Optimize         tuning of all indices' parameters using file statistics
Relative         selection of parameters for a Relative file
Sequential       selection of parameters for a Sequential file
Touchup          remodeling of parameters for a particular index

(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                (Keyword)[Help] : DE

(Type "?" for a list of existing Primary Attributes)
Enter Desired Primary               (Keyword)[TITLE] : ?
```

```
                    Current Primary Attributes

TITLE
SYSTEM
FILE
RECORD
AREA 0
AREA 1
AREA 2
KEY 0
KEY 1

Enter Desired Primary              (Keyword)[TITLE] : *EXIT*
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function               (Keyword)[Help] : MODIFY

(Type "?" for a list of existing Primary Attributes)
Enter Desired Primary              (Keyword)[TITLE] : ?

                    Current Primary Attributes

TITLE
SYSTEM
FILE
RECORD
AREA 0
AREA 1
AREA 2
KEY 0
KEY 1

Enter Desired Primary              (Keyword)[TITLE] : *EXIT*
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function               (Keyword)[Help] : SET

(Analysis Display Emphasis Granularity
 Number_Keys Output Prompting Responses)
Editor characteristic to set       (keyword)[-]    : ?

                    FDL Editor SET Function

Analysis        filespec of FDL Analysis file
Display         type of graph to display
Emphasis        of default bucket size calculations
Granularity     number of areas in Indexed files
Number_Keys     number of keys in Indexed files
Output          filespec of FDL Output file
Prompting       Full or Brief prompting of menus
Responses       usage of default responses in scripts

Editor characteristic to set       (keyword)[-]    : *EXIT*
```

Example 1. Using the EDIT/FDL Invoke Script

```
$ edit/fdl indxback.fdl

                Parsing Definition File

DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXBACK.FDL; will be created.


 Press RETURN to continue (^Z for Main Menu)


(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function              (Keyword)[Help] : in

(Add_Key Delete_Key Indexed Optimize
 Relative Sequential Touchup)
Editing Script Title              (Keyword)[-]    : in

Target disk volume Cluster Size (1-1Giga)[3]      :

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

(Line Fill Key Record Init Add)
Graph type to display             (Keyword)[Line] :


Number of Records that will be Initially Loaded
into the File                     (0-1Giga)[-]    : 1000

(Fast_Convert NoFast_Convert RMS_Puts)
Initial File Load Method          (Keyword)[Fast] : rms

Will Initial Records Typically be Loaded in Order
by Ascending Primary Key          (Yes/No)[No]    : no

Number of Additional Records to be Added After
the Initial File Load             (0-1Giga)[0]    :

Key 0 Load Fill Percent           (50-100)[100]   :

(Fixed Variable)
Record Format                     (Keyword)[Var]  : fi

Record Size                       (1-32231)[-]    : 50

(Bin2 Bin4 Bin8 Int2 Int4 Int8 Decimal String
Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)
Key  0 Data Type                  (Keyword)[Str]  :
```

```
Key  0 Segmentation desired      (Yes/No)[No]     :

Key  0 Length                   (1-50)[-]        : 5

Key  0 Position                 (0-45)[0]        : 0

Key  0 Duplicates allowed       (Yes/No)[No]     :

File Prolog Version             (0-3)[3]         :

Data Key Compression desired    (Yes/No)[Yes]    : n

Data Record Compression desired (Yes/No)[Yes]    : n
```



PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS FLATTER	(3)
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	5	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	50
LM-LOAD METHOD	RMS_PUTS	IL-INITIAL LOAD	1000	AR-ADDED RECORDS	0

(TYPE "FD" TO FINISH DESIGN)
WHICH FILE PARAMETER (MNEMONIC)[REFRESH] :
WHICH FILE PARAMETER (MNEMONIC)[REFRESH] : FD

BU-2414

```
Text for FDL Title Section      (1-126 chars)[null]
: FDL FOR INDEX BACKWARDS

Data File file-spec             (1-126 chars)[null]
: INDXBACK.DAT

(Carriage_Return FORTRAN None Print)
Carriage Control                (Keyword)[Carr] :

Emphasis Used In Defining Default:      (    Flatter_files    )
Suggested Bucket Sizes:                 (      3      3     12 )
Number of Levels in Index:              (      1      1      1 )
Number of Buckets in Index:             (      1      1      1 )
Pages Required to Cache Index:          (      3      3     12 )
Processing Used to Search Index:        (    112    112    453 )

Key  0 Bucket Size              (1-63)[3]        :


Key  0 Name                     (1-32 chars)[null]
: SEQ_NO
```

only really relevant if index compression is on — figures is 'number of lookups'

```
Global Buffers desired          (Yes/No)[No]    :

The Depth of Key  0 is Estimated to be No Greater
than 1 Index levels, which is 2 Total levels.

 Press RETURN to continue (^Z for Main Menu)

(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function            (Keyword)[Help] :  VIEW


        TITLE   "FDL FOR INDEX BACKWARDS"

        IDENT   "14-JAN-1986 13:14:58   VAX-11 FDL Editor"

        SYSTEM
                SOURCE                  VAX/VMS

        FILE
                NAME                    "INDXBACK.DAT"
                ORGANIZATION            indexed

      RECORD
                CARRIAGE_CONTROL        carriage_return
                FORMAT                  fixed
                SIZE                    50

      AREA 0
                ALLOCATION              177
                BEST_TRY_CONTIGUOUS     yes
                BUCKET_SIZE             3
                EXTENSION               45

      AREA 1
                ALLOCATION              3
                BEST_TRY_CONTIGUOUS     yes
                BUCKET_SIZE             3
                EXTENSION               3
```

```
KEY 0
                CHANGES                     no
                DATA_AREA                   0
                DATA_FILL                   100
                DATA_KEY_COMPRESSION        no
                DATA_RECORD_COMPRESSION     no
                DUPLICATES                  no
                INDEX_AREA                  1
                INDEX_COMPRESSION           no
                INDEX_FILL                  100
                LEVEL1_INDEX_AREA           1
                NAME                        "SEQ_NO"
                PROLOG                      3
                SEG0_LENGTH                 5
                SEG0_POSITION               0
                TYPE                        string
 Press RETURN to continue (^Z for Main Menu)

(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function              (Keyword)[Help] : EXIT

DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXBACK.FDL;1    44 lines
```

```
$ TYPE INDXBACK.FDL

TITLE   "FDL FOR INDEX BACKWARDS"

IDENT   "14-JAN-1986 13:14:58   VAX-11 FDL Editor"

SYSTEM
        SOURCE                          VAX/VMS

FILE
        NAME                            "INDXBACK.DAT"
        ORGANIZATION                    indexed

RECORD
        CARRIAGE_CONTROL                carriage_return
        FORMAT                          fixed
        SIZE                            50

AREA 0
        ALLOCATION                      180
        BEST_TRY_CONTIGUOUS             yes
        BUCKET_SIZE                     3
        EXTENSION                       45

AREA 1
        ALLOCATION                      3
        BEST_TRY_CONTIGUOUS             yes
        BUCKET_SIZE                     3
        EXTENSION                       3

KEY 0
        CHANGES                         no
        DATA_AREA                       0
        DATA_FILL                       100
        DATA_KEY_COMPRESSION            no
        DATA_RECORD_COMPRESSION         no
        DUPLICATES                      no
        INDEX_AREA                      1
        INDEX_COMPRESSION               no
        INDEX_FILL                      100
        LEVEL1_INDEX_AREA               1
        NAME                            "SEQ_NO"
        PROLOG                          3
        SEG0_LENGTH                     5
        SEG0_POSITION                   0
        TYPE                            string
```

CREATING DATA FILES

To create an empty data file from the specifications in an FDL file, type:

```
$ CREATE/FDL=fdl-filespec data-file-spec
```

or

```
FDL$CREATE (fdl-str, [file-spec-str], [default-name],
           [result-name], [FID-bld], [flags],
           [stmnt_num], [retlen], [sts], [stv])
```

Example 2. Creating Data Files

```
$ CREATE/FDL=INDXBACK
$ DIR/FULL INDXBACK.DAT

Directory DISK$INSTRUCTOR:[WOODS.RMS.COURSE]

INDXBACK.DAT;1                File ID:  (32055,21.0)
Size:          184/184        Owner:    [VMS.WOODS]
Created:  15-JAN-1986 10:16   Revised:  15-JAN-1986 10:16 (1)
Expires:   <None specified>   Backup:    <No backup done>
File organization:  Indexed, Prolog: 3, Using 1 key
                            In 2 areas
File attributes:    Allocation: 184, Extend: 45. Maximum bucket size: 3
                    Global buffer count: 0. No version limit
                    Contiguous best try
Record format:      Fixed length 50 byte records
Record attributes:  Carriage return carriage control
File protection:    System:R. Owner:RWED. Group:R. World:
Access Cntrl List:  None

Total of 1 file. 184/184 blocks.


 $ T INDXBACK.DAT

 $     <---- Null file
```

Use the CONVERT/FDL command to transfer data from any organization and format to any other organization and record format.

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

        or

CONV$PASS_FILES  (input-file-spec, output-file-spec,
                  [fdl-file-spec,] [exception-file-spec],
                  [flags])

CONV$PASS_OPTIONS ([parameter-list-address], [flags])

CONV$CONVERT ([status-block-address], [flags])
```

Example 3. Transferring Data Using CONVERT

```
$ CONVERT/FDL=INDXBACK/STATISTICS BACKWARDS.DAT INDXBACK.DAT

       or                  |_____ default SYS$OUTPUT

$ ASSIGN/USER INDXBACK.STAT SYS$OUTPUT
$ REC CONV
$ CONVERT/FDL=INDXBACK/STATISTICS BACKWARDS.DAT INDXBACK.DAT
$ TYPE INDXBACK.STAT

 CONVERT Statistics
Number of Files Processed:       1
Total Records Processed:      1000   Buffered I/O Count:          29
Total Exception Records:         0   Direct I/O Count:           217
Total Valid Records:          1000   Page Faults:                256
Elapsed Time:       0 00:00:12.88   CPU Time:       0 00:00:04.54


$ DIR/SIZE INDXBACK.DAT

Directory DISK$INSTRUCTOR:[WOODS.RMS.COURSE]

INDXBACK.DAT:1          184

Total of 1 file, 184 blocks.
```

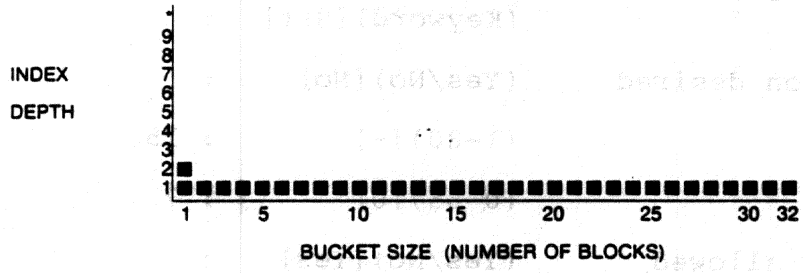
PART 2. EVALUATING/UTILIZING

FDL – Invoke Script

Example 4. Defining Indexed Key Structure Using FDL

```
$ edit/fdl indx1.fdl

                    Parsing Definition File

DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDX1.FDL; will be created.

 Press RETURN to continue (^Z for Main Menu)


(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                (Keyword)[Help] : inv

(Add_Key Delete_Key Indexed Optimize
 Relative Sequential Touchup)
Editing Script Title                (Keyword)[-]    : in

Target disk volume Cluster Size (1-1Giga)[3]        :

Number of Key's to Define           (1-255)[1]      : 2

(Line Fill Key Record Init Add)
Graph type to display               (Keyword)[Line] :


Number of Records that will be Initially Loaded
into the File                       (0-1Giga)[-]    : 500

(Fast_Convert NoFast_Convert RMS_Puts)
Initial File Load Method            (Keyword)[Fast] :

Number of Additional Records to be Added After
the Initial File Load               (0-1Giga)[0]    :

Key 0 Load Fill Percent             (50-100)[100]   :

(Fixed Variable)
Record Format                       (Keyword)[Var]  : f

Record Size                         (1-32231)[-]    : 80

(Bin2 Bin4 Bin8 Int2 Int4 Int8 Decimal String
```

```
Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)
Key  0 Data Type                    (Keyword)[Str]   :

Key  0 Segmentation desired         (Yes/No)[No]     :

Key  0 Length                       (1-80)[-]        : 7

Key  0 Position                     (0-73)[0]        :

Key  0 Duplicates allowed           (Yes/No)[No]     :

File Prolog Version                 (0-3)[3]         :

Data Key Compression desired        (Yes/No)[Yes]    : n

Data Record Compression desired     (Yes/No)[Yes]    : n

Index Compression desired           (Yes/No)[Yes]    : n

(Type "FD" to Finish Design)
```



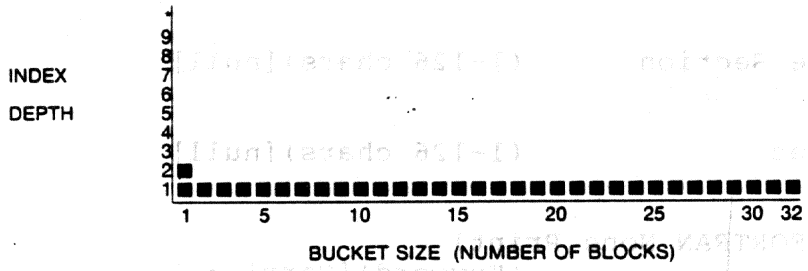
PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS FLATTER	(3)
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	7	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	80
LM-LOAD METHOD	FAST_CONV	IL-INITIAL LOAD	500	AR-ADDED RECORDS	0

(TYPE "FD" TO FINISH DESIGN)
WHICH FILE PARAMETER (MNEMONIC)[REFRESH] : FD

BU-2415

```
Text for FDL Title Section          (1-126 chars)[null]
: fdl for index1

Data File file-spec                 (1-126 chars)[null]
: indx1.dat

(Carriage_Return FORTRAN None Print)
Carriage Control                    (Keyword)[Carr] :

Emphasis Used In Defining Default:  (    Flatter_files    )
Suggested Bucket Sizes:             (      3      3     12 )
Number of Levels in Index:          (      1      1      1 )
Number of Buckets in Index:         (      1      1      1 )
Pages Required to Cache Index:      (      3      3     12 )
Processing Used to Search Index:    (    115    115    463 )
```

```
Key  0 Bucket Size                  (1-63)[3]        :

Key  0 Name                         (1-32 chars)[null]
: seq_no

Global Buffers desired              (Yes/No)[No]     :


The Depth of Key  0 is Estimated to be No Greater
than 1 Index levels, which is 2 Total levels.

 Press RETURN to continue (^Z for Main Menu)

(Line Fill Key)
Graph type to display               (Keyword)[Line]  :


Key  1 Load Fill Percent            (50-100)[100]    :

(Bin2 Bin4 Bin8 Int2 Int4 Int8 Decimal String
Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)
Key  1 Data Type                    (Keyword)[Str]   :

Key  1 Segmentation desired         (Yes/No)[No]     :

Key  1 Length                       (1-80)[-]        : 15

Key  1 Position                     (0-65)[0]        : 7

Key  1 Duplicates allowed           (Yes/No)[Yes]    :

Data Key Compression desired        (Yes/No)[Yes]    : n

Index Compression desired           (Yes/No)[Yes]    : n

 (Type "FD" to Finish Design)
```



PV-PROLOG VERSION	3	KT-KEY 1 TYPE	STRING	EM-EMPHASIS FLATTER	(3)
DK-DUP KEY 0 VALUES	YES	KL-KEY 1 LENGTH	15	KP-KEY 1 POSITION	7
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	80
LM-LOAD METHOD	FAST_CONV	IL-INITIAL LOAD	500	AR-ADDED RECORDS	0

(TYPE "FD" TO FINISH DESIGN)
WHICH FILE PARAMETER (MNEMONIC)[REFRESH] :
WHICH FILE PARAMETER (MNEMONIC)[REFRESH] : FD

BU-2416

```
Emphasis Used In Defining Default:      (     Flatter_files    )
Suggested Bucket Sizes:                 (       3      3     12 )
Number of Levels in Index:              (       1      1      1 )
Number of Buckets in Index:             (       1      1      1 )
Pages Required to Cache Index:          (       3      3     12 )
Processing Used to Search Index:        (      66     66    268 )

Key  1 Bucket Size                 (1-63)[3]        :

Key  1 Changes allowed             (Yes/No)[Yes]    :

Key  1 Name                        (1-32 chars)[null]
: last_name


The Depth of Key  1 is Estimated to be No Greater
than 1 Index levels, which is 2 Total levels.

 Press RETURN to continue (^Z for Main Menu)


        (Add Delete Exit Help Invoke Modify Quit Set View)
        Main Editor Function              (Keyword)[Help] : view


        TITLE   "fdl for indexl"

        IDENT   "14-JAN-1986 13:35:55   VAX-11 FDL Editor"

        SYSTEM
                SOURCE                  VAX/VMS

        FILE
                NAME                    "indexl.dat"
                ORGANIZATION            indexed

        RECORD
                CARRIAGE_CONTROL        carriage_return
                FORMAT                  fixed
                SIZE                    80

        AREA 0
                ALLOCATION              90
                BEST_TRY_CONTIGUOUS     yes
                BUCKET_SIZE             3
                EXTENSION               24

        AREA 1
                ALLOCATION              3
                BEST_TRY_CONTIGUOUS     yes
                BUCKET_SIZE             3
                EXTENSION               3
```

```
AREA 2
        ALLOCATION                  27
        BEST_TRY_CONTIGUOUS         yes
        BUCKET_SIZE                 3
        EXTENSION                   6

AREA 3
        ALLOCATION                  3
        BEST_TRY_CONTIGUOUS         yes
        BUCKET_SIZE                 3
        EXTENSION                   3

KEY 0
        CHANGES                     no
        DATA_AREA                   0
        DATA_FILL                   100
        DATA_KEY_COMPRESSION        no
        DATA_RECORD_COMPRESSION     no
        DUPLICATES                  no
        INDEX_AREA                  1
        INDEX_COMPRESSION           no
        INDEX_FILL                  100
        LEVEL1_INDEX_AREA           1
        NAME                        "seq_no"
        PROLOG                      3
        SEG0_LENGTH                 7
        SEG0_POSITION               0
        TYPE                        string

KEY 1
        CHANGES                     no
        DATA_AREA                   2
        DATA_FILL                   100
        DATA_KEY_COMPRESSION        no
        DUPLICATES                  yes
        INDEX_AREA                  3
        INDEX_COMPRESSION           no
        INDEX_FILL                  100
        LEVEL1_INDEX_AREA           3
        NAME                        "last_name"
        SEG0_LENGTH                 15
        SEG0_POSITION               7
        TYPE                        string

     Press RETURN to continue (^Z for Main Menu)


(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function              (Keyword)[Help] : exit

DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDEX1.FDL;1    65 lines
```

FDL Graphics Output

FDL has two graphics modes:

1. Line

   bucket size versus index depth

2. Surface

   bucket size versus load fill percent versus index depth

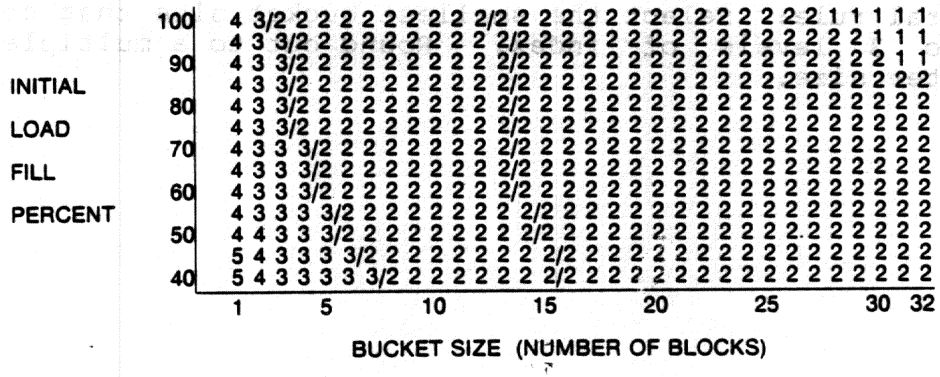
   bucket size versus key length versus index depth

   bucket size versus record size versus index depth

   bucket size versus initial load record count versus index depth

   bucket size versus additional record count versus index depth

A Surface_Plot Graph

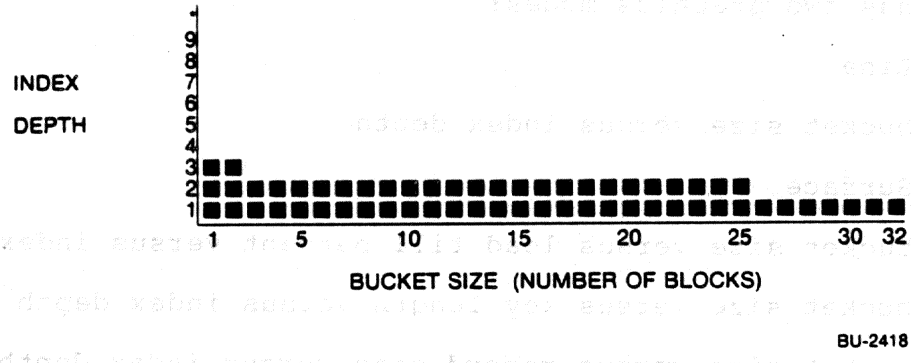


BU-2417

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

The area on the graph within the slash marks represents combinations that RMS will find acceptable. A fill factor of 70% and a bucket size of 10 blocks is a good combination. However, a fill factor of 70% and a bucket size of 15 blocks is poor because it falls outside of the slash boundaries.

A Line_Plot Graph



BU-2418

Evaluating Line Plot

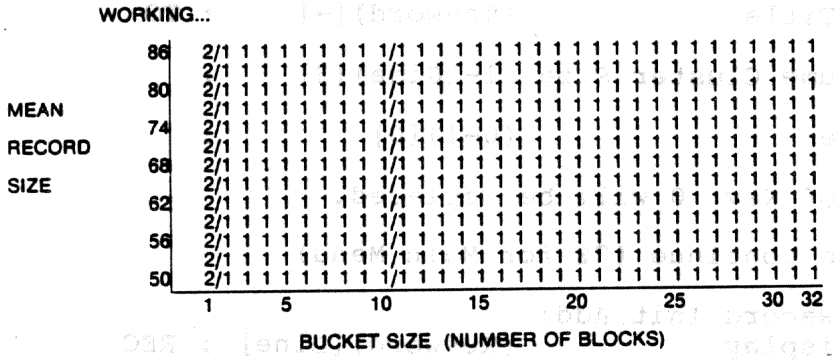
Breakpoints	# Blocks/Bucket
3 to 2	3
2 to 1	~~25~~ 26

General rule: Select the smallest bucket size that corresponds to 2 to 3 levels of index. Round out to a multiple of the disk cluster size.

Example 5. Utilizing EDIT/FDL Touchup to Obtain Surface Graphics Output

```
$ EDIT/FDL INDX1.FDL

                Parsing Definition File
                Definition Parse Complete
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function              (Keyword)[Help] : INV

(Add_Key Delete_Key Indexed Optimize
 Relative Sequential Touchup)
Editing Script Title              (Keyword)[-]    : TO

Target disk volume Cluster Size (1-1Giga)[3]      :

Key of Reference                  (0-1)[0]        :

The Definition of Key  0 will be replaced.

 Press RETURN to continue (^Z for Main Menu)

(Line Fill Key Record Init Add)
Graph type to display             (Keyword)[Line] : REC


Number of Records that will be Initially Loaded
into the File                     (0-1Giga)[-]    : 500

(Fast_Convert NoFast_Convert RMS_Puts)
Initial File Load Method          (Keyword)[Fast] :

Number of Additional Records to be Added After
the Initial File Load             (0-1Giga)[0]    :

Key 0 Load Fill Percent           (50-100)[100]   :

(Fixed Variable)
Record Format                     (Keyword)[Var]  : VAR

Low bound: Record Size            (1-32229)[-]    : 50

High bound: Record Size           (50-32229)[1000]          : 80

(Bin2 Bin4 Bin8 Int2 Int4 Int8 Decimal String
Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)

Key  0 Data Type                  (Keyword)[Str]  :

Key  0 Segmentation desired       (Yes/No)[No]    :
```

```
Key  0 Length                          (1-86)[-]          : 7

Key  0 Position                        (0-79)[0]          :

Key  0 Duplicates allowed              (Yes/No)[No]       :


File Prolog Version                    (0-3)[3]           :

Data Key Compression desired           (Yes/No)[Yes]      : N

Data Record Compression desired        (Yes/No)[Yes]      : N

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



PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS	FLATTER
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	7	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	VARIABLE		
LM-LOAD METHOD	FAST_CONV	IL-INITIAL LOAD	500	AR-ADDED RECORDS	0

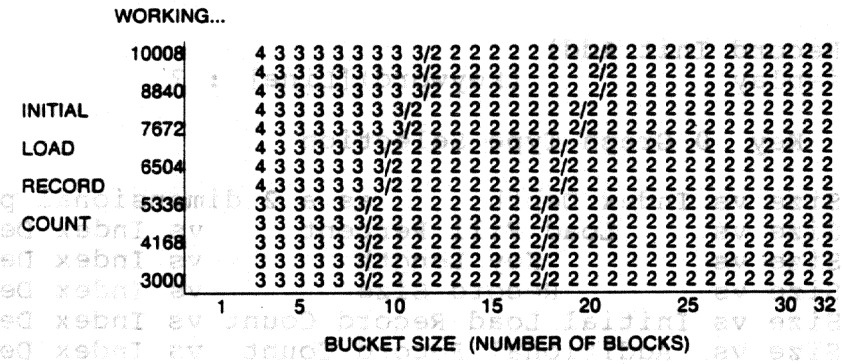
BU-2419

```
(Type "FD" to Finish Design
Which File Parameter     (Mnemonic)[refresh]      : FD

Text for FDL Title Section          (1-126 chars)[null]
: INDEX1 NEW GRPHICS

Data File file-spec                 (1-126 chars)[null]
: INDEX1.DAT

(Carriage_Return FORTRAN None Print)
Carriage Control                    (Keyword)[Carr] :

Mean Record Size                    (1-32229)[-]     : *EXIT*
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                (Keyword)[Help] : QUIT
```

Example 6. FDL Session Utilizing TOUCHUP to Produce Different Graphics Output

```
$ EDIT/FDL FOO

                Parsing Definition File

DISK$INSTRUCTOR:[WOODS.RMS.COURSE]FOO.FDL; will be created.
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function              (Keyword)[Help] : INV

(Add_Key Delete_Key Indexed Optimize
 Relative Sequential Touchup)
Editing Script Title              (Keyword)[-]    : IND

Target disk volume Cluster Size (1-1Giga)[3]      :

Number of Keys to Define          (1-255)[1]      : 2

(Line Fill Key Record Init Add)
Graph type to display             (Keyword)[Line] : ?

                 Key  0 Graph Type Selection

Line    Bucket Size vs Index Depth      as a 2 dimensional plot
Fill    Bucket Size vs     Load Fill Percent     vs Index Depth
Key     Bucket Size vs         Key Length        vs Index Depth
Record  Bucket Size vs        Record Size        vs Index Depth
Init    Bucket Size vs Initial Load Record Count vs Index Depth
Add     Bucket Size vs  Additional Record Count  vs Index Depth

Graph type to display             (Keyword)[Line] : INIT

Low bound: Initial Load of Recs (0-1Giga)[0]      : 3000

High bound: Initial Load of Recs(3000-1Giga)[150000] : 10000

Number of Additional Records to be Added After
the Initial File Load             (0-1Giga)[0]    : 4000

Will Additional Records Typically be Added in
Order by Ascending Primary Key  (Yes/No)[No]      :

Will Added Records be Distributed Evenly over the
Initial Range of Pri Key Values (Yes/No)[No]      :

(Fixed Variable)
Record Format                     (Keyword)[Var]  : FI

Record Size                       (1-32231)[-]    : 1400

(Bin2 Bin4 Bin8 Int2 Int4 Int8 Decimal String
Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)

Key  0 Data Type                  (Keyword)[Str]  :
```

```
Key  0 Segmentation desired          (Yes/No)[No]     :

Key  0 Length                       (1-255)[-]       : 50

Key  0 Position                     (0-1340)[0]      :

Key  0 Duplicates allowed           (Yes/No)[No]     :

File Prolog Version                 (0-3)[3]         :

Data Key Compression desired        (Yes/No)[Yes]    :

Data Record Compression desired     (Yes/No)[Yes]    :

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

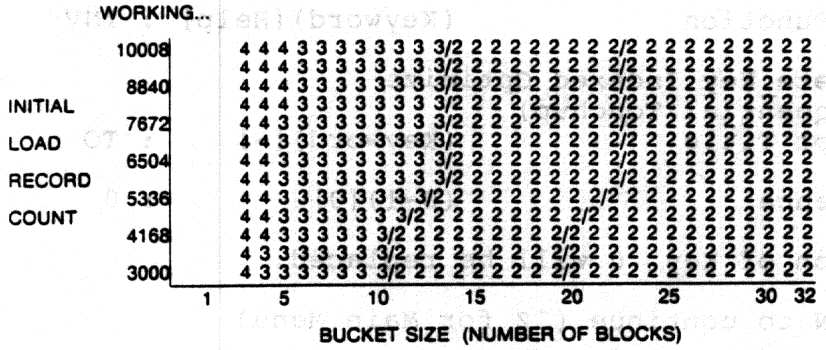


PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS	FLATTER
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	50	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	1400
LM-LOAD METHOD	RMS_PUTS	AR-ADDED RECORDS	4000		

(TYPE "FD" TO FINISH DESIGN)
WHICH FILE PARAMETER (MNEMONIC)[REFRESH] :KL

KEY 0 LENGTH (1-255)[-] :75

BU-2420



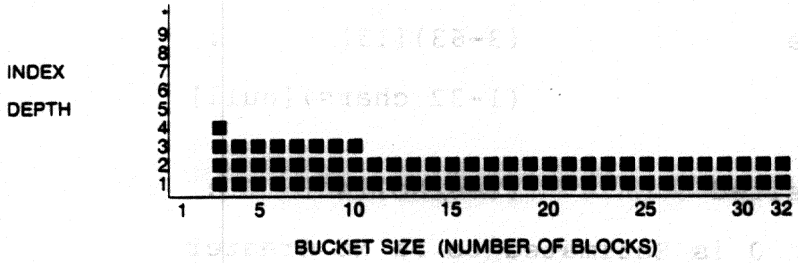
PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS	FLATTER
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	75	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	1400
LM-LOAD METHOD	RMS_PUTS	AR-ADDED RECORDS	4000		

BU-2421

```
(Type "FD" to Finish Design
Which File Parameter    (Mnemonic)[refresh]      : FD

Text for FDL Title Section       (1-126 chars)[null]
: SAMPLE FDL SESSION

Data File file-spec              (1-126 chars)[null]
: FOO.DAT

(Carriage_Return FORTRAN None Print)
Carriage Control                 (Keyword)[Carr] :

Number of Records that will be Initially Loaded
into the file                    (0-1 Giga)[-]:  8000

Emphasis Used In Defining Default:      (    Flatter_files    )
Suggested Bucket Sizes:                 (      6     15     24 )
Number of Levels in Index:              (      3      2      2 )
Number of Buckets in Index:             (    502     67     26 )
Pages Required to Cache Index:          (   3012   1005    624 )
Processing Used to Search Index:        (     75    126    200 )

Key  0 Bucket Size               (1-63)[15]      :

Key  0 Name                      (1-32 chars)[null]
: NONSENSE

Global Buffers desired           (Yes/No)[No]    :

The Depth of Key  0 is Estimated to be No Greater
than 2 Index levels, which is 3 Total levels.

 Press RETURN to continue (^Z for Main Menu)

(Add Delete Exit Help Invoke Modify Quit Set View)
```

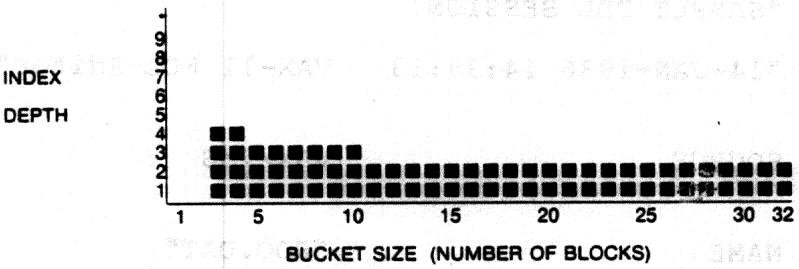
```
Main Editor Function                (Keyword)[Help] : INV

(Add_Key Delete_Key Indexed Optimize
 Relative Sequential Touchup)
Editing Script Title                (Keyword)[-]    : TO

Key of Reference                    (0-1)[0]        : 0

The Definition of Key  0 will be replaced.

 Press RETURN to continue (^Z for Main Menu)

(Line Fill Key Record Init Add)
Graph type to display               (Keyword)[Key]  : LINE

Number of Records that will be Initially Loaded
into the File                       (0-1Giga)[-]    : 8000

(Fast_Convert NoFast_Convert RMS_Puts)
Initial File Load Method            (Keyword)[Fast] : RMS

Will Initial Records Typically be Loaded in Order
by Ascending Primary Key            (Yes/No)[No]    :

Number of Additional Records to be Added After
the Initial File Load               (0-1Giga)[0]    : 4000

Will Additional Records Typical be Added in
Order by Ascending Primary Key      (Yes/No)[No]    :

Will Added Records be Distributed Evenly over the
Initial Range of Pri Key Values     (Yes/No)[No]    :

Key 0 Load Fill Percent             (50-100)[100]   :

(Fixed Variable)
Record Format                       (Keyword)[Var]  : F

Record Size                         (1-32151)[-]    : 1400

(Bin2 Bin4 Bin8 Int2 Int4 In 8 Decimal String
Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)

Key  0 Data Type                    (Keyword)[Str]  :

Key  0 Segmentation desired         (Yes/No)[No]    :

Key  0 Length                       (1-255)[-]      : 50

Key  0 Position                     (0-1350)[0]     :

Key  0 Duplicates allowed           (Yes/No)[No]    :

File Prolog Version                 (0-3)[3]        :
```

```
Data Key Compression desired      (Yes/No)[Yes]    :

Data Record Compression desired (Yes/No)[Yes]    :

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



PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS FLATTER	(12)
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	50	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	1400
LM-LOAD METHOD	RMS_PUTS	IL-INITIAL LOAD	8000	AR-ADDED RECORDS	4000
(TYPE "FD" TO FINISH DESIGN)					
WHICH FILE PARAMETER		(MNEMONIC)[REFRESH]	:KL		
KEY 0 LENGTH		(1-255)[-]	:75		

BU-2422



PV-PROLOG VERSION	3	KT-KEY 0 TYPE	STRING	EM-EMPHASIS FLATTER	(15)
DK-DUP KEY 0 VALUES	NO	KL-KEY 0 LENGTH	75	KP-KEY 0 POSITION	0
RC-DATA RECORD COMP	0%	KC-DATA KEY COMP	0%	IC-INDEX RECORD COMP	0%
BF-BUCKET FILL	100%	RF-RECORD FORMAT	FIXED	RS-RECORD SIZE	1400
LM-LOAD METHOD	RMS_PUTS	IL-INITIAL LOAD	8000	AR-ADDED RECORDS	4000
(TYPE "FD" TO FINISH DESIGN)					
WHICH FILE PARAMETER		(MNEMONIC)[REFRESH]	:FD		

BU-2423

```
Text for FDL Title Section        (1-126 chars)[null]
: SAMPLE FDL SESSION

Data File file-spec               (1-126 chars)[null]
: FOO.DAT

(Carriage_Return FORTRAN None Print)
Carriage Control                  (Keyword)[Carr] :
```

```
Emphasis Used In Defining Default:      (    Flatter_files    )
Suggested Bucket Sizes:                 (      6     15     24 )
Number of Levels in Index:              (      3      2      2 )
Number of Buckets in Index:             (    502     67     26 )
Pages Required to Cache Index:          (   3012   1005    624 )
Processing Used to Search Index:        (     75    126    200 )

Key  0 Bucket Size                 (3-63)[15]       :

Key  0 Name                        (1-32 chars)[null]
: NONSENSE

Global Buffers desired             (Yes/No)[No]     :

 The Depth of Key  0 is Estimated to be No Greater
than 2 Index levels, which is 3 Total levels.

 Press RETURN to continue (^Z for Main Menu)


        (Add Delete Exit Help Invoke Modify Quit Set View)
        Main Editor Function              (Keyword)[Help] : VIEW


        TITLE    "SAMPLE FDL SESSION"

        IDENT    "14-JAN-1986 14:38:13   VAX-11 FDL Editor"

        SYSTEM
                 SOURCE                  VAX/VMS

        FILE
                 NAME                    "FOO.DAT"
                 ORGANIZATION            indexed

        RECORD
                 CARRIAGE_CONTROL        carriage_return
                 FORMAT                  fixed
                 SIZE                    1400

        AREA 0
                 ALLOCATION              46695
                 BEST_TRY_CONTIGUOUS     yes
                 BUCKET_SIZE             15
                 EXTENSION               11685

        AREA 1
                 ALLOCATION              795
                 BEST_TRY_CONTIGUOUS     yes
                 BUCKET_SIZE             15
                 EXTENSION               210
```

```
AREA 2
        ALLOCATION              2442
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             6
        EXTENSION               612

AREA 3
        ALLOCATION              114
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             6
        EXTENSION               30

KEY 0
        CHANGES                 no
        DATA_AREA               0
        DATA_FILL               100
        DATA_KEY_COMPRESSION    yes
        DATA_RECORD_COMPRESSION yes
        DUPLICATES              no
        INDEX_AREA              1
        INDEX_COMPRESSION       yes
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       1
        NAME                    "NONSENSE"
        PROLOG                  3
        SEG0_LENGTH             75
        SEG0_POSITION           0
        TYPE                    string

KEY 1
        CHANGES                 no
        DATA_AREA               2
        DATA_FILL               100
        DATA_KEY_COMPRESSION    yes
        DUPLICATES              yes
        INDEX_AREA              3
        INDEX_COMPRESSION       yes
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       3
        NAME                    "DUMMY"
        SEG0_LENGTH             75
        SEG0_POSITION           100
        TYPE                    string

(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function            (Keyword)[Help] : EXIT

DISK$INSTRUCTOR:[WOODS.RMS.COURSE]FOO.FDL;1   65 lines
$
```

MODULE 3
OVERVIEW OF FILES-11
ON-LINE DISK FILE STRUCTURE

Major Topics

- Disk file structure on VAX/VMS
- File characteristics
- Disk physical characteristics
- Disk organization

Source

Guide to VAX/VMS File Applications — Chapter 1 (Sections 1.1 and 1.2)

MODULE 3
OVERVIEW OF FILES-11
ON-LINE DISK FILE STRUCTURE

DISK FILE STRUCTURE ON VAX/VMS

- The VMS default disk file structure is Files-11 Structure Level 2, also called On-Disk Structure 2 (ODS-2).
- Used by XQPs to maintain and control data on disk volumes.

The ODS-2 Reserved Files

- Define the Files-11 disk file structure
- Are created when a volume is initialized
- Are cataloged in the Master File Directory [0,0] of the volume

Volume Information Contained in ODS-2 Reserved Files

Reserved File	Information
Index File (INDEXF.SYS)	Bootstrap block, home block, back-up homr block, back-up index file header, index file bit map, file headers
Storage Bit Map File (BITMAP.SYS)	Location of available clusters on a volume
Bad Block File (BADBLK.SYS)	Areas of volume not suitable for use
Pending Bad Block Log File (BADLOG.SYS)	Areas of volume suspected of being unsuitable for use
Master File Directory (000000.DIR)	Pointers to all User File Directories
Core Image File (CORIMG.SYS)	Not used by VMS. Provided to preserve structure of previous versions
Volume Set List File (VOLSET.SYS)	Labels of other volumes in the volume set (if more than one)
Continuation File (CONTIN.SYS)	The extension file identifier, if a file crosses from one volume to another in a loosely coupled volume set
Backup Log File (BACKUP.SYS)	History of backups on the volume

Comparing the ODS-1 and ODS-2 Structures

ODS-1 and ODS-2 Comparison

Characteristics	ODS-1	ODS-2
Transportable to RSX/IAS	Yes	No
Subdirectories	No	Yes
Alphanumeric directory names	No	Yes
Alphabetized directory files	No	Yes
Clustered allocation	No	Yes
Tightly coupled volume sets	No	Yes
Backup home block	No	Yes
Extended map pointers	No	Yes
Meaning of protection code E	Extend	Execute

ODS-2 Directories

- Directories associate symbolic file name with file ID.
- User File Directories (UFDs) are entered in the Master File Directory (MFD).
- Subfile directories (SFDs) are entered in their parent directory file.
- Directory names may be in the following formats.
  - UIC forma with the group and member fields in the range 0 to 377 oct 1
  - Alphanumeric format of not more than eight characters
- Subdirectory names must be alphanumeric.

62 file entries per directory file disk block

Once a directory file is > 128 blocks it is no longer cached in memory

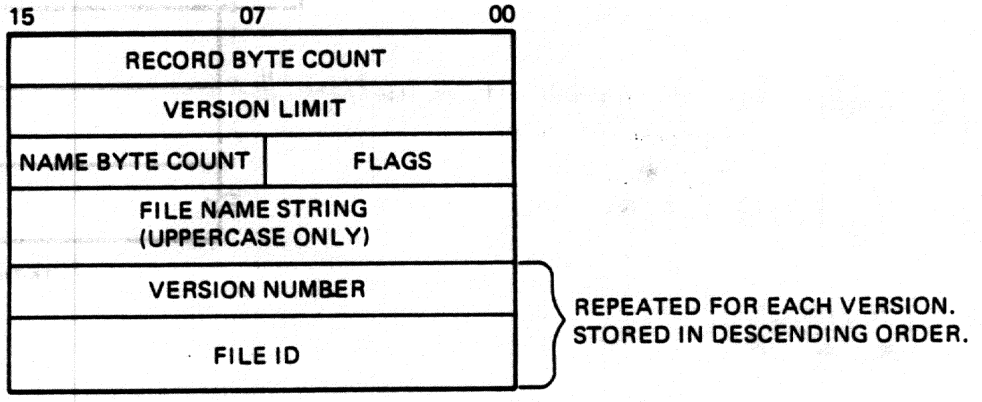
- Directory files
  - alphanumeric_name.DIR;1
  - UIC formatted directory names converted to their numeric equivalent

    [123,012] = 123012.DIR;1
- VAX/VMS supports up to seven levels of subdirectories beneath the UFD level.

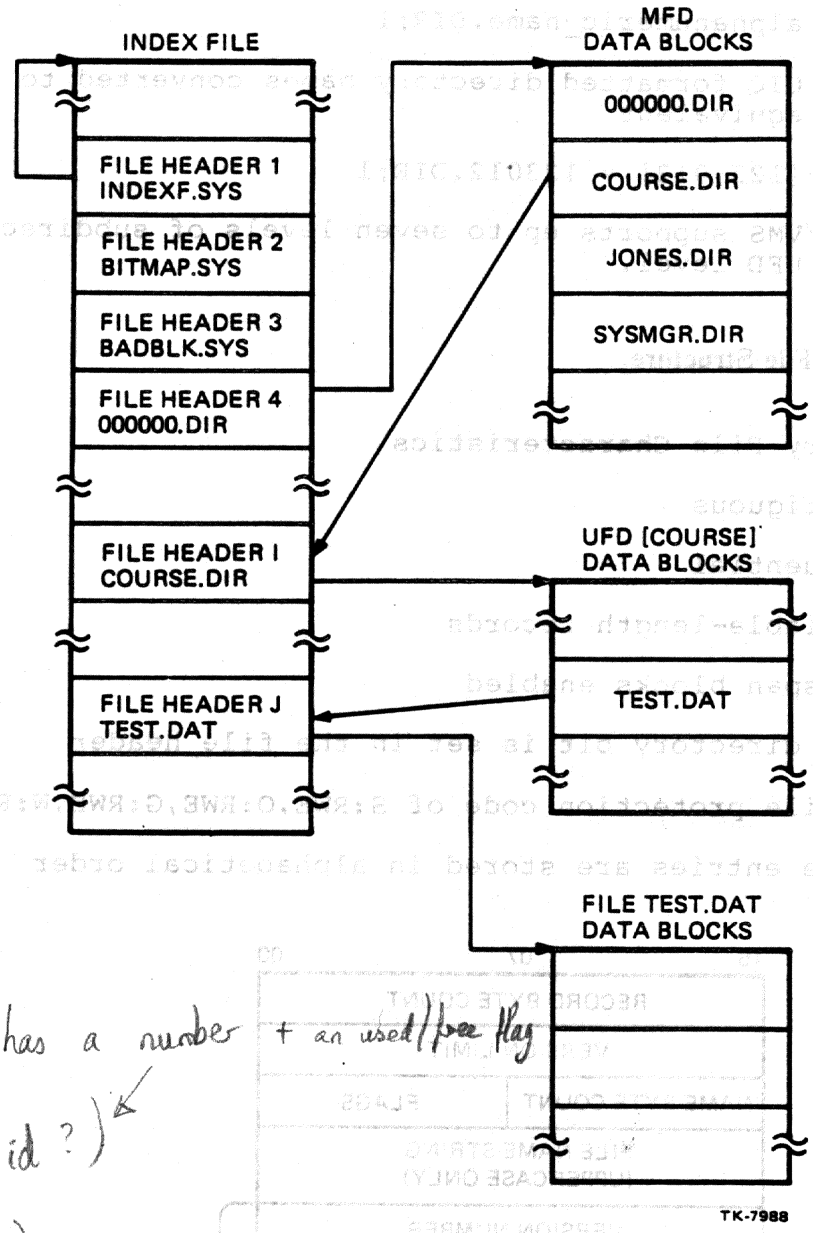
Directory File Structure

Directory File Characteristics

- Contiguous
- Sequential
- Variable-length records
- No span blocks enabled
- The directory bit is set in the file header
- A file protection code of S:RWE,O:RWE,G:RWE,W:RE
- File entries are stored in alphabetical order



Using ODS-2 File Headers to Access a File



Each file header has a number + an used/free flag

$dir/full ($ dir/id ?)

⇒ (n, m, p)

file_header_no

no of times this file_header_no used

Vol. no. in mult-volume set

FILE CHARACTERISTICS

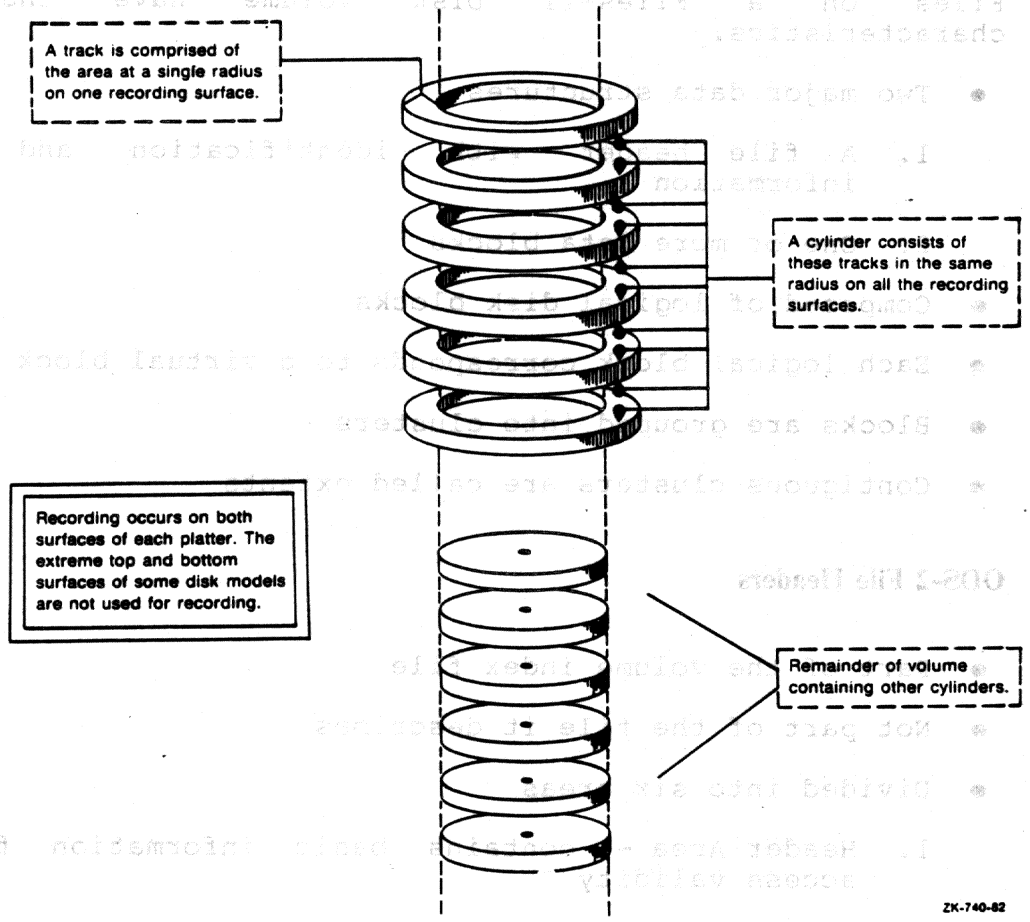
Files on a Files-11 Disk Volume have the following characteristics.

- Two major data structures
  1. A file header with identification and protection information
  2. One or more data blocks
- Composed of logical disk blocks
- Each logical block corresponds to a virtual block in the file
- Blocks are grouped into clusters
- Contiguous clusters are called extents

ODS-2 File Headers

- Part of the volume index file
- Not part of the file it describes
- Divided into six areas
  1. Header Area -- contains basic information for checking access validity
  2. Ident Area -- contains identification and accounting information
  3. Map Area -- contains pointers to the blocks allocated to the file
  4. Access Area -- contains access control list entries if any defined
  5. Reserved Area -- reserved for use by customers and DIGITAL
  6. Checksum Area -- validity check on the header's contents (the last word of the header) — STANDARD CHECKSUM VALUE
- Used to locate file data blocks on the disk.

DISK PHYSICAL CHARACTERISTICS



Seek time	Time needed to position the read/write heads over the correct radius.
Rotational latency time	Time it takes the desired block to move under the read/write heads, once the read/write heads are at the correct radius.

NOTE

The average seek time usually exceeds the average rotational latency by a factor of 2 to 4. Placing related blocks that are likely to be accessed as a unit at or close to the same radius on the disk will provide the best performance for the transfer of data between the disk surface and RMS-maintained buffers.

DISK ORGANIZATION

- Hardware

  Blocks
  Tracks
  Cylinders

- Software

  Clusters

  - Each disk must have at least 100 clusters

  - Defaults

    1 if # blocks < 50,000
    3 if # blocks > 50,000

  - Small clusters

    Efficient use of disk space
    More clusters ==> more overhead

  - Large clusters

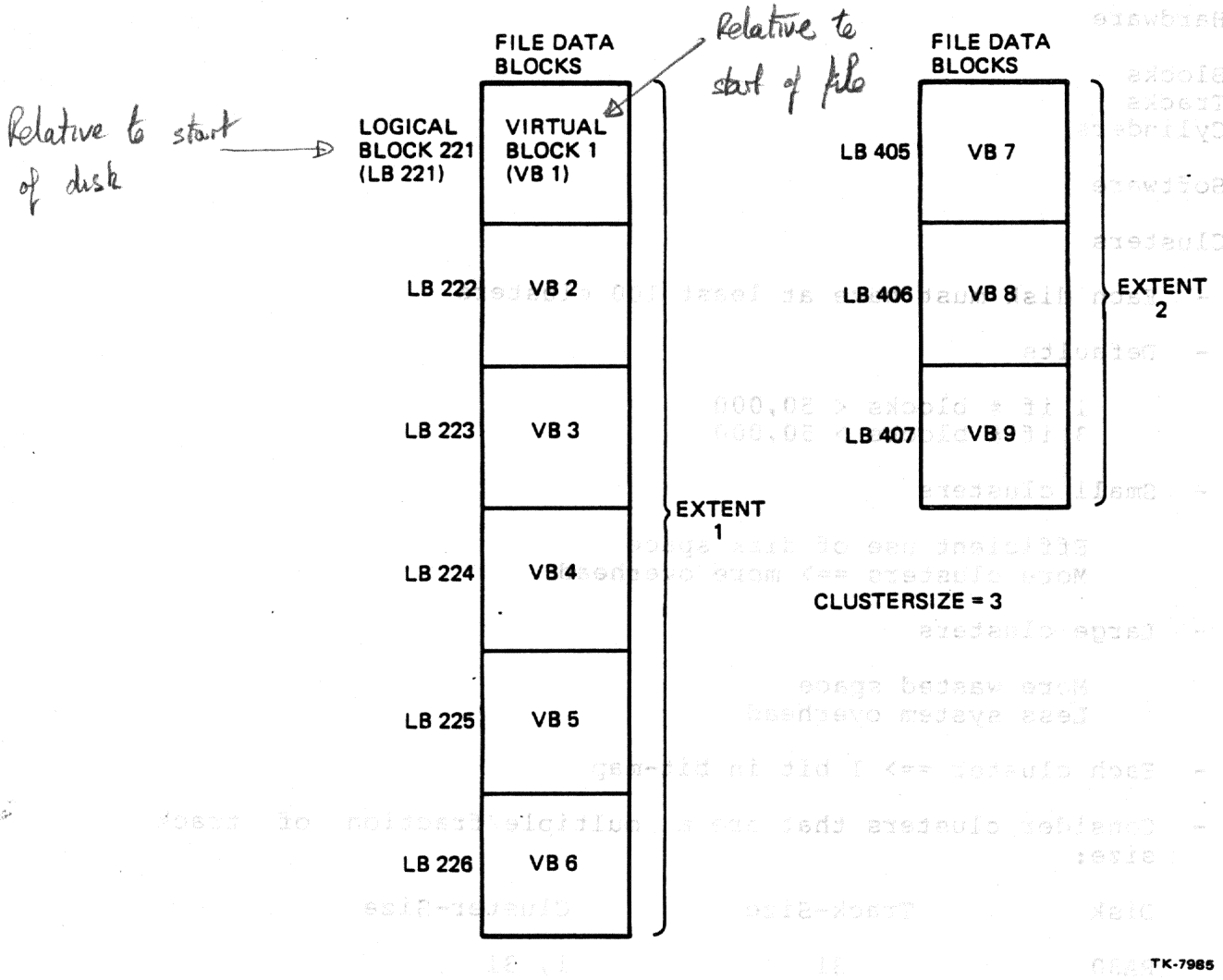
    More wasted space
    Less system overhead

  - Each cluster ==> 1 bit in bit-map

  - Consider clusters that are a multiple/fraction of track size:

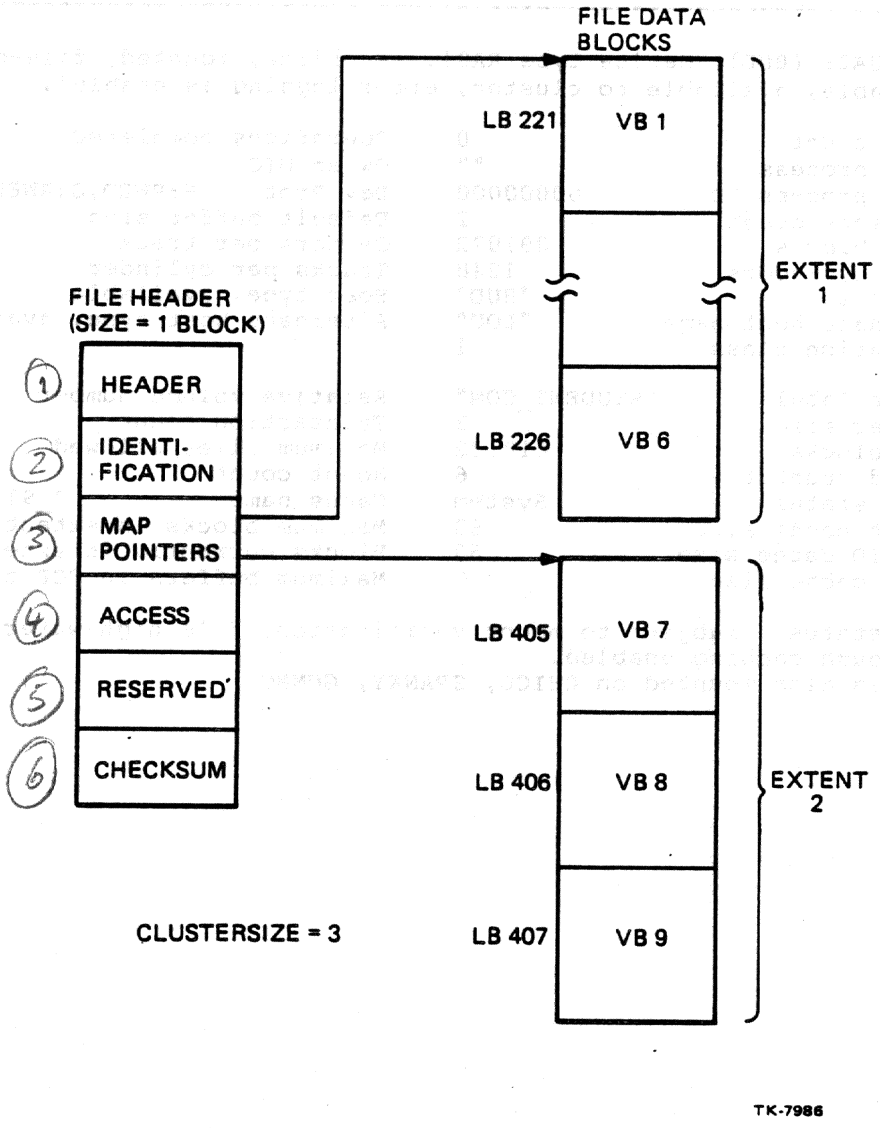
Disk	Track-Size	Cluster-Size
RA80	31	1, 31
RA81	51	1, 3, 17, 51
RM05	32	1, 2, 4, 8, 16, 32
RP07	50	1, 2, 5, 10, 25, 50

Relationship Between Blocks, Clusters, and Extents of a Typical Disk File



```
$ SHOW DEV/FULL  DISK$STUDENT   (to determine disk cluster size)

------------------------------------------------------------------------------

Disk $1$DUA2: (BUD), device type RA81, is online, mounted, file-oriented device,
    shareable, available to cluster, error logging is enabled.

    Error count                    0    Operations completed              186763
    Owner process                  ""   Owner UIC                         [1,1]
    Owner process ID        00000000    Dev Prot    S:RWED,O:RWED,G:RWED,W:RWED
    Reference count                2    Default buffer size                512
    Total blocks              891072    Sectors per track                   51
    Total cylinders             1248    Tracks per cylinder                 14
    Host name                  "BUD"    Host type, available          HS50, yes
    Alternate host name        "LOU"    Alternate host type, avail    HS50, yes
    Allocation class               1

    Volume label       "STUDENT_COM"    Relative volume number              0
    Cluster size                   3    Transaction count                   1
    Free blocks               618132    Maximum files allowed          148512
    Extend quantity                6    Mount count                         4
    Mount status              System    Cache name          "_$1$DUA0:XQPCACHE"
    Extent cache size             63    Maximum blocks in extent cache   61813
    File ID cache size            63    Blocks currently in extent cache 28014
    Quota cache size               0    Maximum buffers in FCP cache       311

  Volume status:  subject to mount verification, file high-water marking, write-
      through caching enabled.
  Volume is also mounted on CHICO, SPANKY, GUMMO.
```

Map Pointers to the Data Blocks of a Disk File



Example 1. Dump of an ODS-2 Disk File

```
$ DUMP/HEADER FILE.TXT


Dump of file WORK:[DORSEY.PROG.EXAMPLES]FILE.TXT;1 on 19-JUN-1984 09:50:33.38
File ID (5346,24,0)  End of file block 1 / Allocated 9

                              File Header

(1) Header area
        Identification area offset:             40
        Map area offset:                        100
        Access control area offset:             255
        Reserved area offset:                   255
        Extension segment number:               0
        Structure level and version:            2, 1
        File identification:                    (5346,24,0)
        Extension file identification:          (0,0,0)
        VAX-11 RMS attributes
            Record type:                        Variable
            File organization:                  Sequential
            Record attributes:                  Implied carriage control
            Record size:                        59
            Highest block:                      9
            End of file block:                  9
            End of file byte:                   176
            Bucket size:                        0
            Fixed control area size:            0
            Maximum record size:                255
            Default extension size:             0
            Global buffer count:                0
            Directory version limit:            0
        File characteristics:                   <none specified>
        Map area words in use:                  4
        Access mode:                            0
        File owner UIC:                         [VMS,DORSEY]
        File protection:                        S:RWED, O:RWED, G:RE, W:
        Back link file identification:          (18439,9,0)
        Journal control flags:                  <none specified>
        Highest block written:                  9

(2) Identification area
        File name:                              FILE.TXT;1
        Revision number:                        2          <-- # times opened for
        Creation date:                          19-JUN-1984 09:30:53.40      WRITE access
        Revision date:                          19-JUN-1984 09:30:53:90
        Expiration date:                        <none specified>
        Backup date:                            <none specified>

    Map area
        Retrieval pointers
            Count:          6       LBN:        147399
                            3                   213246

    Checksum:                                   5051
```

MODULE 4
OVERVIEW OF RMS FILE ORGANIZATIONS, RECORD FORMATS, AND ACCESS METHODS

Major Topics

- RMS File Organization
- Record Format
- Record Access Options

Source

Guide to VAX/VMS File Applications — Chapter 2 (Sections 2.1-2.2)
Chapter 8

RMS FILE ORGANIZATION

File organizations are as follows:

1. Sequential
2. Relative
3. Indexed Sequential

Sequential Files

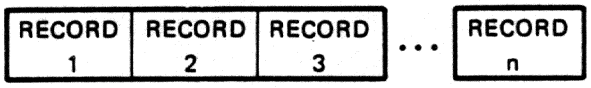
In sequential file organization, records in the file are arranged one after the other. This organization is the only one that supports all record formats: fixed-length, variable-length, variable with fixed-control, and stream (including undefined records).

Sequential files with fixed-length records have no overhead.

Records in sequential files are aligned on an even byte. If record size is an odd number, one more byte is inserted on the disk before the next record. This is transparent to programmers, except when a dump is examined.

Unlike relative and indexed sequential files, sequential files do not have a prolog. Instead, all information about a sequential file is stored in the file header, which can be viewed with the DCL commands DIRECTORY/FULL and DUMP/HEADER, as well as with the Analyze/RMS_File Utility.

Sequential File Organization

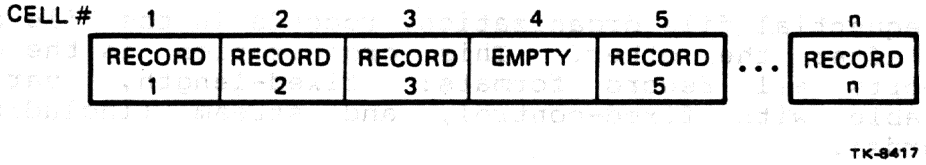


TK-8416

Relative Files

In the relative file organization, records are stored in a series of fixed-length positions called cells. This organization allows random retrieval of records by means of the relative record number, which identifies the position of the record cell relative to the beginning of the file. Relative files also contain a prolog.

Relative File Organization



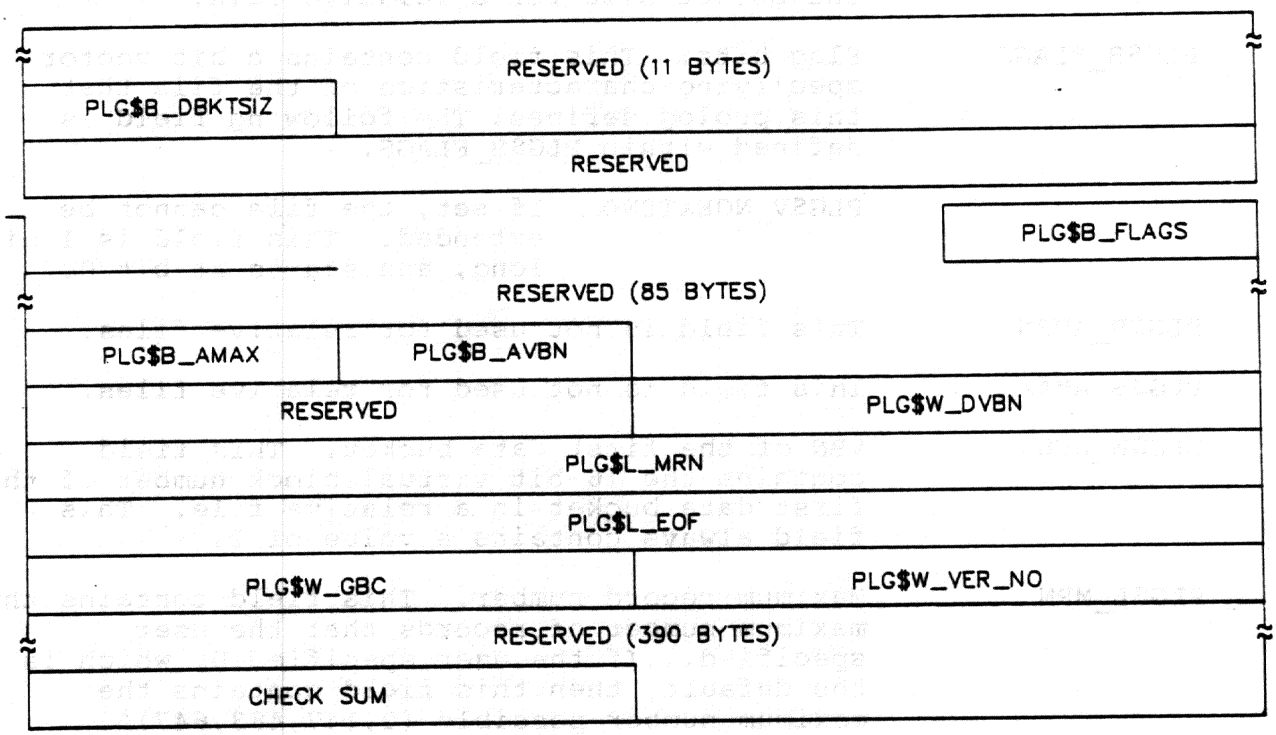
Prolog Description

A relative file starts with a 1-block prolog that contains specific information about the file as a whole. The prolog is located at virtual block 1, and the data buckets begin at virtual block 2. The most important fields for a relative file are the maximum record number field (PLG$L_MRN) and the field containing the end-of-file block number (PLG$L_EOF). The last word of the prolog contains the standard Files-11 additive checksum field.

Format of a Prolog for a Relative File

NOTE

The fields of the figure run from right-to-left



BU-2424

Contents of a Record Prolog for a Relative File

Field Name	Description
PLG$B_DBKTSIZ	Data bucket size. This field is not used for relative files. The FAT$B_BKTSIZE field defines the bucket size for a relative file.
PLG$B_FLAGS	Flag bits. This field contains a bit vector specifying characteristics of the file that this prolog defines. The following field is defined within PLG$B_FLAGS.
	PLG$V_NOEXTEND If set, the file cannot be extended. This field is 1 bit long, and starts at bit 0.
PLG$B_AVBN	This field is not used for relative files.
PLG$B_AMAX	This field is not used for relative files.
PLG$W_DVBN	VBN of the first data bucket. This field contains the 16-bit virtual block number of the first data bucket in a relative file. This field always contains a value of 2.
PLG$L_MRN	Maximum record number. This field contains the maximum number of records that the user specified. If the user specified 0, which is the default, then this field contains the maximum number possible (2,147,483,647)*. This field can be set with the RMS field FAB$L_MRN and the FDL attribute FILE MAX_RECORD_NUMBER.
PLG$L_EOF	VBN of the end-of-file. This field represents the logical end of the file. It contains the virtual block number of the last bucket initialized. Buckets are filled with all zeroes when they are initialized.
PLG$W_VER_NO	This field is not used for relative files.
PLG$W_GBC	This field is not used for relative files.
Checksum	Additive checksum.

* The maximum value may be limited by the number of blocks on the device to be used.

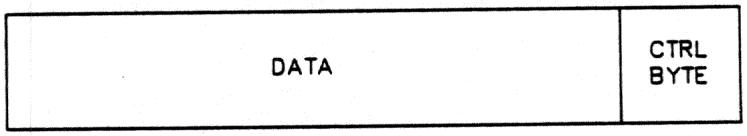
Relative Data Cell and Bucket Format

Records are stored in fixed-length cells in unformatted buckets. The fixed-length cells are numbered consecutively from 1 to n. This number is the relative record number, which indicates the record's position relative to the beginning of the file.

Records are stored starting at byte 0 of each bucket. They are packed contiguously so that they are byte-aligned. Cells (and thus records) cannot span bucket boundaries. If the bucket size is not a multiple of the cell size, then the remaining space in the bucket is unused. The next record in the file is stored in the first cell of the next bucket.

Each bucket contains a fixed number of fixed-length cells, and there are no overhead bytes in the bucket. The virtual blocks in the buckets are initialized (zeroed) when they are first allocated to support the way deleted records are handled.

Each cell contains at least one byte of record overhead. If the cell contains variable-length records, then the overhead is three bytes, which accounts for the 2-byte record-length field.



CELL SIZE = MAXIMUM RECORD SIZE + 1

BU-2425

Control Byte

bit 3	0 =	Cell never contained a record.
	1 =	A record is present in the cell (though it may have been deleted).
bit 2	1 =	Record has been deleted.

RECORD FORMAT

Record formats are:

1. Fixed
2. Variable
3. Variable with fixed control (disk only)
4. Stream (disk only)
5. Undefined (PAGEFILE.SYS, SWAPFILE.SYS) ← not accessed via RMS

Combinations of File Organization and Record Format Accepted by VAX RMS

File Organization	Record Format			
	Fixed	Variable	VFC	Stream
Sequential	Yes	Yes	Yes	Yes
Relative	Yes	Yes	Yes	No
Indexed	Yes	Yes	No	No

RECORD ACCESS OPTIONS

Record access methods are as follows:

1. Sequential
2. Random by key value/relative record number
3. Random by the record's file address (RFA)

Record Access Methods for File Organizations Supported by VAX RMS

Record Access Mode Permitted	File Organization		
	Sequential	Relative	Indexed
Sequential	Yes	Yes	Yes
Random by relative record number	Yes*	Yes	No
Random by key value	No	No	Yes
Random by record's file address	Yes**	Yes	Yes

* Random access by relative record number for sequential files is permitted for fixed-length record format on disk devices.

** Random access by record's file address is permitted only on disk devices.

Random Access to Indexed Files

Each of the program's Get requests in random access mode to an indexed file must specify both a key value and the index that RMS must search (for example, primary index, first alternate key index, second alternate key index, and so on). When RMS finds, by means of the index, the record that matches the key value, it reads the record and passes it to the program. Random access can be accomplished on any key by any of the following methods:

1. Exact match of key values.

2. Approximate match of key values. For example, if accessing an index in ascending sort order, RMS returns the record either equal to the user-supplied key value or with the next greater key value; conversely, if accessing the index in descending sort order (as of VMS 4.4), RMS returns the record either equal to the user-supplied key value or with the next lesser key value.

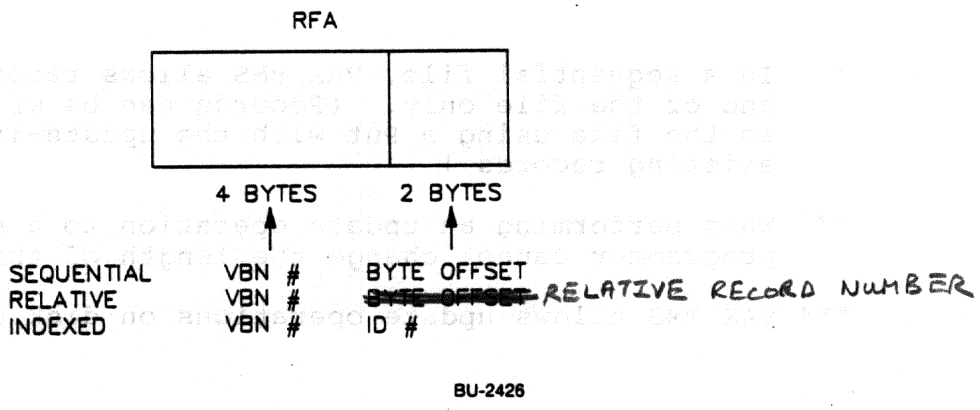
3. Generic match of key values. Generic match is applicable to string data-type keys only. A generic match is a match of some number of leading characters in the key. The number is determined by specifying a search key smaller than the entire field.

4. Combination of approximate and generic match.

Access by Record's File Address

Random-by-RFA access is supported for all file organizations provided that the files reside on disk devices. Whenever a record is accessed successfully from a file of any organization (using any of the record access modes already discussed) an internal representation of the record's location within the file is returned in the RAB field RAB$W_RFA. RMS can later examine the value in the RAB$W_RFA field and use it to retrieve that record, if specifically requested to do so (with a random-by-RFA access request).

The RFA is six bytes in length, which vary in content by file organization, as follows:



In the case of relative and indexed files, the VBN # is the number associated with the beginning block of a bucket.

Example

One example of the use of RFA access is to establish a record position for subsequent sequential access, which also could be done using other random record access modes (except for certain sequential files). Consider a sequential file with variable-length records that can only be accessed randomly using RFA access. Assume the file consists of a list of transactions, sorted previously by account value. Each account may have multiple transactions, so each account value may have multiple records for it in the file. Instead of reading the entire file until it finds the first record for the desired account number, it uses a previously saved RFA value and random-by-RFA access to set the current record position using a Find service at the first record of the desired account number. It can then switch to sequential record access and read all successive records for that account, until the account number changes or the end of the file is reached.

Record Processing Operations for
Sequential, Relative, and Indexed Files

Record Operation Permitted	File Organization		
	Sequential	Relative	Indexed
Read (Get)	Yes	Yes	Yes
Write (Put)	Yes*	Yes	Yes
Find	Yes	Yes	Yes
Delete	No	Yes	Yes
Update***	Yes**	Yes	Yes

* In a sequential file, VAX RMS allows records to be added at the end of the file only. (Records can be written to other points in the file using a Put with the update-if option to overwrite existing records.)

** When performing an update operation to a sequential file, the programmer cannot change the length of the record.

*** VAX RMS allows update operations on disk devices only.

Summary of Advantages and Disadvantages of RMS File Organization

Organization	Advantages	Disadvantages
Sequential	Uses disk and memory efficiently	Limited random access
	Provides optimal usage if the application accesses all records sequentially on each run	Cannot delete records
	Provides flexible record format	Can insert records only at the end of file.
	Allows data to be stored on different types of media	
	Allows records to be read- and write-shared*	
Relative	Allows sequential and random access for all languages	Allows data to be stored on disk only for accessing by relative cell number
	Allows random record deletion and insertion	Requires that files contain a record cell for each relative number allocated (files may not be densely populated)
	Allows records to be read- and write-shared	Requires that record cells be the same size
		Allows record insertion only to empty cells (or at the end of the file)
		Duplicate relative cell numbers are not allowed
Indexed	Allows sequential and random access by key value for all languages and by RFA for some languages	Allows data to be stored on disk only
	Allows random record deletion and insertion	Requires more disk space
	Allows records to be read- and write-shared	Uses more CPU time to process records
	Allows variable-length records to change length on update	Generally requires multiple disk accesses to process a record
	Duplicate key values possible	
	Automatic sort of records by primary and alternate keys; available during sequential access	

* Prior to VAX/VMS Version 4.4, write-sharing for sequential files was restricted to fixed-length 512-byte records.

MODULE 5
INDEXED FILE ORGANIZATION — INTERNAL STRUCTURE AND OVERHEAD

Major Topics

Part 1. Internals

- Overall tree structure and prolog
- Key descriptors
- Area descriptors
- Data bucket structure
- RRVs and bucket splits
- Key and data compression (Prolog 3)
- Index bucket
- Index compression
- Binary versus nonbinary index search
- Secondary index buckets and data records (SIDRs)

Part 2. Simulated Data Example

NOTE

The figures and tables presented in this module are based on preliminary materials prepared for *File and Record Management Internals* presently being written within DIGITAL (anticipated publication date, 1987). Although these materials are still undergoing technical review, they provide the most up-to-date, detailed documentation of the internal layouts used for VAX/VMS indexed files by RMS.

MODULE 5
INDEXED FILE ORGANIZATION — INTERNAL STRUCTURE AND OVERHEAD

PART 1. INTERNALS

OVERALL TREE STRUCTURE AND PROLOG

- Indexed files contain:
  - a prolog
  - key descriptors
  - area descriptors
  - primary index structure
  - secondary index structure(s)

- There are three types of indexed files:
  - Prolog 1
  - Prolog 2
  - Prolog 3 (default)

Characteristics	VMS 3.7+ Prologue 1*	VMS 3.7+ Prologue 2*	VMS 3.7 Prologue 3	VMS 4.0+ Prologue 3
Key data types	String data only	String and numeric data (2 and 4 bytes)	String data only	String data, packed decimal, and binary and integer numbers (2, 4, and 8 bytes)
Number of alternate keys	1-255	1-255	0 (restricted primary key)	1-255
Overlapping segmented keys				
Primary	Yes	Yes	No	No
Alternate	Yes	Yes	--	Yes
Values for alternates can change	Yes	Yes	--	Yes
Values for alternates can be null	Yes	Yes	--	Yes
Data, key, and index compression	No	No	Yes	Yes
CONVERT/RECLAIM	No	No	Yes**	Yes**

* Prologues 1 and 2 operate identically except for differences in key data types supported by each.

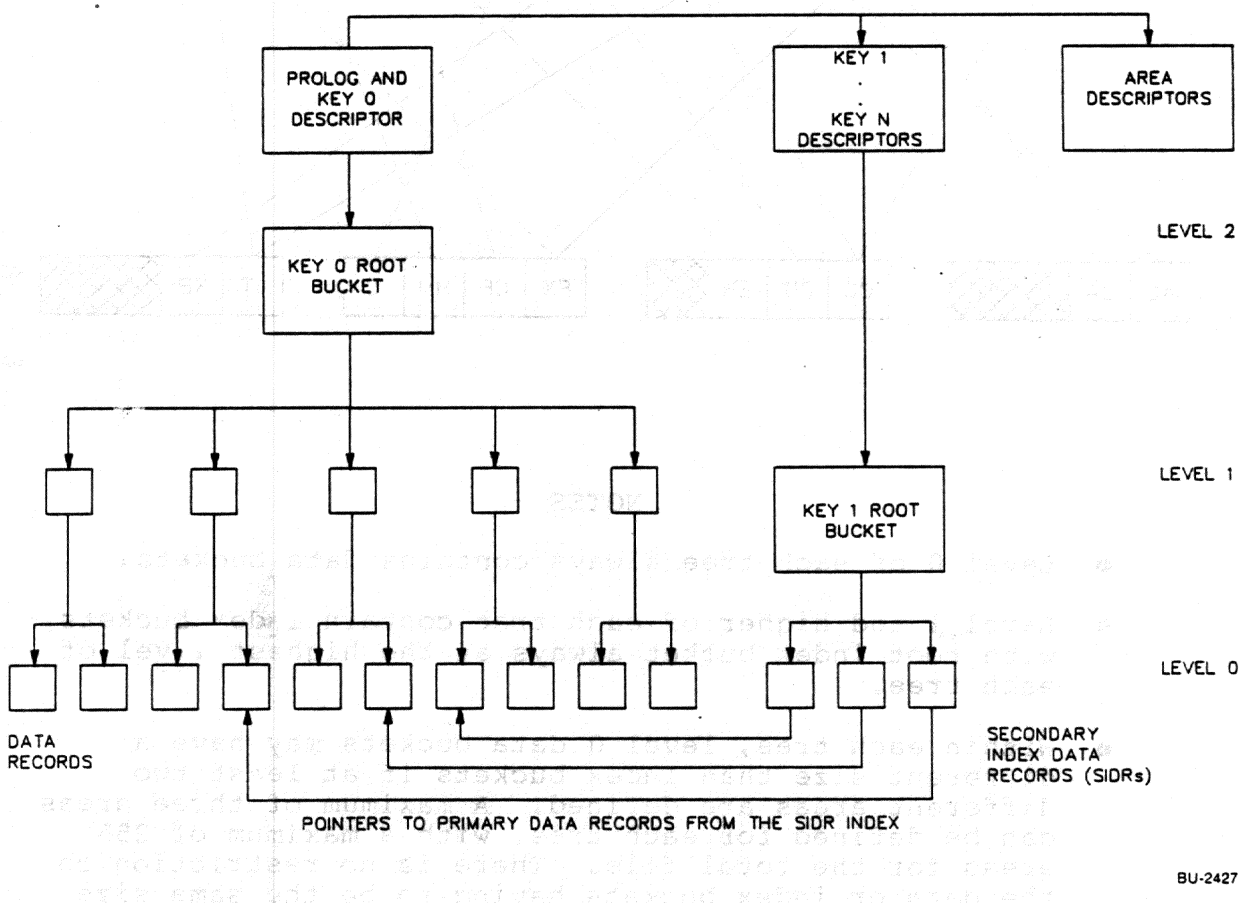
** Empty buckets (all records have been deleted) can be reclaimed for Prologue 3 files without rewriting the file.

Indexed Sequential Files

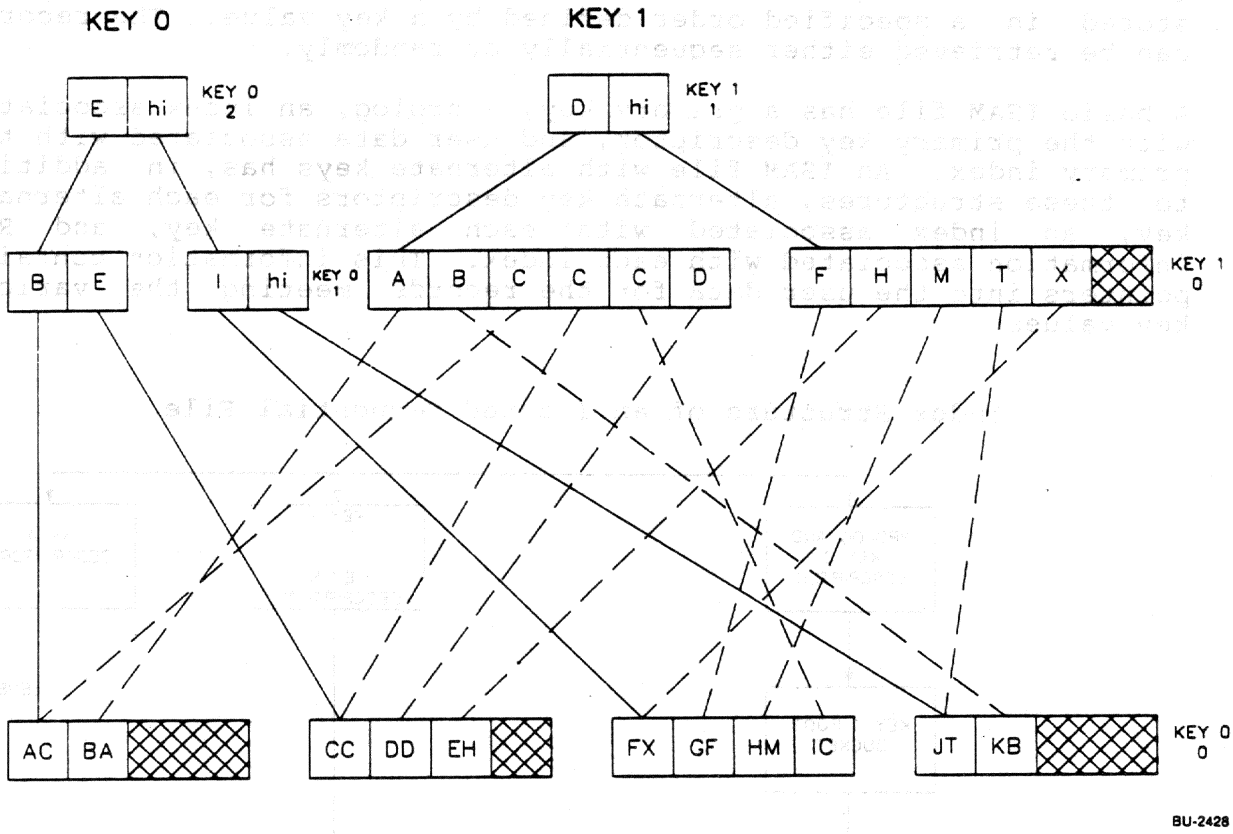
In the indexed sequential (ISAM) file organization, records are stored in a specified order defined by a key value. The records can be retrieved either sequentially or randomly.

A basic ISAM file has a primary key, a prolog, an index associated with the primary key descriptor, and user data associated with the primary index. An ISAM file with alternate keys has, in addition to these structures, alternate key descriptors for each alternate key, an index associated with each alternate key, and RMS information associated with each index. This information contains pointers into the user data for the records meeting the various key values.

Index Structure of an Indexed Sequential File



Tree Structure Associated with Each Key



NOTES

- Level 0 of each tree always contains data buckets.
- Level 1 and higher of each tree contain index buckets, with root index bucket always at the highest level of each tree.
- Within each tree, level 0 data buckets may have a different size than index buckets if at least two different areas are defined. A maximum of three areas can be defined for each tree, with a maximum of 255 areas for the total file. There is no restriction on the data or index buckets having to be the same size across trees.

- Four areas were defined for the above figure:

  -- Area 0 for key 0 level 0 data buckets

  -- Area 1 for key 0 level 1 and higher index buckets

  -- Area 2 for key 1 level 0 data buckets

  -- Area 3 for key 1 level 1 and higher index buckets

  A maximum of six areas could have been defined for these two trees. A minimum of three areas would have been necessary in order to obtain the three different bucket sizes depicted.

- FDL by default uses one bucket size for both the data and index buckets within each tree. The area definition produced by FDL has to be modified by the user in order to obtain a different bucket size for the data and index levels within any tree.

Prolog Description

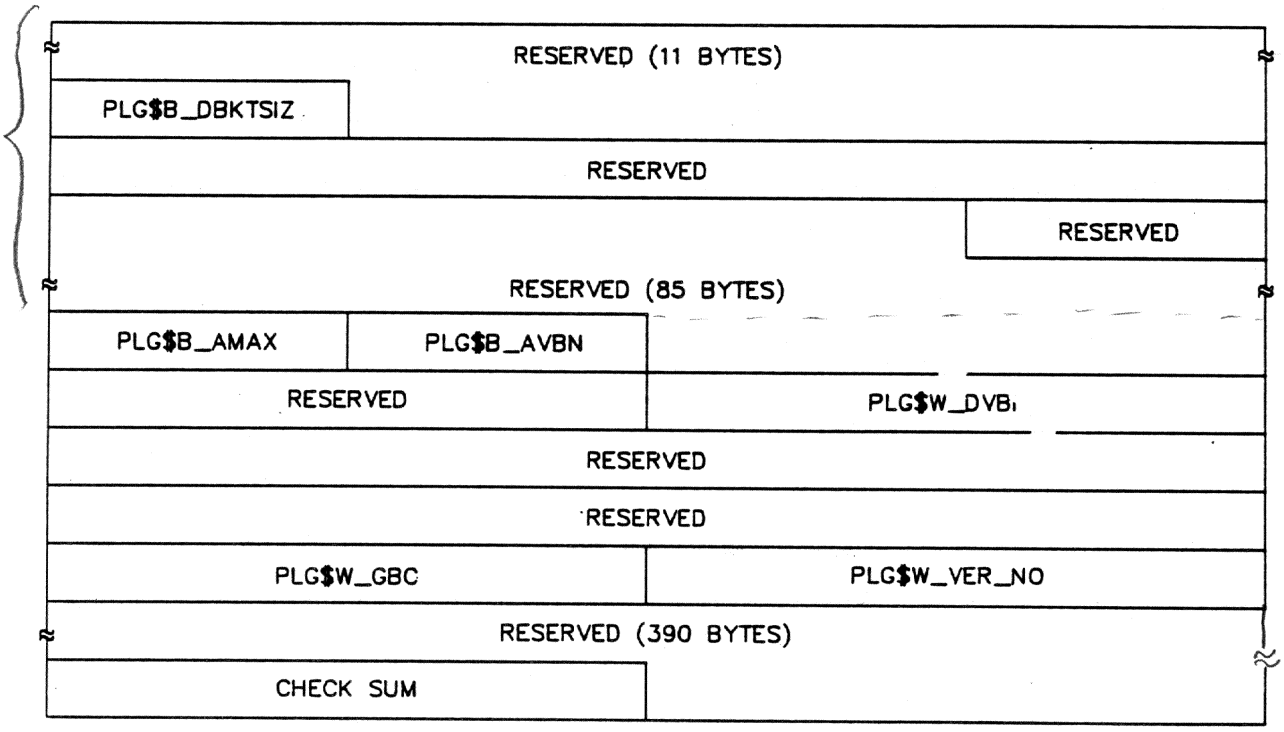
Like a relative file, an ISAM file begins with a prolog, which is a map to the rest of the file. The file-related overhead, which includes structures such as key and area descriptors, may range from 2 to 84 blocks long, starting with virtual block 1. These blocks are allocated from Area 0.

VBN 1 contains the prolog and the primary key descriptor, and the area descriptor follows in VBN 2 if the file has no alternate keys. The prolog descriptor for the primary key has one field in common with the key descriptor--the data bucket size field (PLG$B_DBKTSIZ)-- and the key descriptor for the primary key "overlays" the prolog descriptor. The last word of the prolog contains the standard Files-11 checksum field. The primary key descriptor begins in byte 0 of VBN 1.

Format of a Prolog for an Indexed Sequential File

NOTE

The fields in all the figures in this module run right-to-left.



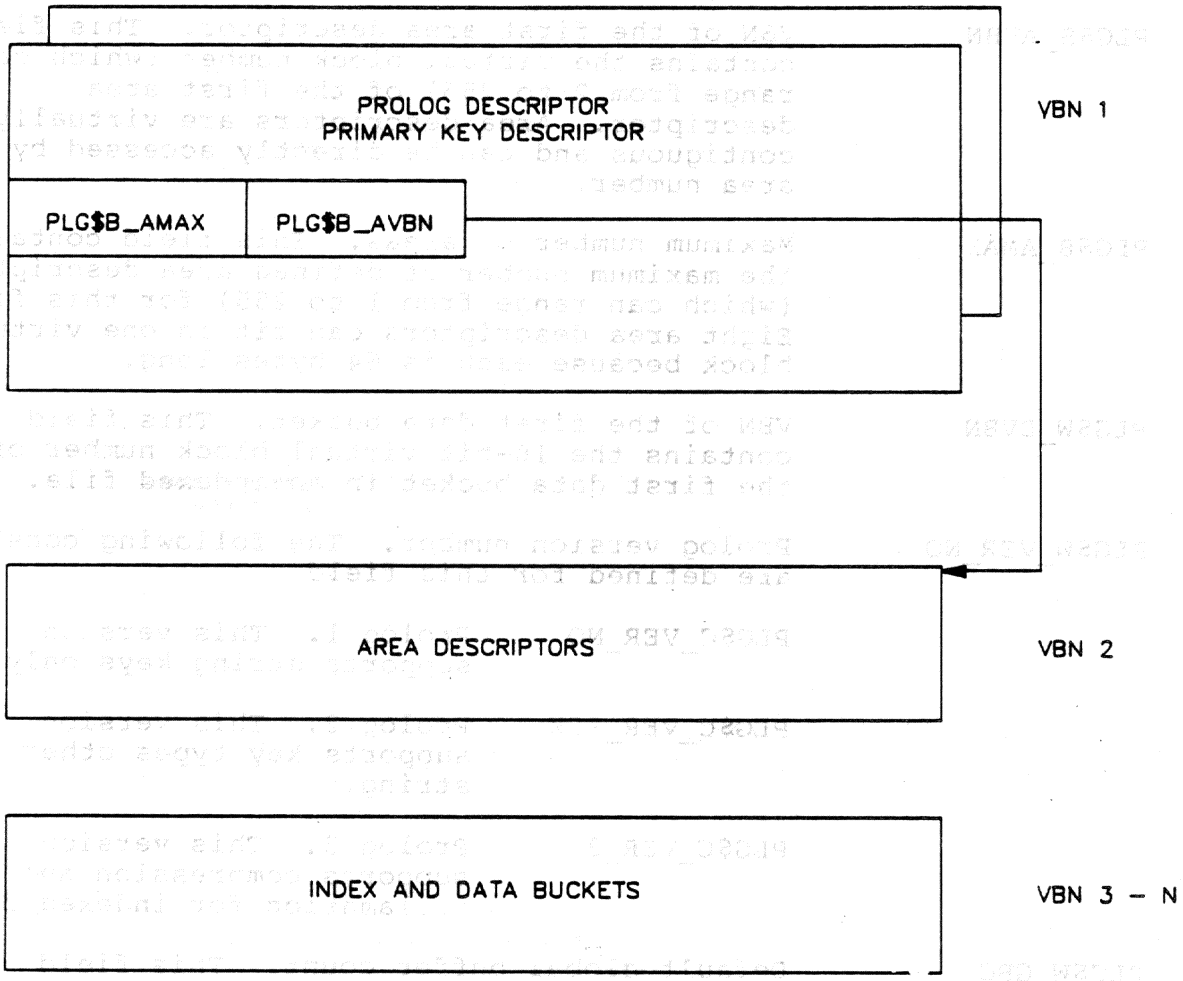
BU-2429

Contents of a Record Prolog for an Indexed Sequential File

Field Name	Description
PLG$B_DBKTSIZ	Data bucket size. This byte contains the bucket size for all data level (level 0) buckets.
PLG$B_AVBN	VBN of the first area descriptor. This field contains the virtual block number (which can range from 2 to 255) of the first area descriptor. Area descriptors are virtually contiguous and can be directly accessed by area number.
PLG$B_AMAX	Maximum number of areas. This field contains the maximum number of defined area descriptors (which can range from 1 to 255) for this file. Eight area descriptors can fit in one virtual block because each is 64 bytes long.
PLG$W_DVBN	VBN of the first data bucket. This field contains the 16-bit virtual block number of the first data bucket in ~~an indexed~~ a relative file.
PLG$W_VER_NO	Prolog version number. The following constants are defined for this field.
	PLG$C_VER_NO — Prolog 1. This version supports string keys only.
	PLG$C_VER_IDX — Prolog 2. This version supports key types other than string.
	PLG$C_VER_3 — Prolog 3. This version supports compression and space reclamation for indexed files.
PLG$W_GBC	Default global buffer count. This field contains the number of global buffers the user requested for the file. The number may range from 0 to 32,767; a value of 0 disables global buffering. This field can be set by the RMS field FAB$W_GBC and the FDL attribute FILE GLOBAL_BUFFER_COUNT.

A maximum of three area descriptors are allowed for any given key. However, up to 255 area descriptors are allowed for a file with multiple keys.

Format of a Prolog for a File with a Single Key

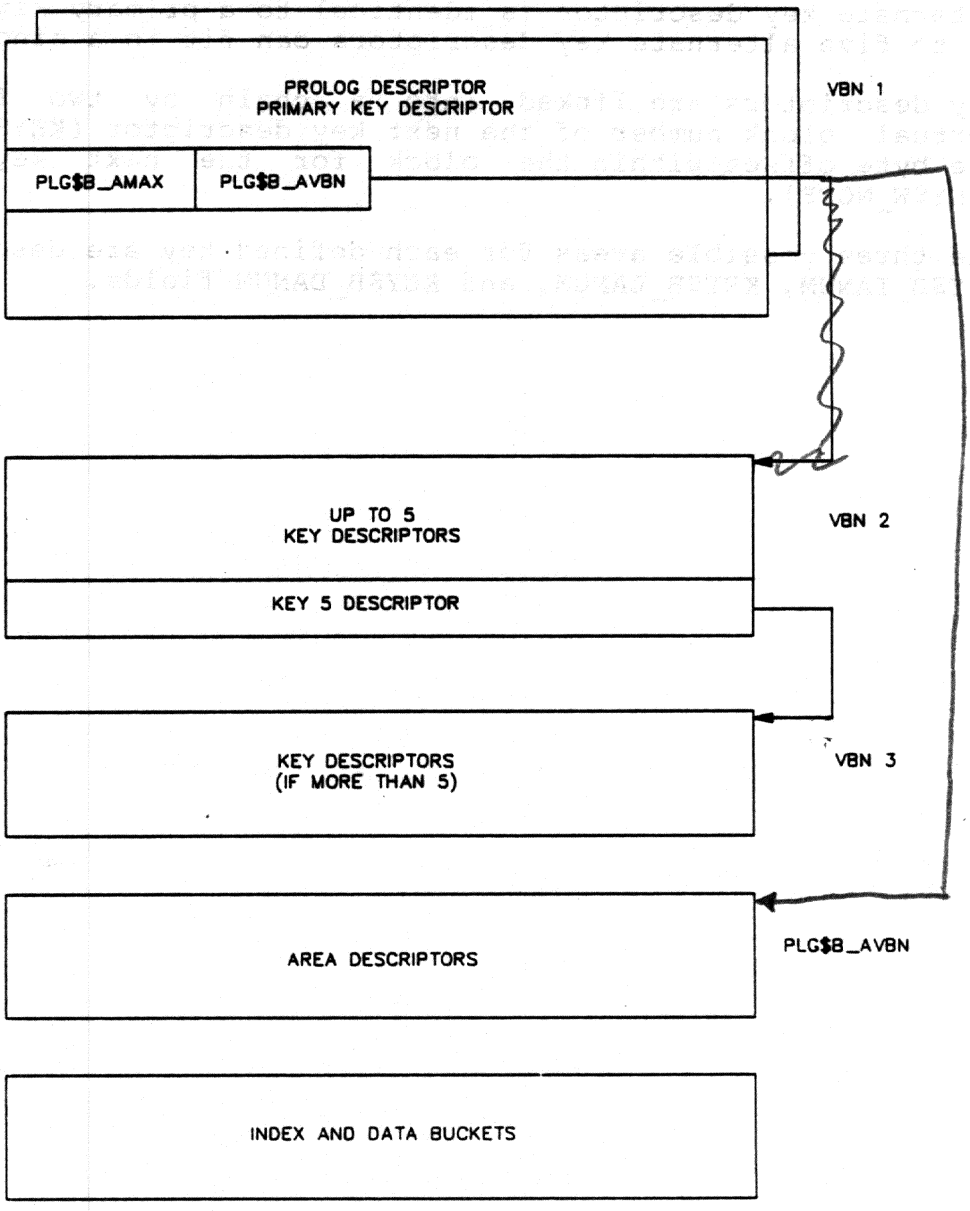


BU-2430

For a file with multiple keys, the virtual blocks containing the index and data buckets start at the value given by the following equation.

$$\frac{\text{PLG\$B\_AMAX}}{8\ \text{(truncated)}} + \text{PLG\$B\_AVBN}$$

Format of a Prolog for a File with Multiple Keys



BU-2431

KEY DESCRIPTORS

A key descriptor gives information about the characteristics of a key in an ISAM file. It supplies all the information RMS needs to retrieve, insert, update, and delete records. Every key has a corresponding key descriptor. Every file has a primary key, therefore it has at least one key descriptor. The primary key descriptor is located at virtual block 1 of the file (byte 0).

Key descriptors for secondary keys are called alternate key descriptors. They start at virtual block 2 of the prolog. An alternate key descriptor is identical to a primary key descriptor. Up to five alternate key descriptors can fit in a single block.

Key descriptors are linked into a chain by two fields: the virtual block number of the next key descriptor (KEY$L_IDXFL) and the byte offset within the block for the next key descriptor (KEY$W_NOFF).

The three possible areas for each defined key are described by the KEYB_IANUM, KEYB_LANUM, and KEY$B_DANUM fields.

Format of a Key Descriptor

KEY$L_IDXFL			
KEY$B_LANUM	KEY$B_IANUM	KEY$W_NOFF	
KEY$B_DATBKTSZ	KEY$B_IDXBKTSZ	KEY$B_ROOTLEV	KEY$B_DANUM
KEY$L_ROOTVBN			
KEY$B_NULLCHAR	KEY$B_SEGMENTS	KEY$B_DATATYPE	KEY$B_FLAGS
KEY$W_MINRECSZ		KEY$B_KEYREF	KEY$B_KEYSZ
KEY$W_DATFILL		KEY$W_IDXFILL	
KEY$W_POSITION1		KEY$W_POSITION	
KEY$W_POSITION3		KEY$W_POSITION2	
KEY$W_POSITION5		KEY$W_POSITION4	
KEY$W_POSITION7		KEY$W_POSITION6	
KEY$B_SIZE3	KEY$B_SIZE2	KEY$B_SIZE1	KEY$B_SIZE
KEY$B_SIZE7	KEY$B_SIZE6	KEY$B_SIZE5	KEY$B_SIZE4
KEY$T_KEYNAM (32 BYTES)			
KEY$L_LDVBN			
KEY$B_TYPE3	KEY$B_TYPE2	KEY$B_TYPE1	KEY$B_TYPE
KEY$B_TYPE7	KEY$B_TYPE6	KEY$B_TYPE5	KEY$B_TYPE4

BU-2432

* 96 bytes
* max. of 5 to a block
* Prolog holds 1° Key Descriptor

Contents of a Key Descriptor

Field Name	Description
KEY$L_IDXFL	VBN for next key descriptor. This field is checked only when the KEY$W_NOFF field contains a value of 0. When the KEY$L_IDXFL and the KEY$W_NOFF fields both contain a value of 0, the last key descriptor has been found.
KEY$W_NOFF	Offset to next key descriptor. This field contains the offset to the next key descriptor in the chain of key descriptors. This offset is relative to the beginning of the virtual block number contained in the KEY$L_IDXFL field.
KEY$B_IANUM	Index area number. This field contains the index bucket area number to be used for the index buckets for the key, from level 2 to the root bucket. It represents the area identification number contained in the AREA$B_AREAID field. It can range from 0 to 254. The default is 0, which indicates area 0. This field is set with the RMS field XAB$B_IAN and the FDL attribute KEY INDEX_AREA.
KEY$B_LANUM	Level 1 area number. This field contains the area number of the lowest level (level 1) of the index. It represents the area identification number contained in the AREA$B_AREAID field, ranging from 0 to 254. If this field contains a value of 0, only the KEY$B_IANUM field is used. This field is set with the RMS field XAB$B_LAN.
KEY$B_DANUM	Data area number. This field contains the area number of the data level (level 0) of the index buckets for the key. It represents the area identification number contained in the AREA$B_AREAID field. It can range from 0 to 254. The default is 0, which indicates area 0. This field is set with the RMS field XAB$B_DAN and the FDL attribute KEY DATA_AREA.

Contents of a Key Descriptor (Cont.)

Field Name	Description
KEY$B_ROOTLEV	Root level. This field contains the level number of the root bucket for the key. In other words, this field contains the height of the index tree. Levels are numbered from 0 to n, where 0 indicates the data level and n indicates the root level. This field sets the RMS field XAB$B_LVL after an Open or Display service.
KEY$B_IDXBKTSZ	Index bucket size. This field contains the size (number of virtual blocks) of the index-level buckets (level 1 to n) for the key. This field sets the RMS field XAB$B_IBS after an Open or Display service.
KEY$B_DATBKTSZ	Data bucket size. This field contains the size (number of virtual blocks) of the data-level buckets (level 0) for the key. This field sets the RMS field XAB$B_DBS after an Open or Display service.
KEY$L_ROOTVBN	VBN of the root bucket. Contains the virtual block number of the index root bucket for the key. After an Open or Display service, this field sets the RMS field XAB$L_RVB.
KEY$B_FLAGS	Flag bits. This field may be set with the RMS field XAB$B_FLG. The following 1-bit fields are defined within KEY$B_FLAGS.
KEY$V_DUPKEYS	Set if duplicate key values are allowed. This field starts at bit 0. It may be set with the RMS field XAB$V_DUP.
KEY$V_CHGKEYS	Set if the key value may change on an update operation. This field starts at bit 1. It may be set with the RMS field XAB$V_CHG.
KEY$V_NULKEYS	Set if a null key character is enabled. This field starts at bit 2. It may be set with the RMS field XAB$V_NUL.

Contents of a Key Descriptor (Cont.)

Field Name		Description
KEY$B_FLAGS (Cont.)	KEY$V_IDX_COMPR	Set if the index is compressed. This field starts at bit 3. This field may be cleared with the RMS field XAB$V_IDX_NCMPR.
	KEY$V_INITIDX	Set if the index must be initialized. This field is used only when RMS creates the index for this key. Because area number information is not normally stored in memory for an open indexed file, the required area numbers to create (and initialize) the index are stored in the root bucket field. When a bucket split occurs and additional space has to be allocated, the area number stored in the bucket which is splitting is used as the area number for the new bucket. This field starts at bit 4.
	KEY$V_KEY_COMPR	Set if the key is compressed in data records. The key must be a Prolog 3 string key. This field starts at bit 6. This field may be overridden with the RMS field XAB$V_KEY_NCMPR.
	KEY$V_REC_COMPR	Set if the data portion of the record is compressed. This bit applies only to Prolog 3 files. This field starts at bit 7. This field may be overridden with the RMS field XAB$V_DAT_NCMPR.

Contents of a Key Descriptor (Cont.)

Field Name	Description
KEY$B_DATATYPE	Key data type. This field is used at file creation to declare the type of data in the key within each data record. It may be set with the RMS field XAB$B_DTP. The following constants may be specified.
	KEY$C_STRING — Left-justified string of unsigned, 8-bit bytes. This is the default.
	KEY$C_SGNWORD — Signed binary word.
	KEY$C_UNSGNWORD — Unsigned binary word.
	KEY$C_SGNLONG — Signed binary longword.
	KEY$C_UNSGNLONG — Unsigned binary longword.
	KEY$C_PACKED — Packed decimal string.
	KEY$C_SGNQUAD — Signed binary quadword.
	KEY$C_UNSGNQUAD — Unsigned binary quadword.
	KEY$C_DSTRING — String data type for descending keys (as of VMS 4.4).
	KEY$C_DSGNWORD — Signed binary word data type for descending keys (as of VMS 4.4).
	KEY$C_DUNSGNWORD — Unsigned binary word data type for descending keys (as of VMS 4.4).
	KEY$C_DSGNLONG — Signed binary longword data type for descending keys (as of VMS 4.4).
	KEY$C_DUNSGNLONG — Unsigned binary longword data type for descending keys (as of VMS 4.4).
	KEY$C_DPACKED — Packed decimal string data type for descending keys (as of VMS 4.4).

Contents of a Key Descriptor (Cont.)

Field Name	Description	
KEY$B_DATATYPE (Cont.)	KEY$C_DSGNQUAD	Signed binary quadword for descending keys (as of VMS 4.4).
	KEY$C_DUNSGNQUAD	Unsigned binary quadword for descending keys (as of VMS 4.4).
KEY$B_SEGMENTS	Number of segments. This field contains the number of key segments that make up the key. Only string keys may have multiple segments. A maximum of eight segments is allowed. After an Open or Display service, this field sets the RMS field XAB$B_NSG.	
KEY$B_NULLCHAR	Null character. This field contains any user-selected ASCII character value. This field is set with the RMS field XAB$B_NUL.	
KEY$B_KEYSZ	Total key size. This field contains the sum (in bytes) of the values in the fields KEY$B_SIZE through KEY$B_SIZ7. After an Open or Display service, this field sets the RMS field XAB$B_TKS.	
KEY$B_KEYREF	Key of reference. This field tells whether a primary key or an alternate key has been defined. It contains a value ranging from 0 to 254, indicating which key has been defined. A value of 0 indicates the primary key, 1 indicates the first alternate key, and so on. This field is set with the RMS field XAB$B_REF.	
KEY$W_MINRECSZ	Minimum record length. This field contains the minimum length (in bytes) needed to hold the key. After an Open or Display service, this field sets the RMS field XAB$W_MRL.	

Contents of a Key Descriptor (Cont.)

Field Name	Description
KEY$W_IDXFILL	Index fill quantity. This field indicates the maximum number of bytes in an index bucket. The largest possible fill is the bucket size, in blocks, multiplied by 512. This field is set with the RMS field XAB$W_IFL.
KEY$W_DATFILL	Data fill quantity. This field indicates the maximum number of data bytes in a data bucket. The largest possible fill size is the bucket size, in blocks, multiplied by 512. This field is set with the RMS field XAB$W_DFL.
KEY$W_POSITION	Segment position. This field marks the beginning position of the first of up to eight key segments. It is set with the RMS field XAB$W_POS0.
KEY$W_POSITION1	Position 1. This field marks the beginning position of the second key segment. It is set with the RMS field XAB$W_POS1.
KEY$W_POSITION2	Position 2. This field marks the beginning position of the third key segment. It is set with the RMS field XAB$W_POS2.
KEY$W_POSITION3	Position 3. This field marks the beginning position of the fourth key segment. It is set with the RMS field XAB$W_POS3.
KEY$W_POSITION4	Position 4. This field marks the beginning position of the fifth key segment. It is set with the RMS field XAB$W_POS4.
KEY$W_POSITION5	Position 5. This field marks the beginning position of the sixth key segment. It is set with the RMS field XAB$W_POS5.
KEY$W_POSITION6	Position 6. This field marks the beginning position of the seventh key segment. It is set with the RMS field XAB$W_POS6.
KEY$W_POSITION7	Position 7. This field marks the beginning position of the eighth (and last) key segment. It is set with the RMS field XAB$W_POS7.

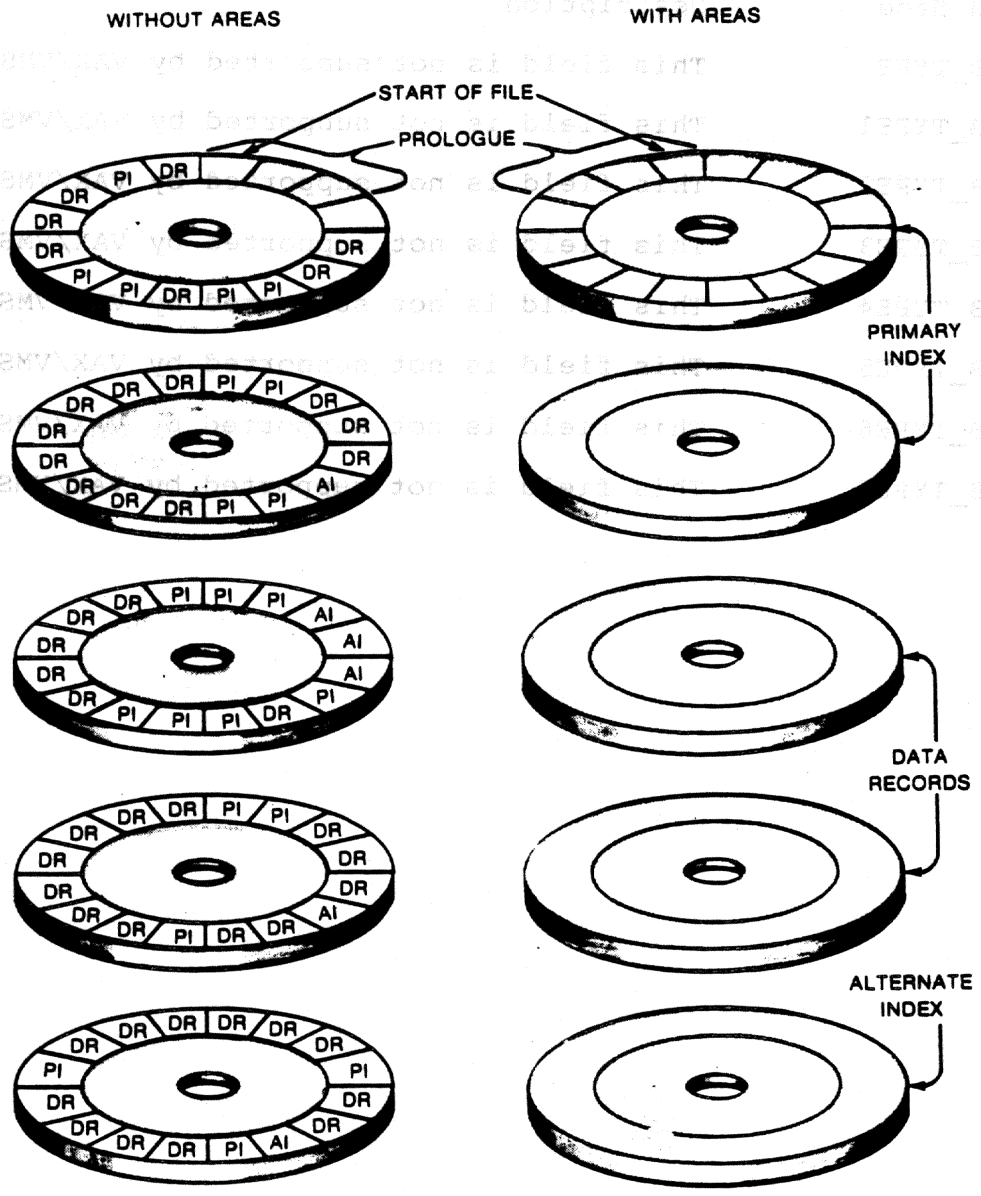
Contents of a Key Descriptor (Cont.)

Field Name	Description
KEY$B_SIZE	Segment size. This field contains the size of the first key segment. It is set with the RMS field XAB$B_SIZ0.
KEY$B_SIZE1	Size 1. This field contains the size of the second key segment. It is set with the RMS field XAB$B_SIZ1.
KEY$B_SIZE2	Size 2. This field contains the size of the third key segment. It is set with the RMS field XAB$B_SIZ2.
KEY$B_SIZE3	Size 3. This field contains the size of the fourth key segment. It is set with the RMS field XAB$B_SIZ3.
KEY$B_SIZE4	Size 4. This field contains the size of the fifth key segment. It is set with the RMS field XAB$B_SIZ4.
KEY$B_SIZE5	Size 5. This field contains the size of the sixth key segment. It is set with the RMS field XAB$B_SIZ5.
KEY$B_SIZE6	Size 6. This field contains the size of the seventh key segment. It is set with the RMS field XAB$B_SIZ6.
KEY$B_SIZE7	Size 7. This field contains the size of the eighth key segment. It is set with the RMS field XAB$B_SIZ7.
KEY$T_KEYNAM	Optional key name. This 32-byte field contains the name of the key as an ASCII string. It is set with the RMS field XAB$L_KNM.
KEY$L_LDVBN	VBN of the first data bucket. This field contains the starting virtual block number for the first data-level bucket for the key. After an Open or Display service, this field sets the RMS field XAB$L_DVB.

Contents of a Key Descriptor (Cont.)

Field Name	Description
KEY$B_TYPE	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE1	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE2	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE3	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE4	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE5	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE6	This field is not supported by VAX/VMS Version 4.4.
KEY$B_TYPE7	This field is not supported by VAX/VMS Version 4.4.

Indexed File With and Without Areas



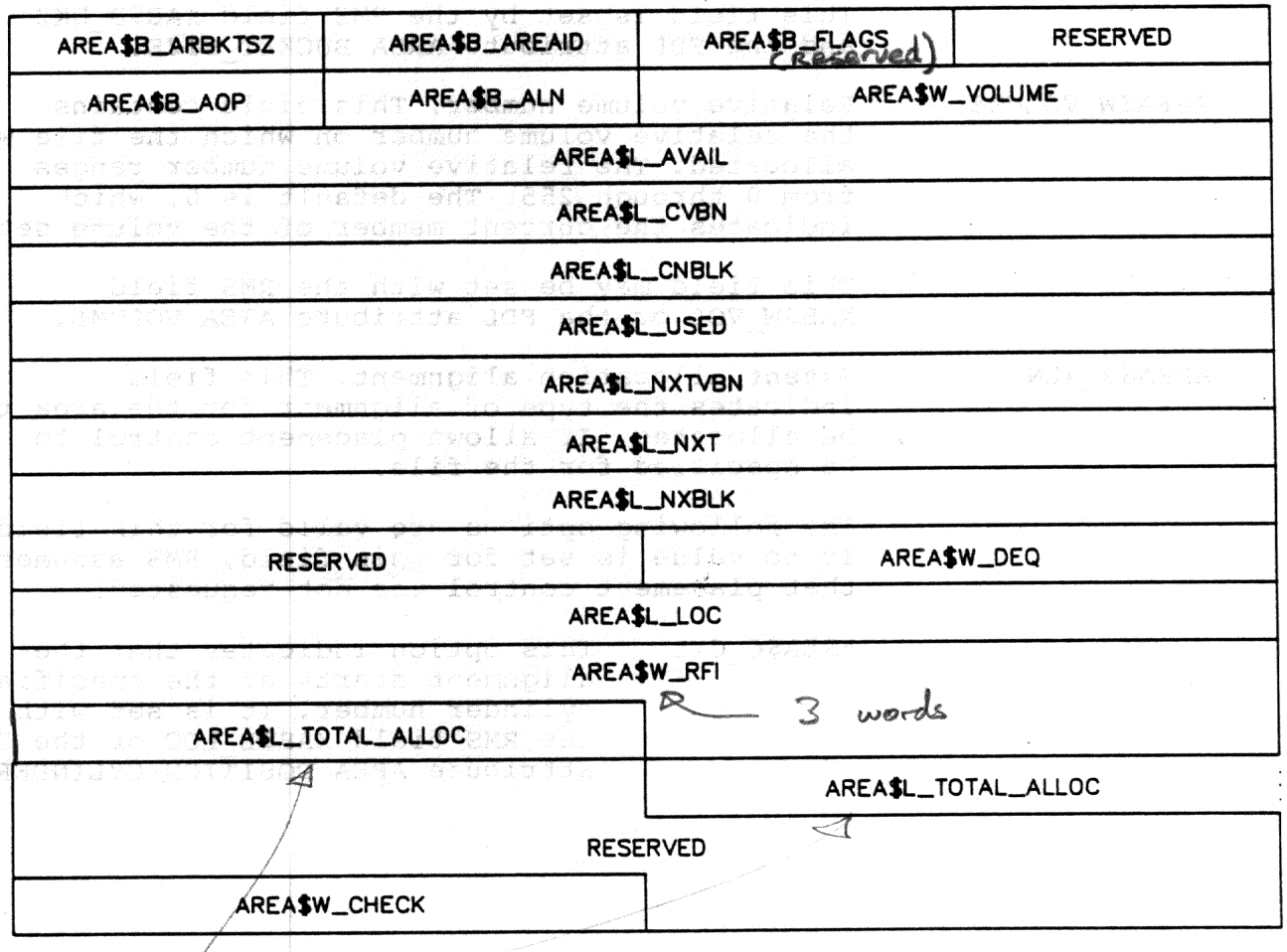
AREA DESCRIPTORS

In an ISAM file, the user may independently allocate and manage sections of contiguous virtual blocks, called areas, according to how each will be used. The area descriptor contains this function-specific information. There is an area descriptor for every area in the file.

Defining multiple areas allows the user to declare different bucket sizes for index buckets and data buckets. Areas also allow the user to control where the various elements or sections of the file are placed on the disk.

Area descriptors follow the last key descriptor of the file and occupy contiguous virtual blocks. Up to eight area descriptors can fit in one virtual block.

Format of an Area Descriptor



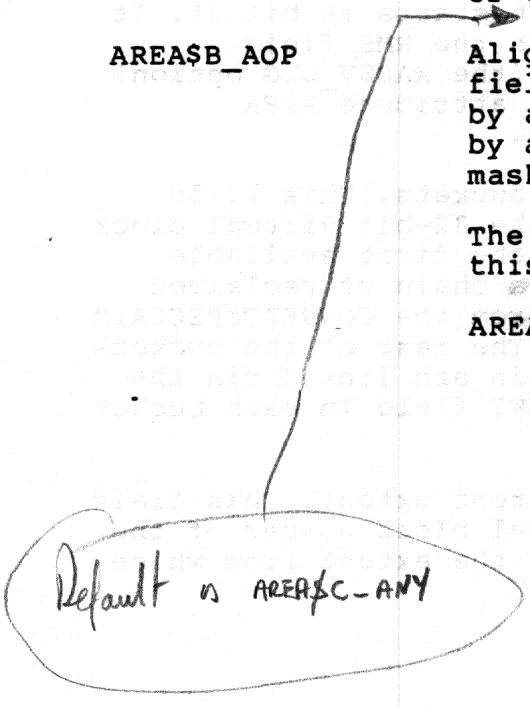
BU-2433

Contents of an Area Descriptor

Field Name	Description
AREA$B_AREAID	Area ID. This field contains the area identification number, which can range from 0 to 254. It indicates the target area for RMS operations. It is also used as a redundancy check because all area descriptors are located at a fixed position relative to the start of the area descriptor blocks. This field is set by the RMS field XAB$B_AID or the FDL attribute AREA n, where n indicates the area number.
AREA$B_ARBKTSZ	Area bucket size. This field contains the bucket size for the area, which can range from 1 to 63 blocks. It represents the granularity of the allocation. This field is set by the RMS field XAB$B_BKZ and the FDL attribute AREA BUCKET_SIZE.
AREA$W_VOLUME	Relative volume number. This field contains the relative volume number on which the file was allocated. The relative volume number ranges from 0 through 255. The default is 0, which indicates the current member of the volume set. This field may be set with the RMS field XAB$W_VOL or the FDL attribute AREA VOLUME.
AREA$B_ALN	Extent allocation alignment. This field indicates the type of alignment for the area to be allocated. It allows placement control to be specified for the file. The following options are valid for this field. If no value is set for this field, RMS assumes that placement control was not requested. AREA$C_CYL — This option indicates that the alignment starts at the specified cylinder number. It is set with the RMS field XAB$L_LOC or the FDL attribute AREA POSITION CYLINDER.

Contents of an Area Descriptor (Cont.)

Field Name	Description	
AREA$B_ALN (Cont.)	AREA$C_LBN	This option indicates that the alignment starts at the specified logical block. It may be set with the RMS field XAB$L_LOC or the FDL attribute AREA POSITION LOGICAL.
	AREA$C_RFI	This option indicates that the alignment starts as close as possible to the file specified by the related file identification field (XAB$W_RFI), at the virtual block number of the file specified in the location field (XAB$L_LOC). It is also set with the FDL attribute AREA POSITION FILE_ID or the FDL attribute AREA POSITION FILE_NAME.
	AREA$C_VBN	Virtual block alignment. This option indicates that the alignment starts at the specified virtual block. It is set with the RMS field XAB$L_LOC or the FDL attribute AREA POSITION VIRTUAL.
	This field is set with the RMS field XAB$B_ALN or the FDL attribute AREA POSITION.	
AREA$B_AOP	Alignment options. This field is a binary bit field where each allocation option is defined by a certain bit. Each option is identified by a symbolic bit offset and has a corresponding mask value.	
	The following fields (or masks) are defined for this field.	
	AREA$V_HARD	This option specifies absolute alignment. If the requested alignment cannot be performed, an error is returned.
		This option starts at bit 24. It is set with the RMS field XAB$B_AOP (the XAB$V_HRD option) or the FDL attribute AREA EXACT_POSITIONING.



Contents of an Area Descriptor (Cont.)

Field Name	Description	
AREA$B_AOP (Cont.)	AREA$V_ONC	This option requests that RMS locate the allocation on any available cylinder boundary. This option starts at bit 25. It is set with the RMS field XAB$B_AOP (the XAB$V_ONC option) or the FDL attribute AREA POSITION ANY_POSITION.
	AREA$V_CBT	This option indicates that the allocation or a later extension should occupy contiguous blocks, if possible. This option starts at bit 29. It is set with the RMS field XAB$B_AOP (the XAB$V_CBT option) or the FDL attribute AREA BEST_TRY_CONTIGUOUS.
	AREA$V_CTG	This option indicates that the initial allocation or later extensions must use only contiguous blocks. This option starts at bit 31. It is set with the RMS field XAB$B_AOP (the XAB$V_CTG option) or the FDL attribute AREA CONTIGUOUS.
	AREA$L_AVAIL	Available buckets. This field contains the 32-bit virtual block number of the first available bucket in a chain of reclaimed buckets (from the CONVERT/RECLAIM utility). The rest of the buckets on the chain are linked via the BKT$L_NXTBKT field in each bucket header.
AREA$L_CVBN		Starting VBN for the current extent. This field contains the first virtual block number of the current extent, which is the extent from which buckets are allocated.

Contents of an Area Descriptor (Cont.)

Field Name	Description
AREA$L_CNBLK	Number of blocks in current extent. This field contains the number of blocks that were allocated to this extent. The AREA$L_CVBN and AREA$L_CNBLK fields describe the result of an Extend operation for the current extent.
AREA$L_USED	Number of blocks used. This field contains the number of blocks that have been allocated from the current extent.
AREA$L_NXTVBN	Next VBN to use. This field contains the virtual block number of the starting block number of the next bucket allocated from the current extent.
AREA$L_NXT	Starting VBN for next extent. This field contains the starting virtual block number for the next extent in the chain. When there are no more empty blocks in the current extent, the next extent is made the current extent, and the next extent is initialized (zeroed). The area can only be extended when the next extent description has been zeroed. Thus, a value of 0 indicates that the current extent is the last (or only) extent in the chain.
AREA$L_NXBLK	Number of blocks in next extent. This field contains the number of blocks that were allocated to the next extent. The AREA$L_NXT and AREA$L_NXBLK fields describe the result of an Extend operation for the next extent.
AREA$W_DEQ	Default extend quantity. This field contains the default ~~file~~ extend quantity, which is the number of blocks to be added when RMS needs to extend the area. The user specifies this value, which can range from 0 through 65,535. If a value of 0 is specified, the file will be extended using a default extension value determined by RMS. A size less than 1 bucket will never be used. This field is set with the RMS field XAB$W_DEQ or the FDL attribute AREA EXTENSION.
AREA$L_LOC	Starting LBN on volume. This field contains the starting logical block number of the last extent of the area.

area

Contents of an Area Descriptor (Cont.)

Field Name	Description
AREA$W_RFI	Related file ID. This field contains the file ID of a related file.
AREA$L_TOTAL_ALLOC	Total block allocation. This field contains the total number of blocks initially allocated for an area during a Create operation or the number of blocks to be added to the area during an Extend operation. This field sets the XAB$L_ALQ field during Open, Create, Display, and Extend services.
AREA$W_CHECK	Checksum. This field allows the standard Files-11 checksum value to be stored in the last word of the area descriptor block.

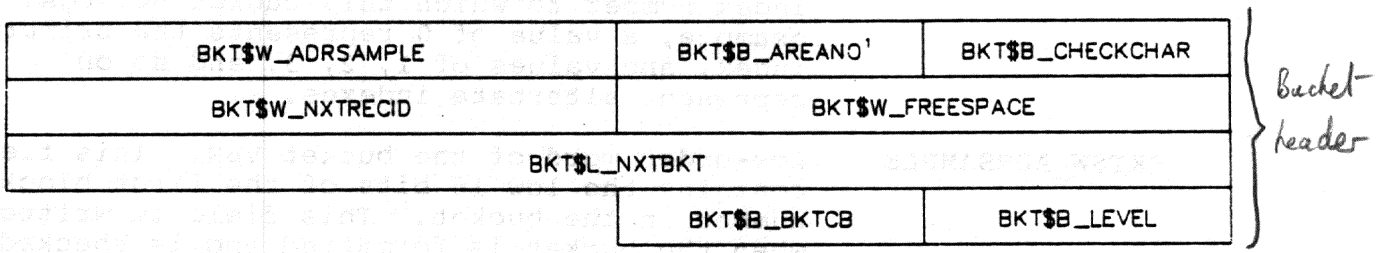
DATA BUCKET STRUCTURE

Following the prolog, key, and area descriptors in an ISAM file are storage structures called buckets, which can either be data buckets or index buckets. The size of a bucket is always a multiple of 512 bytes (1 to 63 blocks).

The bucket size is defined by the user. The size of the index buckets may be different within the same index, but the bucket size on each level for each key of reference is the same.

Buckets have two logical regions: a header area and an area that may be used to store records. The size of the actual bucket header is 14 bytes although the total size of bucket overhead is 15 bytes; the extra byte is a check byte at the end of each bucket.

Format of a Data Bucket for an Indexed Sequential File



[1] BKT$B_INDEXNO FOR PROLOG 3 FILES

BU-2434

Contents of a Data Bucket

Field Name	Description
BKT$B_CHECKCHAR	Bucket check character. This field contains a 1-byte check character. Whenever a bucket is written, the value in the check byte is incremented and copied into the first and last byte of the bucket. Whenever a bucket is read, the check bytes are compared for equality. By this technique, hardware failures during transfer are detectable.
BKT$B_AREANO	Area number or index number (BKT$B_INDEXNO).
BKT$B_INDEXNO	For Prolog 1 and 2 files, BKT$B_AREANO contains the area number of the area from which the bucket was allocated. For Prolog 3 files, BKT$B_INDEXNO contains the index number to which this bucket belongs. For example, a value of 0 represents the primary index, and values of 1, 2, 3, and so on represent alternate indexes.
BKT$W_ADRSAMPLE	Low-order word of the bucket VBN. This field contains the low 16 bits of the first block number in the bucket. This field is written when the bucket is formatted and is checked when the bucket is read into memory.
BKT$W_FREESPACE	First free byte of unused space in the bucket. This field contains the byte address relative to the start of the bucket of the first free byte in the unused portion of the record storage area of the bucket.
BKT$W_NXTRECID	Next available record ID. This field contains the ID number to use for the next record placed in the bucket. When the current ID is used, the value in this field is incremented by 1. The record ID does not depend on the record's physical position in the bucket because records are ordered by key value. Record IDs are assigned in different ways, depending on the prolog version of the file.

Contents of a Data Bucket (Cont.)

Field Name	Description
BKT$W_NXTRECID (Cont.)	For Prolog 3 files, the way the record ID is assigned depends on whether the bucket is newly created or a reclaimed bucket. If the bucket is new, the first record is assigned an ID of 1, and subsequent record IDs are assigned in order. If the bucket is reclaimed, the record ID is assigned from the value in the BKT$W_NXTRECID field, and subsequent IDs are assigned in order. For Prolog 1 and 2 files, the BKT$W_NXTRECID field consists of two 1-byte fields, BKT$B_NXTRECID and BKT$B_LSTRECID, that indicate a range of record IDs. The low byte (BKT$B_NXTRECID) contains the next record ID that can be assigned. The high byte contains the upper limit of 255 (hex FF), which is the highest record ID that can be assigned. However, if the bucket is new, the first record is always assigned an ID of 1.
BKT$L_NXTBKT	VBN of the next bucket. This field contains the starting virtual block number of the next bucket at this level of data buckets; it is a horizontal link to the next bucket. This pointer always points to a bucket of the same size. The last bucket in a level points to the first bucket in the level.
BKT$B_LEVEL	Bucket level number. This field contains the level number relative to the data level for this bucket. Data-level buckets contain a value of 0. The lowest level in the index structure is represented by a value of 1, the next highest level by a value of 2, and so on. The root level of the index always has the highest value.

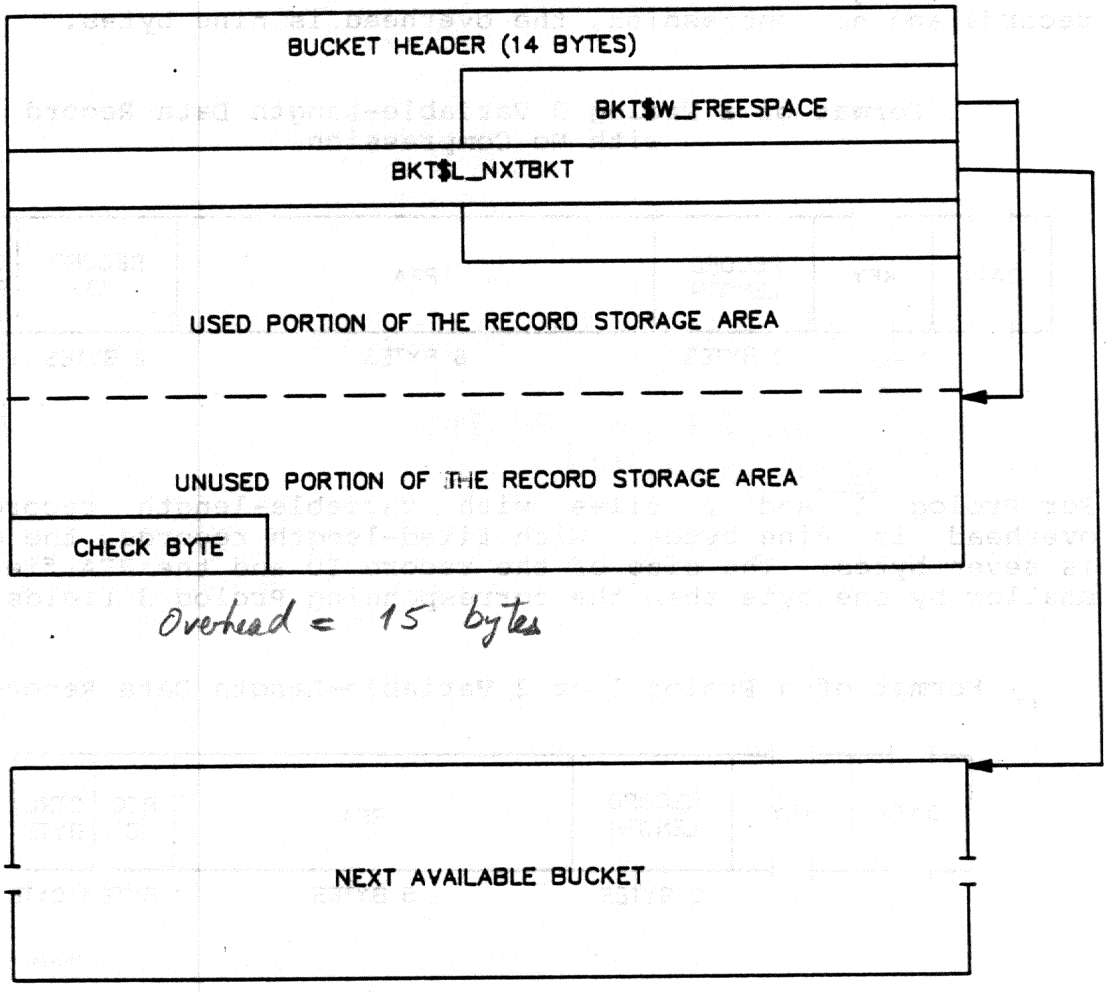
Contents of a Data Bucket (Cont.)

Field Name	Description
BKT$B_BKTCB	Bucket control bits. This field is a bit-encoded byte field and is used when RMS processes a bucket. The following bits are defined within this field.
	BKT$V_LASTBKT — Last bucket in the horizontal chain for that level. This 1-bit field starts at bit 0.
	BKT$V_ROOTBKT — Root bucket. This 1-bit field starts at bit 1.
	BKT$V_PTR_SZ — Size of the VBN pointers in this bucket. This 2-bit field starts at bit 3 and is used only for Prolog 3 index buckets.

The record storage region has two parts: a used portion occupied by records, and an unused portion. The used portion of the record storage region starts at the first byte after the bucket header area, and it ends at the byte address that is a byte less than the address stored in the BKT$W_FREESPACE field. The record structures in the bucket depend on how the bucket is used.

The unused portion of the record storage area starts at the byte address stored in the BKT$W_FREESPACE field and ends at the byte before the check byte field, which is the last byte in the bucket. Available buckets are chained together in a linked list.

The Format of a Bucket



BU-2435

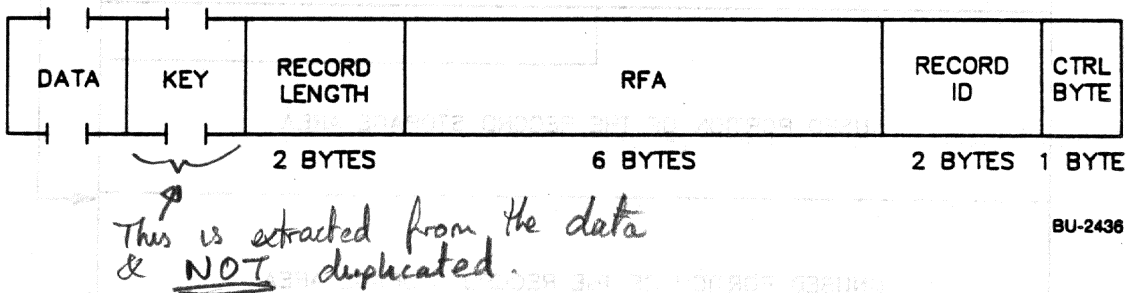
Data Record Format

The format of a data record depends on whether the file is a Prolog 3 file or a Prolog 1 or 2 file. Prolog 3 data records are the only records that allow data or key compression.

As the record of a Prolog 3 file is inserted in the bucket, the key is always placed at the front of the data record, even if there is no key compression. If the key field is in the middle of the record, it is still extracted and placed at the front of the record to improve performance. However, it is inserted at the proper place before the record is retrieved and returned to the user.

Data record overhead for Prolog 3 files with no compression and variable-length records is eleven bytes. With fixed-length records and no compression, the overhead is nine bytes.

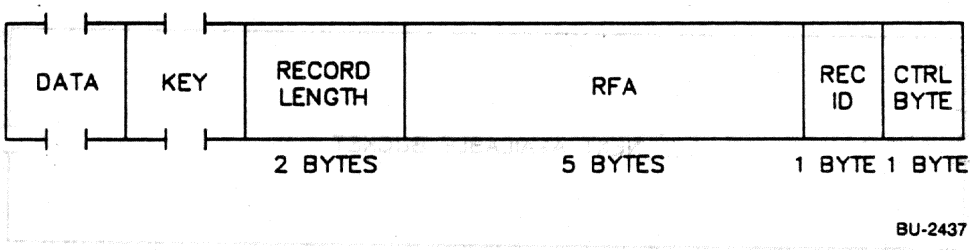
Format of a Prolog 3 Variable-Length Data Record with No Compression



This is extracted from the data & NOT duplicated.

For Prolog 1 and 2 files with variable-length records, the overhead is nine bytes. With fixed-length records, the overhead is seven bytes. The size of the record ID and the RFA fields are smaller by one byte than the corresponding Prolog 3 fields.

Format of a Prolog 1 or 2 Variable-Length Data Record



* Note that 'Guide to Tcle Applications' does not explain this in full.

The fields of a data record are described in the following table.

Field Name	Description
Control byte	Record control byte. The bits are defined as follows: 0-1 VBN pointer size 2 Deleted record flag 3 Record reference vector record flag The Analyze/RMS_File Utility (ANALYZE/RMS_FILE) can display the position and state of high-order six bits that indicate whether the record is deleted or whether it is an record reference vector (RRV). The low-order 2 bits are relatively meaningless. For example, a typical record that has not been deleted and is not an RRV has a control byte of hex 02, which has no particular significance because all data records have RFA VBN pointer sizes of four bytes.
Record ID	Record identifier. This field occupies two bytes.*
RFA	Record file address. This field serves as a record reference pointer if the record is an RRV. The RRV pointer is six bytes long and is composed of the record identifier field and the virtual block number. The VBN portion of the RFA has a fixed size of four bytes, and the record ID occupies two bytes.**
Record length	Size of the record. This 2-byte field is present only for variable-length records and fixed-length records for which either data or key compression is enabled.

* For Prolog 3 files. For Prolog 1 and 2 files, the record ID occupies one byte.

** For Prolog 3 files. For Prolog 1 and 2 files, an RRV occupies five bytes because the record ID is only one byte.

RECORD REFERENCE VECTORS (RRVs) AND BUCKET SPLITS

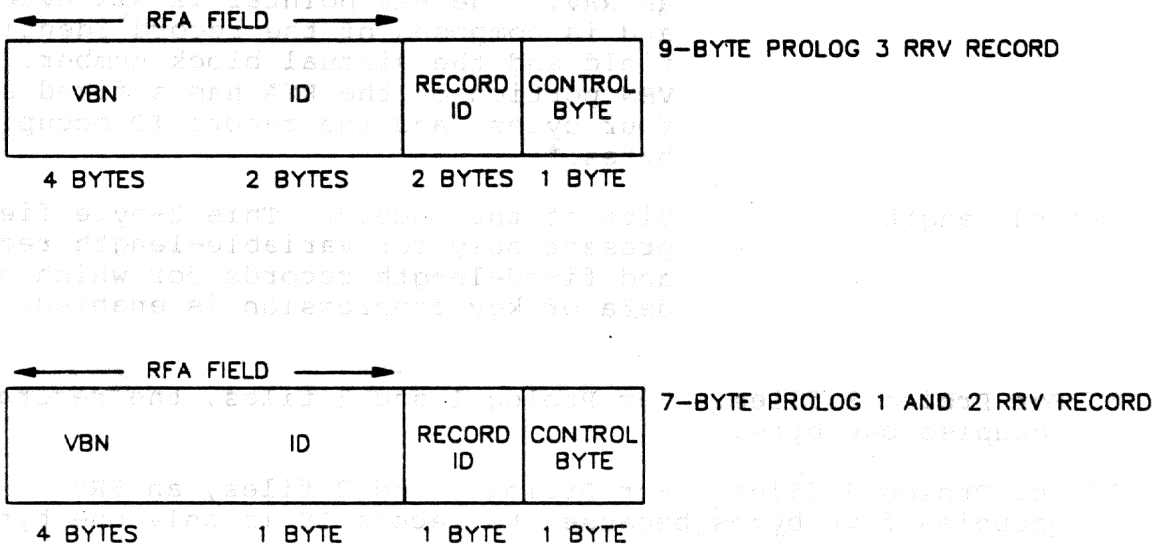
When RMS tries to insert a record into an already full bucket, some of the old records must be moved to a new bucket to make room for the new record. This process is called a bucket split.

When the records are moved out of their original bucket, RMS creates special records in the original bucket that act as pointers to the new bucket to which the records have been moved. These special records are called record reference vectors, or RRVs, and they remain in the bucket in which the records were originally inserted to act as "forwarding addresses."

An RRV is created only when a record is moved for the first time. If the record has been moved before, an RRV is not created; RMS finds the RRV in the record's original bucket and updates it with the record's new address. Even in a worst-case situation where a record has been moved many times, RMS can find the record with its RRV with only one level of indirection.

RRV records are nine bytes long in Prolog 3 files and seven bytes long in Prolog 1 or 2 files. Every data record contains an RFA portion consisting of a 1- or 2-byte record ID and a 4-byte VBN. Initially, the record ID refers to itself (that is, it contains a copy of the preceding field, the record's own ID) and the VBN is the virtual block number of the bucket in which the record is currently located.

Formats of RRV Records

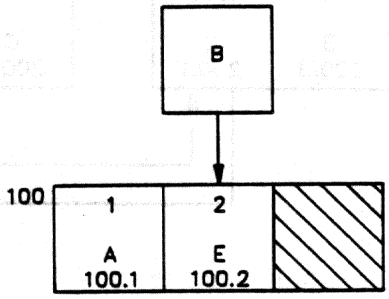


BU-2438

Simple Bucket Splits

Example 1. Simple Bucket Split

Record B is Added to the File

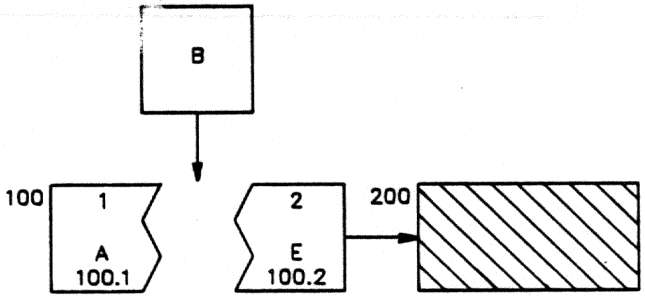


VBN = 100
Rec. ID = 1, 2

BU-2439

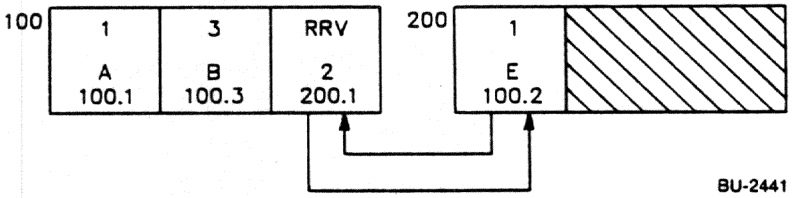
In general, RMS tries to keep about half of the records in the original bucket and move the others to a new bucket, with the bucket with the most space available containing the most data after the split.

Record B Causes a Bucket Split



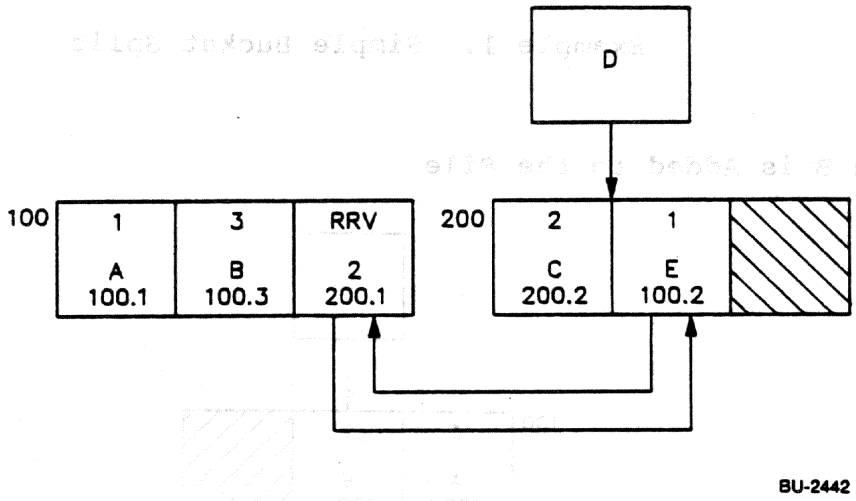
BU-2440

Record E is Moved



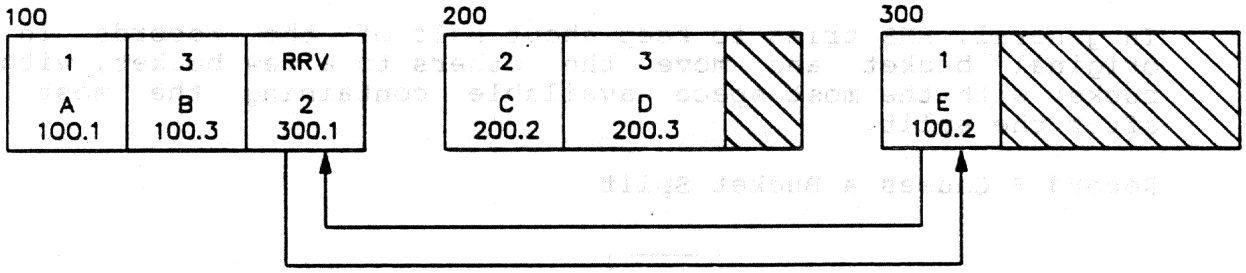
BU-2441

Records C and D are Added to the File



BU-2442

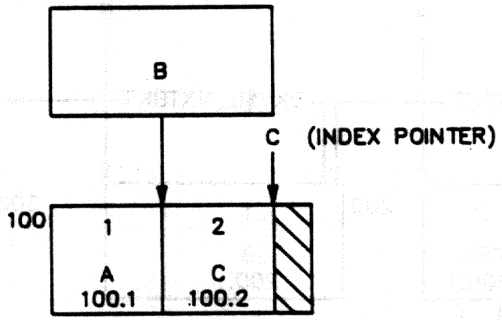
Record E is Moved to a New Bucket



BU-2443

Example 2. Multibucket Split

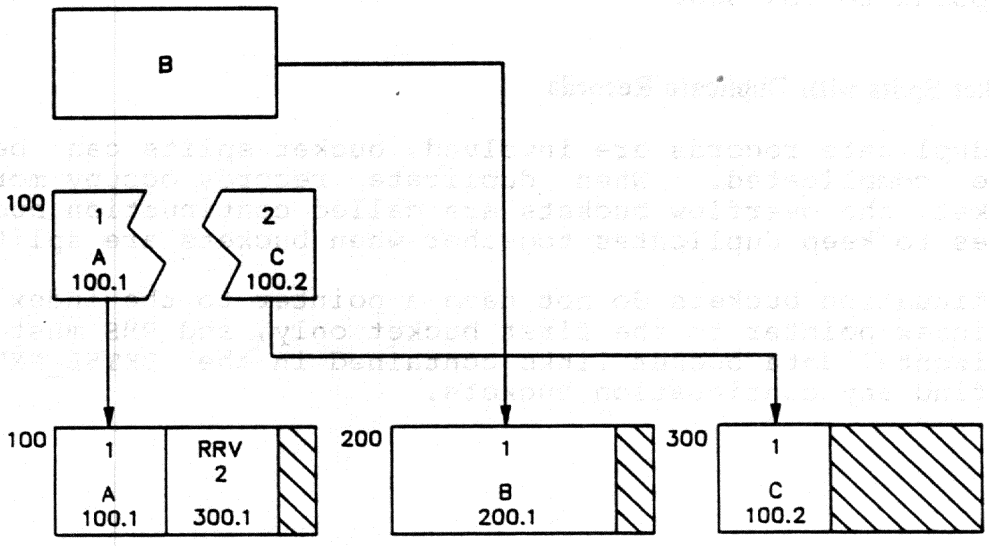
Record B is Added to the File



BU-2444

To resolve this situation, a multibucket split must take place. The existing bucket containing records A and C is split. Two new buckets are created; record B is moved into the middle bucket, and record C is moved from the old bucket into the last bucket.

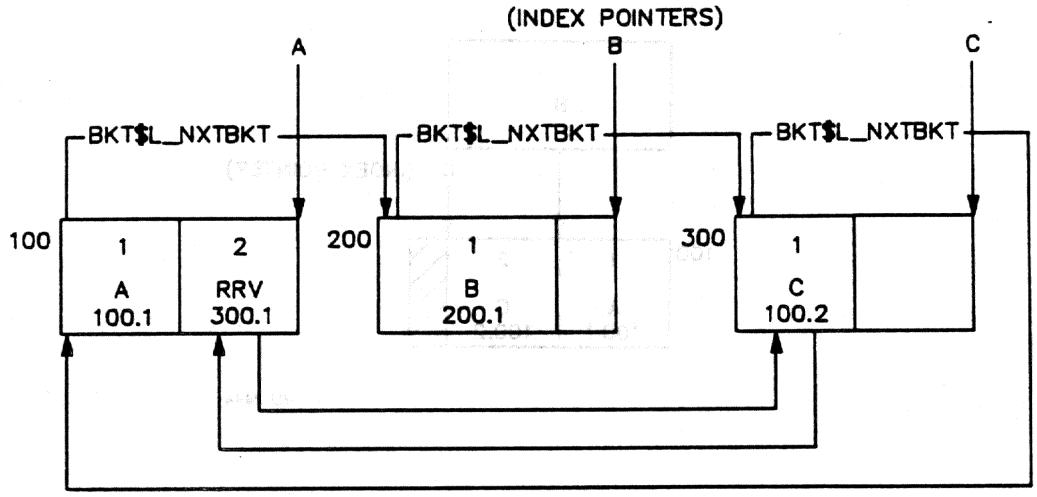
A Multibucket Split Occurs



BU-2445

An RRV record is created in the old bucket to point to the new location of record C, and record C points back to its original location. The next-bucket pointers are also updated. The last bucket in the chain points back to the first bucket.

The Bucket Structure and Index are Updated



The index structure is also updated. The index pointer to the left bucket still points to VBN 100, but its key value now becomes A. Two new index records have to be created as a result of the split. First, an index pointer with key value B must be created to point to VBN 200. A pointer with key value C must be created to point to VBN 300.

Bucket Splits with Duplicate Records

If duplicate records are involved, bucket splits can become even more complicated. When duplicate records occupy more than one bucket, the overflow buckets are called continuation buckets. RMS tries to keep duplicates together when buckets are split.

Continuation buckets do not have a pointer to the index. There is an index pointer to the first bucket only, and RMS must follow the horizontal data bucket links contained in the BKT$L_NXTBKT field to find any continuation buckets.

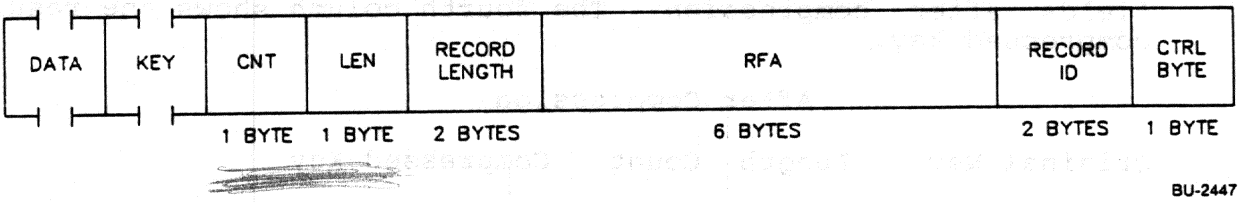
KEY AND DATA COMPRESSION (PROLOG 3)

Compression of data and the primary key is allowed for Prolog 3 data records.

Primary Key Compression

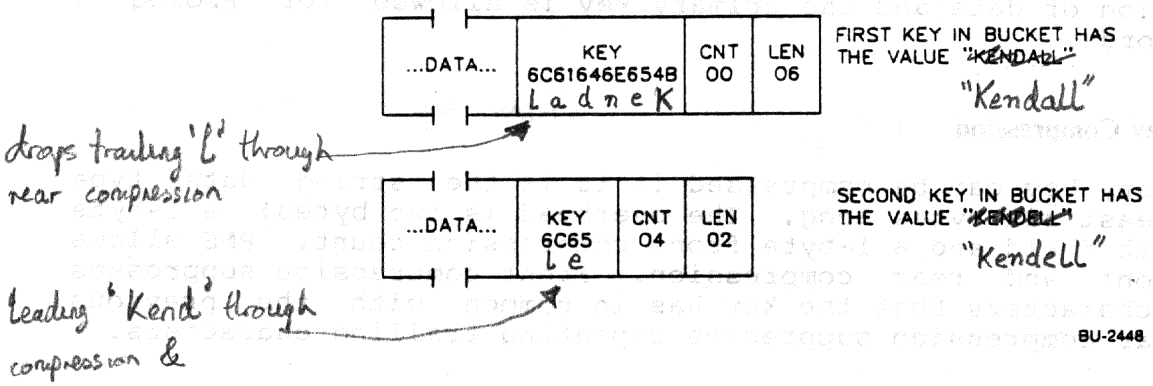
The primary key can be compressed if it is the string data type and at least six bytes long. The overhead is two bytes: a 1-byte key length field and a 1-byte front compression count. RMS allows both front and rear compression. Front compression suppresses leading characters that the key has in common with the previous key. Rear compression suppresses repeating trailing characters.

Format of a Compressed Data Record



Field Name	Description
Record length	Size of the record after compression. This field is two bytes long.
Len	Length of the key with compression. This 1-byte field gives the length of the key as it is stored on the disk. This value allows for the truncation of repeating end characters (rear compression) because the true length of the key may be obtained from the key descriptor. Any difference in size not accounted for by the front compression is due to the rear compression. The last character in the key is the character compressed. This length does not include the count field.
Cnt	Count of the front bytes with compression. This field contains the number of front characters the key has in common with the previous key; the first key in a bucket is always fully expanded although repeating characters at the end of the key are truncated.

Example 3. Two Data Records with Key Compression



The following table shows a longer example of key compression. The first column shows the original(uncompressed) keys of the file. The second and third columns show the length and count fields after compression. The fourth column shows the resulting compressed key.

	After Compression		
Original Key	Length	Count	Compressed Key
Barren*	7	0	Barren*
Barret*	2	5	t*
Barrett	1	6	t
Barron*	3	4	on*
Benson*	6	1	enson*

Total Length = LEN + CNT + X

- LEN: # bytes physically stored
- CNT: # bytes front compressed
- X: # bytes rear compressed

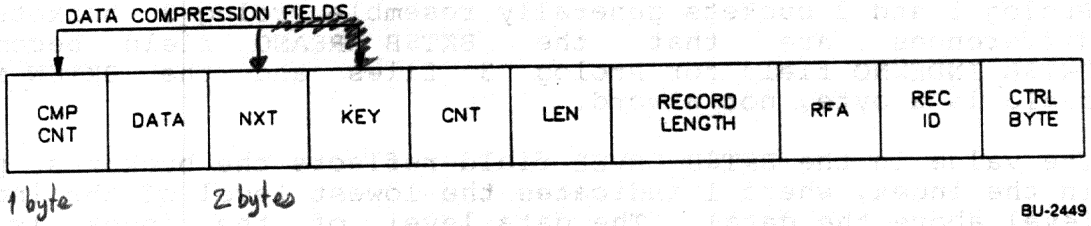
X = Total Length - (LEN + CNT)

- X: # bytes rear end compressed

Data Compression — typically string but others allowed

The data portion of a data record can also be compressed if the sequence of repeating characters is five or more characters. The overhead required for this type of compression is three bytes. The compression control information is stored within the data instead of in the record header.

Format of a Data Record with Compressed Data

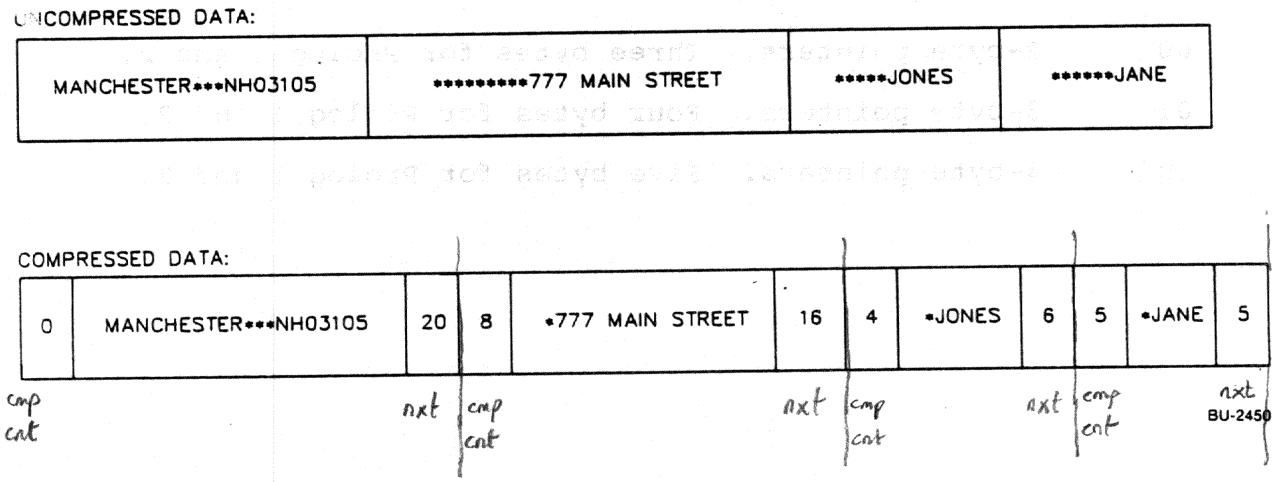


The nxt field is a count of characters in the data field that follows it. It occupies one word. If no repeating characters were found in the data section and no characters were compressed, this field contains the size of the whole data field.

The cmp cnt field is a count of the characters that were compressed. It is 1 byte long. If no characters were compressed, the cmp cnt field contains a value of 0.

Example 4 shows how a 66-byte record with repeating sequences within the data portion compressed to a 59-byte record. Note that the fields (but not the characters) of the record run right-to-left.

Example 4. Compressed Data Record



INDEX BUCKET

The index of an ISAM file is structured as a balanced tree. The buckets of the index structure are the nodes of the tree. Each bucket contains a logical range of key values.

Index buckets generally resemble one another, regardless of their position in the index. The value in the BKT$B_INDEXNO field reflects which key of reference the index bucket belongs to, where a value of 0 indicates the primary key, a 1 indicates the first alternate key, a 2 indicates the second, and so on.

Prolog 1 and 2 buckets generally resemble Prolog 3 buckets. Two differences are that the BKT$B_AREANO field becomes the BKT$B_INDEXNO field for Prolog 3 files and the BKT$W_NXTRECID field is a byte, not a word.

The value in the BKT$B_LEVEL field reflects the bucket's position in the index, where 1 indicates the lowest level of the index (the level above the data). The data level of the index is always level 0, and the root level is always the highest level.

Each level of the index is horizontally linked by the next bucket pointers. The linked list is circular because the last bucket (its address is contained in the BKT$L_NXTBKT field) points back to the first bucket. The BKT$V_LASTBKT flag is set in the last bucket to indicate that it is the last bucket in the chain and that the next bucket in the chain will be the first.

For all three prolog versions, RMS saves index bucket space by using the smallest possible field size to represent the VBN pointer of a bucket. For Prolog 3 files, however, all VBN pointers in a particular index bucket are the same size, which is the length of the largest pointer in the bucket.

Bits 3 and 4 of the bucket control byte (BKT$B_BKTCB) indicate the pointer size for a Prolog 3 index bucket. The following table shows the bit patterns and their meanings.

Bits	Meaning
00	2-byte pointers. Three bytes for Prolog 1 and 2.
01	3-byte pointers. Four bytes for Prolog 1 and 2.
10	4-byte pointers. Five bytes for Prolog 1 and 2.

Index Record Format

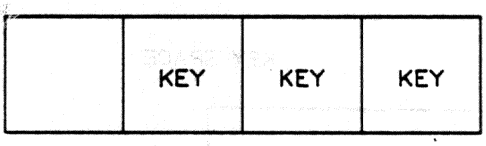
Index records for the primary key and for the upper-level alternate indexes have the same format. However, the format depends on the prolog version of the file; index records for Prolog 1 and 2 files differ from Prolog 3 files.

Prolog 3 index records have two parts: the key and the VBN pointer. They have no overhead. Keys are stored in either ascending or descending order starting at the beginning of the record storage area of the bucket. The associated VBN pointers are stored after the keys at the end of the used portion of the record storage area. In other words, the keys and the VBNs are at opposite ends of the bucket; RMS does not consider the VBN pointer to be part of the index record.

The key part of the index record includes the key and control information needed to describe the key. The key represents the highest possible key value in the bucket pointed to by the bucket pointer in the record. When RMS performs an index search, it follows the first path for which the search key is less than or equal to the key value stored in the index record. Index records may be either fixed- or variable-length.

Fixed-length index records are used for keys that have not been compressed. These records have no control information, and each key is the same length.

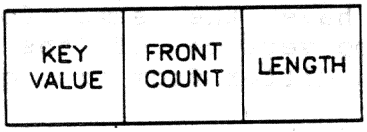
Format of the Key Part of a Fixed-Length Index Record



BU-2451

Variable-length index records are used for keys that have been compressed. All variable index records have count fields to represent the front and rear compression.

Format of the Key Part of a Variable-Length Index Record With Compression



BU-2452

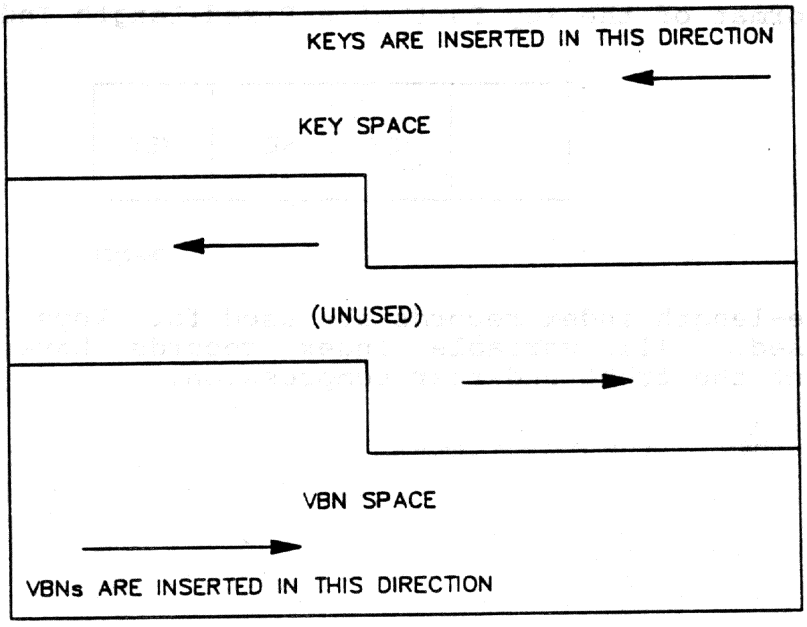
The length field contains the number of characters in the key value, including the compressed characters if index key compression is enabled. The front count field contains the number of leading characters that were compressed.

The VBN pointer associated with each index record is stored at the end of the index bucket. The size of all the VBN pointers within a bucket is the same, but it may vary from bucket to bucket. There is no overhead associated with the VBN pointer list.

In a Prolog 3 index bucket with uncompressed keys, the VBN pointer for the first key of the bucket is at the end of the bucket VBN space. As more keys and more VBNs are added to the bucket, they approach one another.

Format of the VBN Pointer List

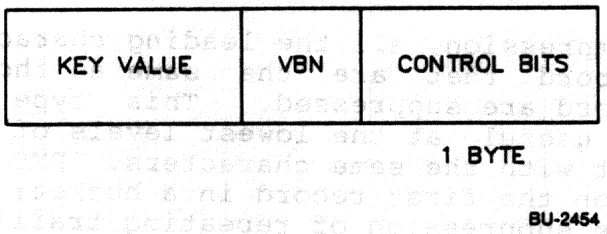
May be fixed length or variable length keys (latter only applies if index compression is used)



BU-2453

Prolog 1 and 2 records cannot be compressed. Therefore, the format of index records for the primary key and the upper-level alternate indexes for these files are identical.

Format of a Prolog 1 or 2 Index Record



The first field contains index record control bits and a pointer size. It is a 1-byte bit-encoded field. Bits 0 and 1 represent the VBN pointer size. The following table shows the bit patterns and their meanings.

Bits	Meaning
00	2-byte pointer size
01	3-byte pointer size
10	4-byte pointer size

The second field is a variable-byte bucket pointer containing the virtual block number of associated data bucket.

The third field is a variable-byte key value representing the highest key value in the corresponding data bucket.

INDEX COMPRESSION

Index records, like data records, can be compressed to save space in the file. Index compression is done exactly like data key compression. RMS compresses both repeating leading and trailing characters by default, as well as character strings with a length greater than six characters.

With front compression, all the leading characters in the key of an index record that are the same as those in the key of the preceding record are suppressed. This type of compression is particularly useful at the lowest levels of the index where many keys may start with the same characters. RMS performs no front compression on the first record in a bucket; it is fully expanded except for the suppression of repeating trailing characters. On all other keys, a field in the key overhead contains the front compression count.

With rear compression, RMS suppresses repeating trailing characters in the key. A key length field is used to determine the number of characters truncated. When the key is expanded, RMS gets the fixed-length of the key from the appropriate key descriptor.

The following table shows the index key compression of a lower-level index, where many of the keys begin and end with identical characters. The length of the string keys is fixed at eleven bytes.

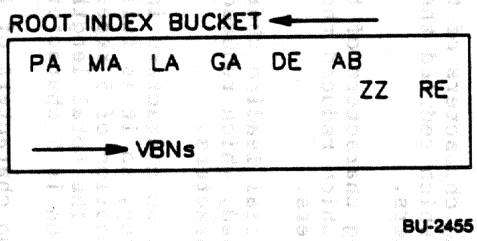
Original Key (Total length = 11)	After Compression Len	Cnt	Rear End	Compressed Key	Description
Barren*****	7	0	4	Barren*	This key is the first, so it is fully expanded except for the truncation of 4 repeating trailing characters. The total length of the key after the truncation is 7 charecters and no leading characters were compressed.
Barret*****	2	5	4	t*	The 5 leading characters of this key were compressed, which reduced the length of the key to 2 characters.
Barrette***	3	6	2	te*	The 6 leading characters of this key were compressed, which reduced the length of the key to 3 characters.
Barron*****	3	4	4	on*	The 4 leading characters of this key were compressed, which reduced the length of the key to 3 characters.
Benson*****	6	1	4	enson*	Only the initial leading character of this key was compressed, which reduced the length of the key to 6 characters.
Johns******	6	0	5	Johns*	This key has no characters in common with the preceeding key, so it is fully expanded except for the truncation of 5 repeating trailing characters. The total length of the key after the truncation is 5 characters.
Johnson****	3	5	3	on*	The 5 leading characters of this key were compressed, which reduced the length of the key to 3 characters.
Johnston***	4	5	2	ton*	The 5 leading characters of this key were compressed, which reduced the length of the key to 4 characters.
Johnstone**	2	8	1	e*	The 8 leading characters of this key were compressed, which reduced the length of the key to 2 characters.

Can give 30% improvement in CPU but greater overhead in disk I/O

BINARY VERSUS NONBINARY INDEX SEARCH

If index compression has not been enabled, RMS will do a binary search through index buckets for the requested key value, including binary and integer keys. This is why all the VBN pointers in a given index bucket for a Prolog 3 file are the same size.

Example 5. Retrieval of Record With Key Value = RA



1. If index compression is not enabled, a binary search of the index bucket is performed.

   Step

   1) LA | MA

   2) PA | RE

2. If index compression is enabled, a nonbinary search of the index bucket is performed.

	# Comparisons
Binary Search	2
Nonbinary	7

Conclusion

When might you want to enable index compression? Enable index compression if there are large string key values that, if compressed, could cache the whole index tree in memory in, for example, half the space. In this case, consider making the index bucket smaller in order to have more levels in the tree than normal to reduce the CPU time required to do the nonbinary search.

SECONDARY INDEX BUCKETS AND DATA RECORDS (SIDRs)

Secondary Index Bucket Format

The alternate index of a file is very similar in structure to the primary index. The main difference is that instead of containing data records at level 0, alternate indexes contain secondary index data records (SIDRs), which are individual pointers to primary index records with a particular alternate key value.

Alternate index buckets are similar in structure to primary index buckets. The only difference is that alternate index buckets do not have a check byte as the last byte of the bucket.

The bucket overhead of a Prolog 3 SIDR bucket differs from that of a Prolog 1 and 2 SIDR bucket. In the bucket header, the index number (BKT$B_INDEXNO) replaces the area number (BKT$B_AREANO), and the next record ID (BKT$W_NXTRECID) is a word instead of a byte.

Secondary Index Data Record Format

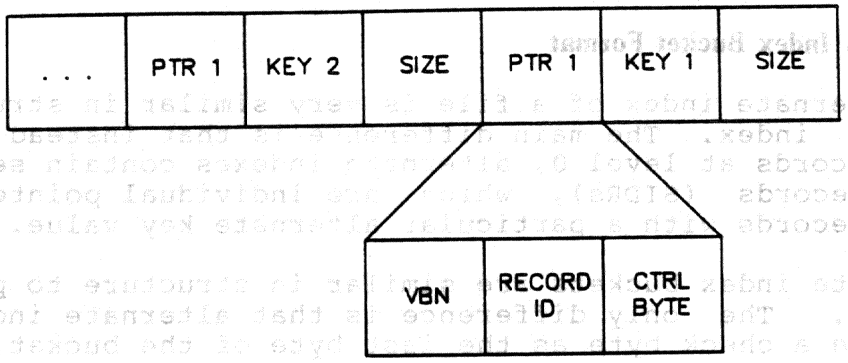
Like the primary index, the alternate index has at least two levels: an upper level containing the actual index entries for that particular key, and a data level.

Upper-level secondary index records for Prolog 1, 2, and 3 files look just like their corresponding upper-level primary index records.

However, instead of data, the lowest level of an alternate index contains a pointer array. This array is a list of pointers called SIDRs, which point back to the data level (level 0) of the primary index. SIDRs consist of a key and an RRV pointer. It is the RRV pointer that actually points back to the primary data record with that secondary key value.

Secondary data index records have a different format in Prolog 3 files than in Prolog 1 and 2 files. Keys may either be compressed or uncompressed, which is specified by the KEYNCMPR option in the key XAB.

Format of Prolog 3 Secondary Index Data Records



BU-2456

The 2-byte field reflects the size in bytes of the whole pointer array.

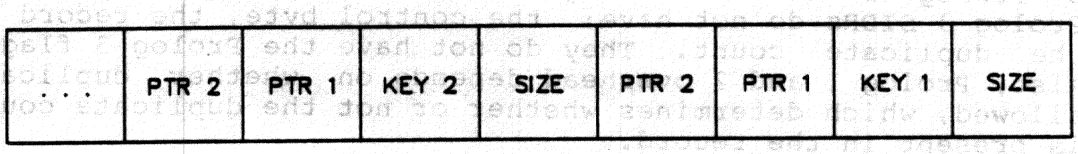
Three bits are defined within the control byte field.

Bit Name	Position	Meaning
DELETED	Bit 2	If set, the record has been deleted.
NOPTRSZE	Bit 4	If set, the record has no RRV, and the SIDR has been deleted.
FIRST_KEY	Bit 7	If set, the record is the first entry for that SIDR.

Like ordinary data records, SIDRs can also have duplicate records. Duplicates for SIDRs mean that more than one pointer exists for the same key value.

SIDR duplicates are not separate records. For each duplicate, another pointer field is appended to the SIDR. The overhead of Prolog 3 SIDRs is fixed whether or not duplicates are allowed. In this example, both key 1 and key 2 have duplicate records.

Format of Prolog 3 Secondary Index Data Records with Duplicates



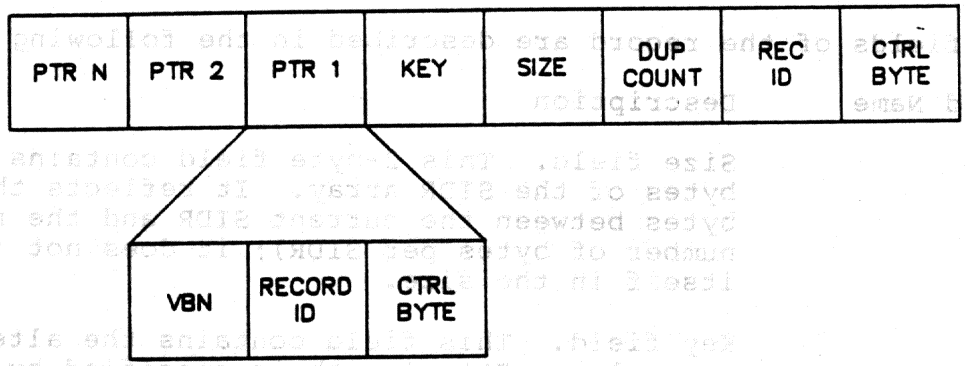
BU-2457

The fields of the record are described in the following table.

Field Name	Description
Size	Size field. This 2-byte field contains the size in bytes of the SIDR array. It reflects the number of bytes between the current SIDR and the next (the number of bytes per SIDR); it does not include itself in the size.
Key	Key field. This field contains the alternate key value. It's length is specified by the user.
Key ptr	Pointer field. This field is the RRV pointer from the alternate key back to the primary key with which it is associated. It is 4 to 7 bytes long and has three parts:
	Ctrl byte — The control byte indicates the size of the VBN and has flags that indicate whether the record has been deleted (or is a pointer to a deleted record).
	Record ID — This word contains the record ID of the primary data record that contains the given secondary key.
	VBN — This field contains the VBN of the bucket where the given primary data record is located. It can range from 2 to 4 bytes long. The combination of the record ID and the VBN forms the RFA of the primary data record that contains the secondary key.

SIDR format for a Prolog 1 or 2 file differs from SIDR format for a Prolog 3 file. Prolog 1 and 2 SIDRs have three fields that Prolog 3 SIDRs do not have: the control byte, the record ID, and the duplicate count. They do not have the Prolog 3 flags field. Also, Prolog 1 and 2 overhead depends on whether duplicates are allowed, which determines whether or not the duplicate count field is present in the record.

Format of Prolog 1 and 2 Secondary Index Data Records with Duplicate Records



BU-2458

Field Name	Description
Ctrl byte	Pointer size and data record control bits. This 1-byte field contains one of two values: 01 The 4-byte duplicates count field is present. 10 There is no duplicates count field. Prolog 3 SIDRs do not have this field.
Rec ID	Record ID. This field is 1 byte long. Prolog 3 SIDRs do not have this field.
Dup count	Duplicates count. This 4-byte field contains the number of duplicate records, unless the value in the control byte field is 10 binary, which indicates this field is not present. Prolog 3 SIDRs do not have this field. It is not supported by VAX/VMS Version 4.4, so it always contains a value of 0.
Size	Size of the rest of the array. This field is 2 bytes long, and the size does not include itself.
Key	Key field. This field contains the alternate key value. It's length is specified by the user.

Field Name	Description
Ptr	Key pointer. This field is the RRV pointer from the alternate key back to the primary key with which it is associated. It is 5 to 7 bytes long and has three parts:
Cntl byte	Bits 0 and 1 of the control byte indicate the size of the VBN. The following values are defined: 00 2 bytes 01 3 bytes 10 4 bytes Two other bits of the control byte are defined. If bit 2 is set, the record has been deleted. If bit 5 is set, the pointer has been deleted because an Update operation changed the key value.
Record ID	This byte contains the ID of the primary data record entry.
VBN	This field contains the VBN of the bucket where the given primary data record is located.

The pointer field is repeated for every duplicate record.

If the array continues into another index bucket, everything is repeated, except that the duplicates count is absent; this absence is reflected in the data record control bits.

Compression

As in the primary index, keys may also be compressed in the secondary index. The two bytes at the front of the key field indicate the key length and number of characters that were compressed at the front of the key.

PART 2. SIMULATED DATA EXAMPLE

Bucket = 1 block (all areas)

Record = Fixed-length 112 (no compression)

Bytes

0 - 109	Name	Primary index
110 - 111	Seq_No	Alternate index

Example 1 is a step-by-step illustration of an indexed file. Eleven records are entered in random order as follows:

Order of Entry

1 RAKOS

2 ASHE

3 TODD

4 JONES

5 VAIL

6 BUSH

7 EVANS

8 SACK

9 MAYO

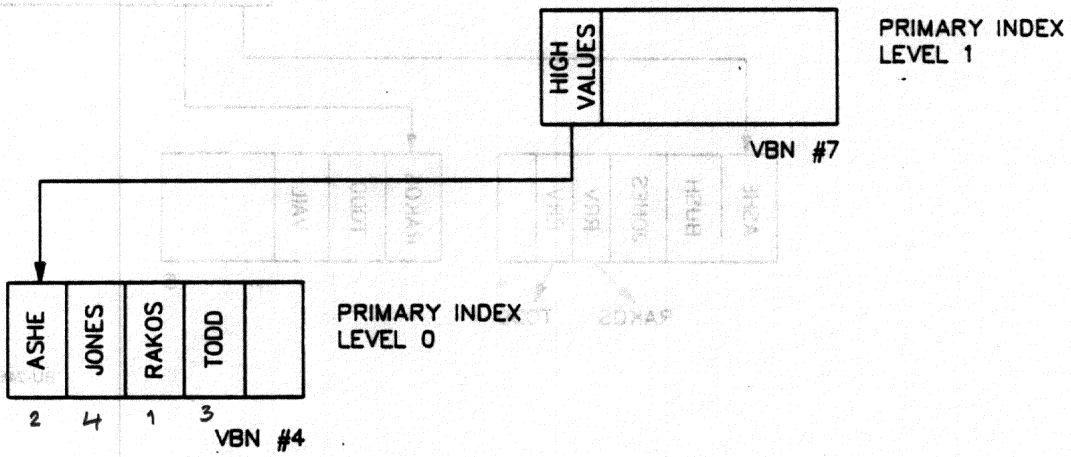
10 WOODS

11 SMITH

Bucket will contain: 14 bytes bucket header
1 byte checksum
4 * { 9 bytes record header
112 bytes data
~~112 bytes 2key~~
~~5047~~

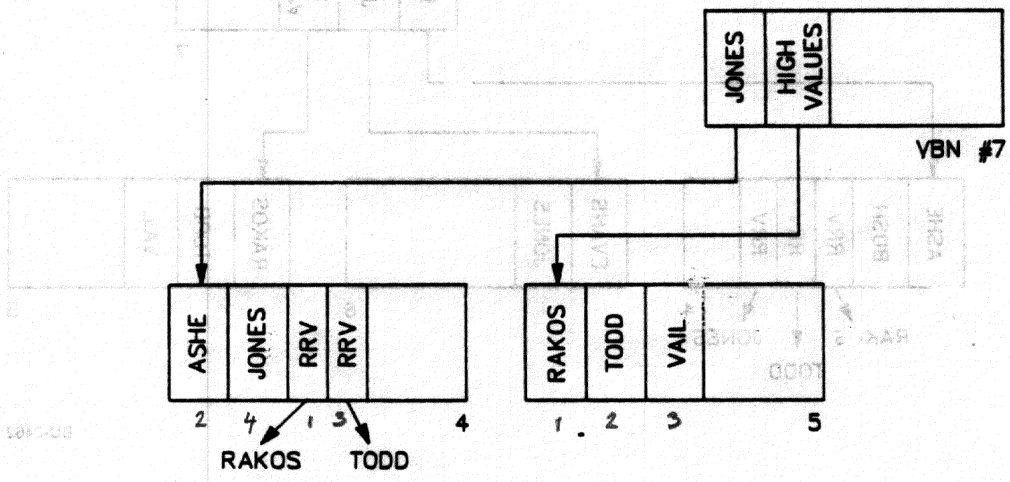
Example 1. Entering Data Records into a New Indexed File in Random Order from Program Control

Indexed File After Four Records Added



BU-2459

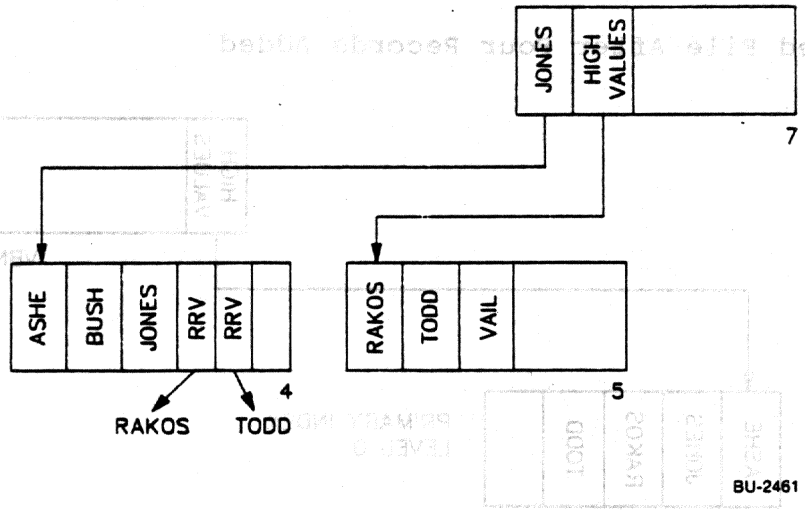
VAIL Inserted



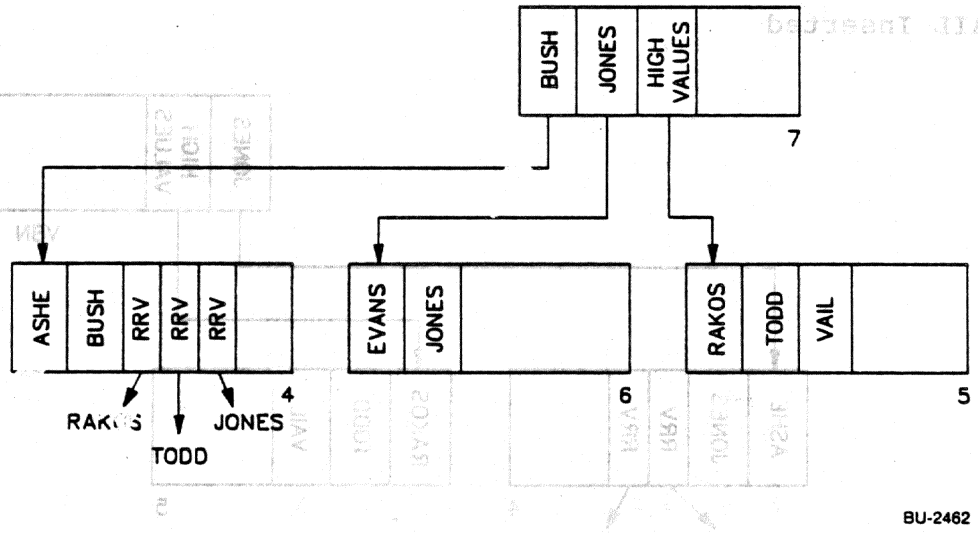
BU-2460

Split occurs because records not added in sorted order (RMS bases this on order of last two records in bucket)

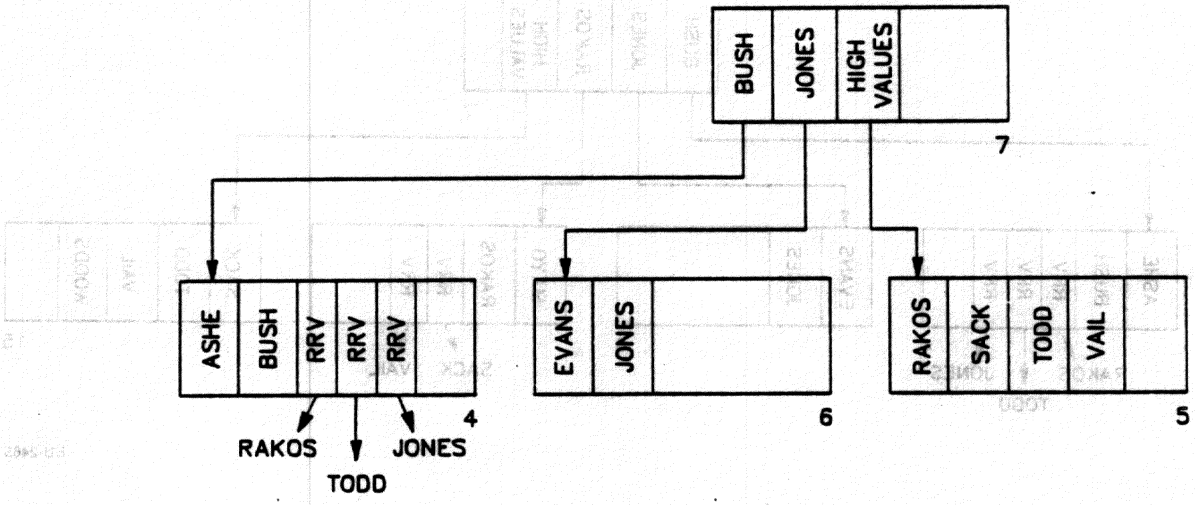
BUSH Inserted



EVANS Inserted

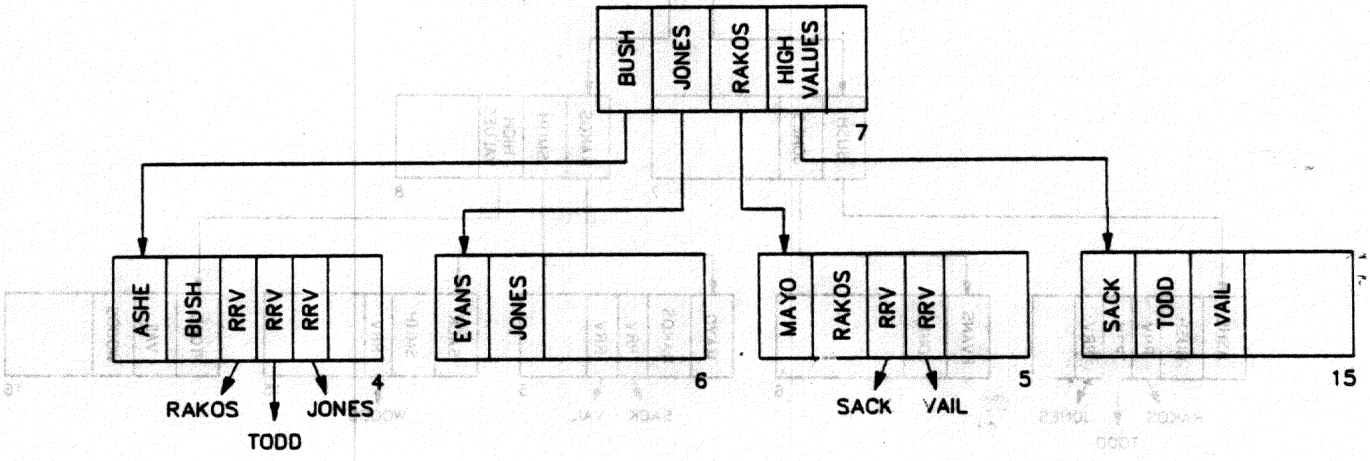


SACK Inserted



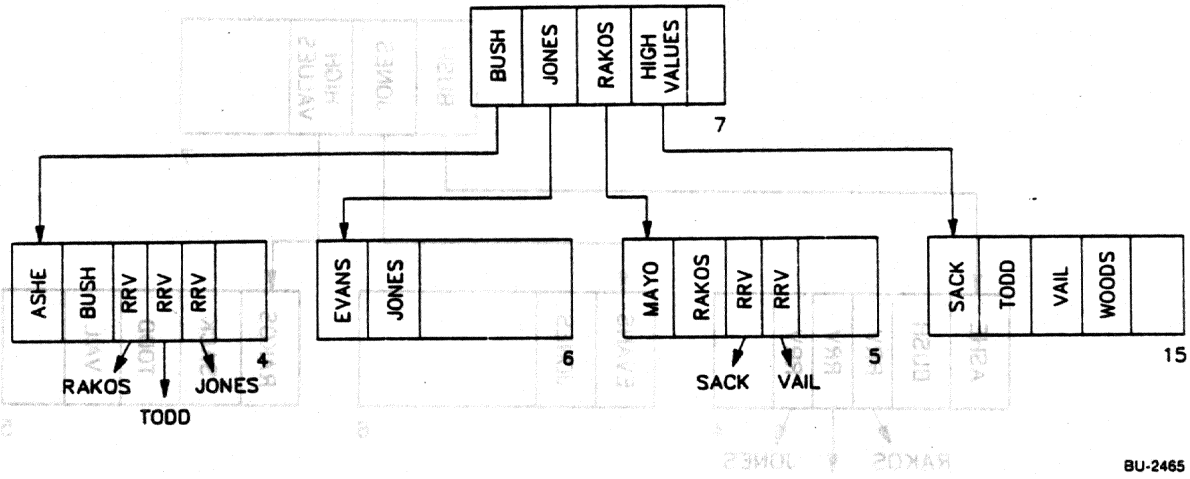
BU-2463

MAYO Inserted



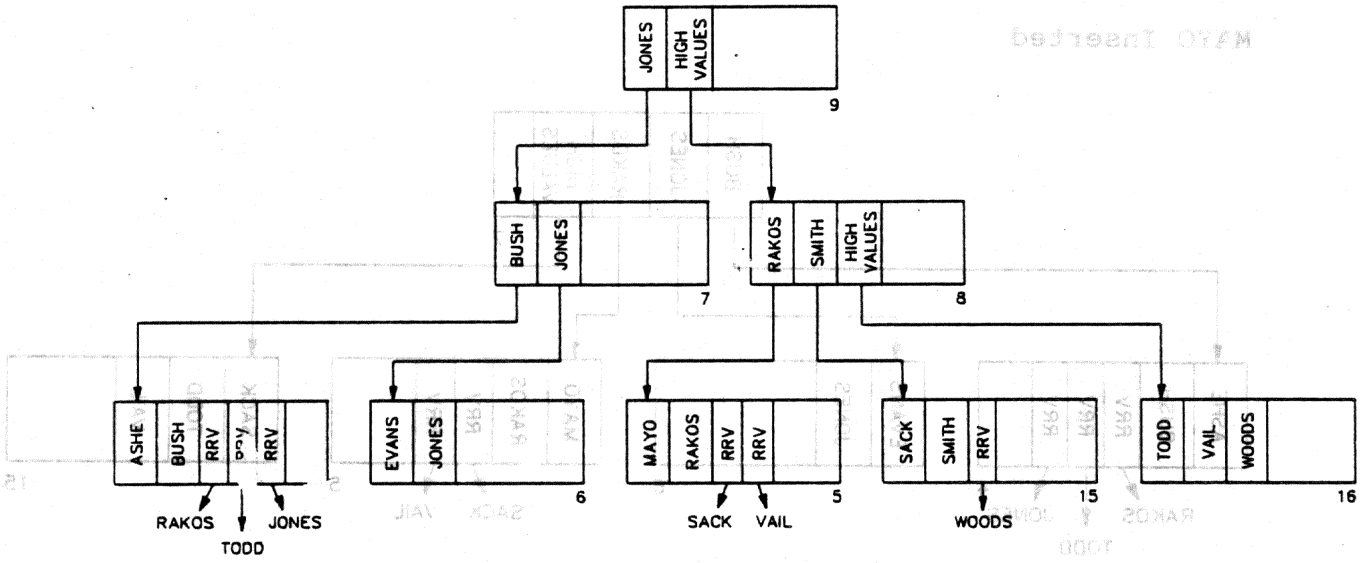
BU-2464

WOODS Inserted



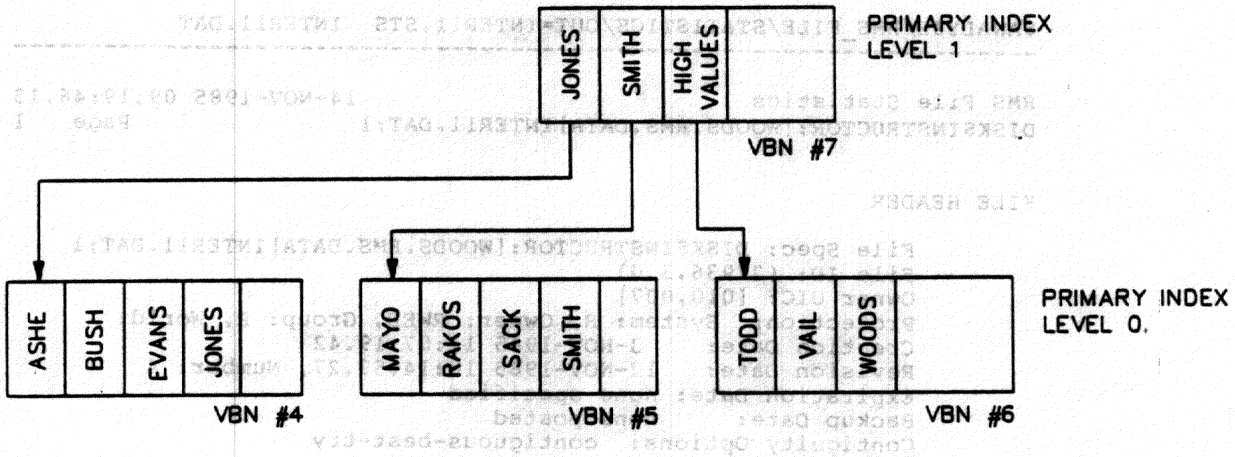
BU-2465

SMITH Inserted



BU-2466

If the eleven records in Example 1 had been entered sorted in the order of the primary key, the indexed file organization at the end of the data load would have been:



	Random Data Entry	Sorted* Data Entry
# Index Levels	2	1
# Index Buckets - Primary	3	1
# Data Buckets - Primary	5	3
# RRVs	5	0

* Sorted in order of primary key of reference

Example 2. Statistics Produced by ANALYZE for the Indexed File Created in Example 1

(Sheet 1 of 3)

```
$ANALYZE/RMS_FILE/STATISTICS/OUT=INTER11.STS  INTER11.DAT
------------------------------------------------------------------------

RMS File Statistics                              14-NOV-1985 09:19:46.13
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER11.DAT;1                 Page   1


FILE HEADER

        File Spec: DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER11.DAT;1
        File ID: (27936,3,0)
        Owner UIC: [010,007]
        Protection:  System: R, Owner: RWED, Group: R, World:
        Creation Date:    3-NOV-1985 11:07:19.42
        Revision Date:   12-NOV-1985 18:14:59.27, Number: 4
        Expiration Date: none specified
        Backup Date:     none posted
        Contiguity Options:  contiguous-best-try
        Performance Options: none
        Reliability Options: none
        Journaling Enabled:  none


RMS FILE ATTRIBUTES

        File Organization: indexed
        Record Format: fixed
        Record Attributes:   carriage-return
        Maximum Record Size: 112
        Longest Record: 112
        Blocks Allocated: 16, Default Extend Size: 1
        Bucket Size: 1
        Global Buffer Count: 0


FIXED PROLOG

        Number of Areas: 3, VBN of First Descriptor: 3
        Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 15, Blocks: 2, Used: 2, Next: 17
        Default Extend Quantity: 1
        Total Allocation: 8

STATISTICS FOR AREA #0

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #1 (VBN 3, offset %X'0040')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 7, Blocks: 6, Used: 3, Next: 10
        Default Extend Quantity: 1
        Total Allocation: 6
```

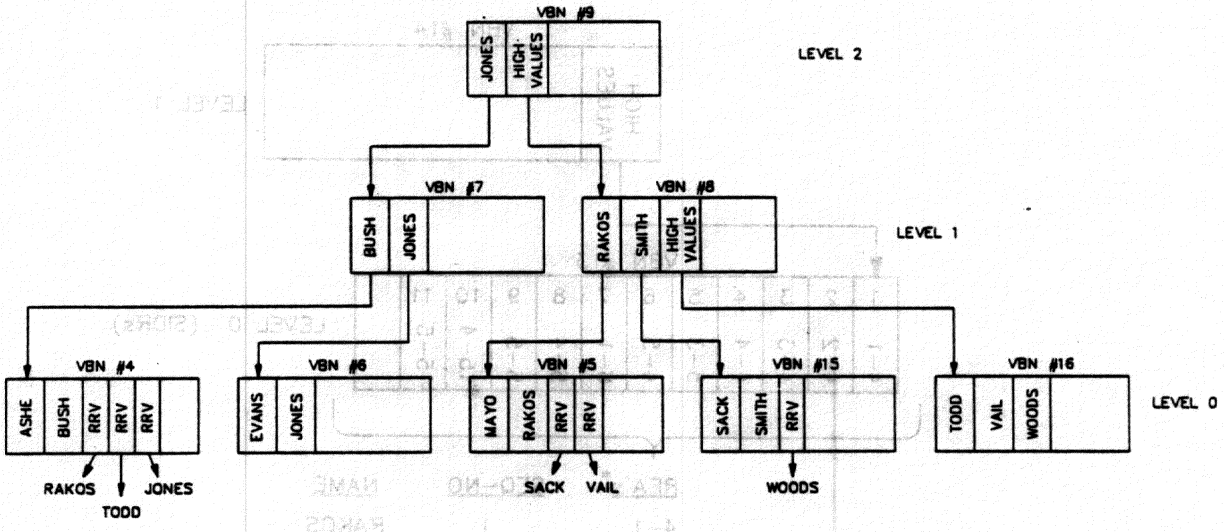
Example 2 (Sheet 2 of 3)

```
RMS File Statistics                                14-NOV-1985 09:19:46.27
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER11.DAT;1                      Page   2


STATISTICS FOR AREA #1

        Count of Reclaimed Blocks:                  0

AREA DESCRIPTOR #2 (VBN 3, offset %X'0080')

        Bucket Size: 1
        Reclaimed Bucket VBN: 0
        Current Extent Start: 13, Blocks: 2, Used: 2, Next: 15
        Default Extend Quantity: 2
        Total Allocation: 2

STATISTICS FOR AREA #2

        Count of Reclaimed Blocks:                  0

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

        Next Key Descriptor VBN: 2, Offset: %X'0000'
        Index Area: 1, Level 1 Index Area: 1, Data Area: 0
        Root Level: 2
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 9
        Key Flags:
                (0)  KEY$V_DUPKEYS     0
                (3)  KEY$V_IDX_COMPR   0
                (4)  KEY$V_INITIDX     0
                (6)  KEY$V_KEY_COMPR   0
                (7)  KEY$V_REC_COMPR   0
        Key Segments: 1
        Key Size: 110
        Minimum Record Size: 110
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:      0
        Segment Sizes:        110
        Data Type: string
        Name: "LAST_NAME"
        First Data Bucket VBN: 4

STATISTICS FOR KEY #0

        Number of Index Levels:                     2
        Count of Level 1 Records:                   5
        Mean Length of Index Entry:               112
        Count of Index Blocks:                      3
        Mean Index Bucket Fill:                   54%
        Mean Index Entry Compression:              0%
```

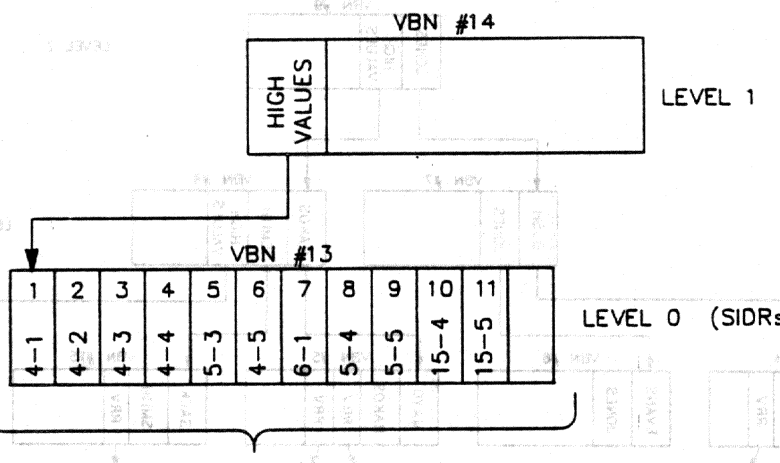
Example 2 (Sheet 3 of 3)

```
RMS File Statistics                                14-NOV-1985 09:19:46.51
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER11.DAT;1                      Page   3

        Count of Data Records:                          11
        Mean Length of Data Record:                    112
        Count of Data Blocks:                            5
        Mean Data Bucket Fill:                         56%
        Mean Data Key Compression:                      0%
        Mean Data Record Compression:                   0%
        Overall Space Efficiency:                      15%

KEY DESCRIPTOR #1 (VBN 2, offset %X'0000')

        Index Area: 2, Level 1 Index Area: 2, Data Area: 2
        Root Level: 1
        Index Bucket Size: 1, Data Bucket Size: 1
        Root VBN: 14
        Key Flags:
                (0)  KEY$V_DUPKEYS     0
                (1)  KEY$V_CHGKEYS     0
                (2)  KEY$V_NULKEYS     0
                (3)  KEY$V_IDX_COMPR   0
                (4)  KEY$V_INITIDX     0
                (6)  KEY$V_KEY_COMPR   0
        Key Segments: 1
        Key Size: 2
        Minimum Record Size: 112
        Index Fill Quantity: 512, Data Fill Quantity: 512
        Segment Positions:     110
        Segment Sizes:           2
        Data Type: unsigned word
        Name: "SEQ_NO"
        First Data Bucket VBN: 13

STATISTICS FOR KEY #1

        Number of Index Levels:                         1
        Count of Level 1 Records:                       1
        Mean Length of Index Entry:                     4
        Count of Index Blocks:                          1
        Mean Index Bucket Fill:                        4%
        Mean Index Entry Compression:                   0%

        Count of Data Records:                         11
        Mean Duplicates per Data Record:                0
        Mean Length of Data Record:                     9
        Count of Data Blocks:                           1
        Mean Data Bucket Fill:                         22%
        Mean Data Key Compression:                      0%


The analysis uncovered NO errors.
```

Primary Index Tree



Alternate Index Tree



RFA's*	SEQ-NO	NAME
4-1	1	RAKOS
4-2	2	ASHE
4-3	3	TODD
4-4	4	JONES
5-3	5	VAIL
4-5	6	BUSH
6-1	7	EVANS
5-4	8	SACK
5-5	9	MAYO
15-4	10	WOODS
15-5	11	SMITH

*RFA = VBN # - ID #

BU-2469

MODULE 6
RMS UTILITIES

Major Topics

Part 3. Introduction
- ANALYZE/RMS_FILE utility
- Measuring run-time performance

Part 4. Evaluating/Utilizing
- ANALYZE statistics output
- RTL LIB$SHOW_TIMER output

Source

Guide to VAX/VMS File Applications — Chapter 10 (Section 10.1)
VAX/VMS ANALYZE/RMS-File Utility Reference Manual

MODULE 6
RMS UTILITIES

Major Topics

Part 3 Introduction
- ANALYZE/RMS_FILE utility
- Measuring run-time performance

Part 4 Evaluating/Utilizing
- ANALYZE statistics output
- RTL LIBSSHOW_TIMER output

Source
Guide to VAX/VMS File Applications — Chapter 10 (Section 10.1)
VAX/VMS ANALYZE/RMS-File Utility Reference Manual

PART 3. INTRODUCTION

Analyzing File Structure

ANALYZE/RMS_FILE

- Produces a statistical report on the file structure

  $ ANALYZE/RMS_FILE/STATISTICS file-spec

- Produces a summary report containing file structure information

  $ ANALYZE/RMS_FILE/SUMMARY file-spec

- Produces a summary of the file structure and checks its integrity

  $ ANALYZE/RMS_FILE/CHECK file-spec

  This option will be covered in Module 17, Data Recovery.

- Allows you to explore the structure of a file interactively. For example, in an indexed file you can follow the whole path from prolog block 1 down to a data record, using any index, and dumping any buckets you want on the way down.

  $ ANALYZE/RMS_FILE/INTERACTIVE file-spec

  This option will be covered in Module 16, RMS Utilities (Part 6).

- ANALYZE/RMS FILE/FDL can be used to create an FDL file from an existing data file.

NOTE

Use the following qualifier for output to be copied to a file rather than to be sent to the SYS$OUTPUT default.

/OUTPUT=file-spec

Measuring Run-Time Performance

The five available statistics for measuring run-time performance are listed below.

Shown on line	Description
ELAPSED = hhhh:mm:ss.cc	Elapsed real time
CPU = hhhh:mm:ss.cc	Elapsed CPU time
BUFIO = nnnn	Count of buffered I/O operations
DIRIO = nnnn	Count of direct I/O operations
PAGEFAULTS = nnnn	Count of page faults

LIB$SHOW_TIMER returns the times and counts accumulated since the last call to LIB$INIT_TIMER. By default, when neither code nor action-rtn is specified in the call, LIB$SHOW_TIMER writes to SYS$OUTPUT a line giving the information listed above.

LIB$STAT_TIMER returns to its caller one of five available statistics. Unlike LIB$SHOW_TIMER, which formats the values for output, LIB$STAT_TIMER returns the value as an unsigned longword or quadword.

The elapsed time is returned in the system quadword format. Therefore, the receiving area should be eight bytes long. All other returned values are longwords.

The following summary illustrates the differences between LIB$SHOW_TIMER and LIB$STAT_TIMER.

Code	Statistic	Format for LIB$SHOW_TIMER	Format for LIB$STAT_TIMER
1	Elapsed real time	hhhh:mm:ss.cc	Quadword in system time format
2	Elapsed CPU time	hhhh:mm:ss.cc	Longword in 10-millisecond increments
3	Count of buffered I/O operations	nnnn	Longword
4	Count of direct I/O operations	nnnn	Longword
5	Count of page faults	nnnn	Longword

Example 1. Measuring Performance with RTL Routines

This example illustrates the use of Run-Time Library routines to measure the performance of a program. The routines are used to collect and display information on the resource usage of the program.

BASIC

- The call to LIB$INIT_TIMER stores the current values of the program statistics to be measured. Since no storage block was specified, the values are kept in storage space maintained by the RTL routines.
- Initializing the array consumes system resources.
- The call to LIB$SHOW_TIMER obtains the accumulated times and counts since the call to LIB$INIT_TIMER. Because no code or action-routine has been specified, the statistics are output to the terminal in ASCII format.

Example 1 (Sheet 1 of 5)

```
10 !                                                    LIBTIMER.BAS
   !
   !        This program illustrates the use of the RTL
   !        performance measurement routines.
   !
    EXTERNAL LONG FUNCTION LIB$INIT_TIMER, LIB$SHOW_TIMER
    DECLARE LONG result, j, k
    DIMENSION  LONG iarray (99%, 99%)
   !
    result = LIB$INIT_TIMER()
    CALL LIB$STOP (result BY VALUE) IF (result AND 1%)=0%
   !
   !        Initialize the array
    FOR k = 0% TO 99% STEP 1%
        FOR j = 0% to 99% STEP 1%
            iarray (j,k) = 1%
        NEXT j
    NEXT k
   !
    result = LIB$SHOW_TIMER()
    PRINT 'Usage values after array initialization'
    CALL LIB$STOP (result BY VALUE) IF (result AND 1%)=0%
   !
    END
$ BASIC LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER
 Elapsed: 00:00:00.36  CPU: 0:00:00.21  BUFIO: 0  DIRIO: 0  FAULTS: 1
Usage values after array initializaton
```

COBOL

- The call to LIB$INIT_TIMER stores the current values of the program statistics to be measured. Since no storage block was specified, the values are kept in storage space maintained by the RTL routines.

- Initializing the array consumes system resources. Notice that many page faults are incurred because the array was not accessed in the most efficient order.

- The call to LIB$SHOW_TIMER obtains the accumulated times and counts since the call to LIB$INIT_TIMER. Since no code or action-routine has been specified, the statistics are output to the terminal in ASCII format.

Example 1 (Sheet 2 of 5)

```
                                                    LIBTIMER.COB
      *
      IDENTIFICATION DIVISION.
      *
      PROGRAM-ID. LIBTIMER.
      *
      *   This program illustrates the use of the RTL
      *   performance measurement routines.
      *
      DATA DIVISION.

      WORKING-STORAGE SECTION.
      01  ARRAY.
          02  DIM1        OCCURS 100 TIMES.
          03  DIM2        OCCURS 100 TIMES.
          05  IARRAY      PIC S9(9) COMP.
      01  I               PIC S9(9) COMP.
      01  J               PIC S9(9) COMP.
      01  RESULT          PIC S9(9) COMP.

      PROCEDURE DIVISION.
      BEGIN.
      *
          CALL 'LIB$INIT_TIMER' GIVING RESULT.
          IF RESULT IS FAILURE CALL 'LIB$STOP' USING BY VALUE RESULT.
      *
      *   Initialize the table
          PERFORM VARYING J FROM 1 BY 1 UNTIL I > 100
                  PERFORM VARYING I FROM 1 BY 1 UNTIL I > 100
                          MOVE 0 TO IARRAY(I,J)
                  END-PERFORM
          END-PERFORM
      *
          CALL 'LIB$SHOW_TIMER' GIVING RESULT.
          DISPLAY 'Usage values after table initialization'.
          IF RESULT IS FAILURE CALL 'LIB$STOP' USING BY VALUE RESULT.
          STOP RUN.
```

```
$ COBOL LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER
 ELAPSED: 00:00:00.11  CPU: 0:00:00.06  BUFIO: 0  DIRIO: 0  FAULTS: 80
Usage values after table initialization
$
```

FORTRAN

- The call to LIB$INIT_TIMER stores the current values of the program statistics to be measured. Since no storage block was specified, the values are kept in storage space maintained by the RTL routines.

- The call to LIB$SHOW_TIMER obtains the accumulated times and counts since the call to LIB$INIT_TIMER. Since no code or action-routine has been specified, the statistics are output to the terminal in ASCII format.

- Initializing the array consumes system resources. Notice that many page faults are incurred because the array was not accessed in the most efficient order.

Example 1 (Sheet 3 of 5)

```
 1      C**                                                LIBTIMER.FOR
 2      C
 3      C   This program illustrates the use of the RTL
 4      C   performance measurement routines.
 5      C
 6          IMPLICIT        INTEGER*4  (A-Z)
 7          DIMENSION       IARRAY (100, 100)
 8      C
 9          RESULT = LIB$INIT_TIMER()
10          IF (.NOT. RESULT) CALL LIB$STOP(%VAL( RESULT))
11      C
12      C   Initialize the array
13          DO  J=1, 100
14              DO  K=1, 100
15                  IARRAY(J,K)= 1
16              ENDDO
17          ENDDO
18      C
19          RESULT= LIB$SHOW_TIMER()
20          IF (.NOT. RESULT) CALL LIB$STOP(%VAL( RESULT))
21          TYPE *, 'Usage values after array initialization'
22      C
23          END

$ FORTRAN LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER
 ELAPSED: 00:00:00.21  CPU: 0:00:00.19  BUFIO: 0  DIRIO: 0  FAULTS: 80
Usage values after array initialization
```

- The call to LIB$INIT_TIMER stores the current values of the program statistics to be measured. Since no storage block was specified, the values are kept in storage space maintained by the RTL routines.

- The INDEX and AOBLEQ statements are used to implement two loops from 1 to 100 to initialize the array.

- The call to LIB$SHOW_TIMER obtains the accumulated times and counts, since the call to LIB$INIT_TIMER. No code or action-routine has been specified, so the statistics are output to the terminal in ASCII format.

Example 1 (Sheet 4 of 5)

```
                                                        LIBTIMER.MAR
;
;   This program illustrates the use of the RTL
;   performance measurement routines.
;
        .TITLE  LIBTIMER
        .MACRO  LIB_ERROR ?NO_ERROR
        BLBS    R0,NO_ERROR
        PUSHL   R0
        CALLS   #1,G^LIB$STOP
        NO_ERROR:
        .ENDM   LIB_ERROR
;
        .PSECT  NOSHARED_DATA   PIC, NOEXE, LONG
IARRAY:     .BLKL   100*100
;
        .PSECT  CODE            PIC, SHR, NOWRT, LONG
        .ENTRY  BEGIN   ^M<R2,R3>
;
        CALL    #0, G^LIB$INIT_TIMER
        LIB_ERROR
;
;
        MOVL    #1, R1
30$:        MOVL    #1, R2
;
;   Do IARRAY (J,K)=1; J=1,100; K=1,100
40$:        INDEX   R2, #1, #100, #25, #0, R3
        INDEX   R1, #1, #100,  #1, R3, R3
;
        MOVL    #1, IARRAY-104[R3]
;
        AOBLEQ  #100, R2, 40$
        AOBLEQ  #100, R1, 30$
;
        CALLG   #0, G^LIB$SHOW_TIMER
        LIB_ERROR
        OUTPUT T=<Usage values after computation:>
;
        MOVL    #SS$_NORMAL, R0
        RET
        .END    BEGIN
```

```
$ MACRO LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER
 ELAPSED: 00:00:00.56  CPU: 0:00:00.21  BUFIO: 0  DIRIO: 0  FAULTS: 22
Usage values after computation:
```

PASCAL

- The call to LIB$INIT_TIMER stores the current values of the program statistics to be measured. Since no storage block was specified, the values are kept in storage space maintained by the RTL routines.

- Initializing the array consumes system resources. Notice that many page faults are incurred.

- The call to LIB$SHOW_TIMER obtains the accumulated times and counts since the call to LIB$INIT_TIMER. Since no code or action-routine has been specified, the statistics are output to the terminal in ASCII format.

Example 1 (Sheet 5 of 5)

```
 1       PROGRAM libtimer (INPUT, OUTPUT);
 2
 3       (*   LIBTIMER.PAS
 4       (*   This program illustrates the use of the RTL
 5       (*   performance measurement routines.
 6
 7       VAR  iarray: ARRAY [1..100, 1..100] OF INTEGER;
 8            count1, count2: INTEGER;
 9            lib_stat: INTEGER;
10
11       FUNCTION LIB$INIT_TIMER(  VAR handle_adr:  INTEGER
12            := %IMMED 0): INTEGER; EXTERN;
13
14       FUNCTION LIB$SHOW_TIMER(
15            handler_adr: INTEGER := %IMMED 0;
16            code: INTEGER := %IMMED 0; %IMMED [UNBOUND]
17            FUNCTION action_rtn :INTEGER := %IMMED 0;
18            %IMMED user_arg: INTEGER := %IMMED 0):
19            INTEGER; EXTERN;
20
21       PROCEDURE LIB$STOP( %IMMED cond_value: INTEGER); EXTERN;
22
23       BEGIN
24          lib_stat:= LIB$INIT_TIMER;
25          IF NOT ODD( lib_stat) THEN LIB$STOP( lib_stat);
26
27          (* Initialize the array *)
28          FOR count1:= 1 TO 100 DO
29             FOR count2:= 1 TO 100 DO
30            iarray[count1,count2]:= 1;
31
32          lib_stat:= LIB$SHOW_TIMER;
33          WRITELN( 'Usage values after array initialization');
34          IF NOT ODD( lib_stat) THEN LIB$STOP( lib_stat)
35       END.
```

```
$ PASCAL LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER
 ELAPSED: 00:00:00.15  CPU: 0:00:00.14  BUFIO: 0  DIRIO: 0  FAULTS: 78
Usage values after array initialization
```

Example 2. Comparison of FDL Calculations for Loading Data by RMS_Puts versus FAST_Convert

(Sheet 1 of 2)

```
RMS_PUTS
Initial     = 1000
Additional =    0

Not loaded in order of key.
-------------------------------------------------
$TYPE BACK1.FDL
-------------------------------------------------

TITLE

IDENT    " 1-JAN-1986 20:08:21    VAX-11 FDL Editor"

SYSTEM
        SOURCE                    VAX/VMS

FILE
        NAME                      "INDXBACK.DAT"
        ORGANIZATION              indexed

RECORD
        CARRIAGE_CONTROL          carriage_return
        FORMAT                    fixed
        SIZE                      50

AREA 0
        ALLOCATION                180
        BEST_TRY_CONTIGUOUS       yes
        BUCKET_SIZE               3
        EXTENSION                 45

AREA 1
        ALLOCATION                3
        BEST_TRY_CONTIGUOUS       yes
        BUCKET_SIZE               3
        EXTENSION                 3

KEY 0
        CHANGES                   no
        DATA_AREA                 0
        DATA_FILL                 100
        DATA_KEY_COMPRESSION      no
        DATA_RECORD_COMPRESSION   no
        DUPLICATES                no
        INDEX_AREA                1
        INDEX_COMPRESSION         no
        INDEX_FILL                100
        LEVEL1_INDEX_AREA         1
        NAME                      "SEQ_NO"
        PROLOG                    3
        SEG0_LENGTH               5
        SEG0_POSITION             0
        TYPE                      string
```

```
FAST_CONVERT
Initial      = 1000
Additional   =    0
-------------------------------------------------
$TYPE BACK2.FDL
-------------------------------------------------

TITLE

IDENT    " 1-JAN-1986 20:12:50    VAX-11 FDL Editor"

SYSTEM
        SOURCE                    VAX/VMS

FILE
        NAME                      "INDXBACK.DAT"
        ORGANIZATION              indexed

RECORD
        CARRIAGE_CONTROL          carriage_return
        FORMAT                    fixed
        SIZE                      50

AREA 0
        ALLOCATION                123
        BEST_TRY_CONTIGUOUS       yes
        BUCKET_SIZE               3
        EXTENSION                 30

AREA 1
        ALLOCATION                3
        BEST_TRY_CONTIGUOUS       yes
        BUCKET_SIZE               3
        EXTENSION                 3

KEY 0
        CHANGES                   no
        DATA_AREA                 0
        DATA_FILL                 100
        DATA_KEY_COMPRESSION      no
        DATA_RECORD_COMPRESSION   no
        DUPLICATES                no
        INDEX_AREA                1
        INDEX_COMPRESSION         no
        INDEX_FILL                100
        LEVEL1_INDEX_AREA         1
        NAME                      "SEQ_NO"
        PROLOG                    3
        SEG0_LENGTH               5
        SEG0_POSITION             0
        TYPE                      string
```

Example 2 (Sheet 2 of 2)

```
$DIFF BACK1.FDL BACK2.FDL

************
File DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK1.FDL;1
     1    TITLE    "BACK1 - INITIAL 1000; 0 - LOADED RMS_PUTS NOT IN ORDER"
     2
     3    IDENT    " 1-JAN-1986 20:08:21    VAX-11 FDL Editor"
     4
******
File DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK2.FDL;1
     1    TITLE    "BACK2 - INITIAL 1000; 0 - FAST_CONVERT LOAD"
     2
     3    IDENT    " 1-JAN-1986 20:12:50    VAX-11 FDL Editor"
     4
************
************
File DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK1.FDL;1
    18            ALLOCATION              180
    19            BEST_TRY_CONTIGUOUS     yes
    20            BUCKET_SIZE             3
    21            EXTENSION               45
    22
******
File DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK2.FDL;1
    18            ALLOCATION              123
    19            BEST_TRY_CONTIGUOUS     yes
    20            BUCKET_SIZE             3
    21            EXTENSION               30
    22
************

Number of difference sections found: 2
Number of difference records found: 7

DIFFERENCES /IGNORE=()/MERGED=1-
    DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK1.FDL;1-
    DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK2.FDL;1
```

Example 3. Loading BACKWARDS Data Using CONVERT

```
$CONVERT/SORT/STATISTICS/FDL=BACK2  BACKWARDS.DAT  INDXBACK.DAT

CONVERT Statistics
Number of Files Processed:          1
Total Records Processed:         1000    Buffered I/O Count:           29
Total Exception Records:            0    Direct I/O Count:            221
Total Valid Records:             1000    Page Faults:                 264
Elapsed Time:          0 00:00:15.90    CPU Time:          0 00:00:04.82


$DIR/FULL INDEXBACK.DAT

Directory DISK$INSTRUCTOR:[WOODS.RMS.COURSE]

INDXBACK.DAT;1                File ID:  (31322,14,0)
Size:          128/128        Owner:    [VMS,WOODS]
Created:   1-JAN-1986 20:35   Revised:   1-JAN-1986 20:35 (1)
Expires:   <None specified>   Backup:    <No backup done>
File organization:  Indexed, Prolog: 3, Using 1 key
                    In 2 areas
File attributes:    Allocation: 128, Extend: 30, Maximum bucket size: 3,
                       Global buffer count: 0, No version limit
                    Contiguous best try
Record format:      Fixed length 50 byte records
Record attributes:  Carriage return carriage control
File protection:    System:R, Owner:RWED, Group:R, World:
Access Cntrl List:  None

Total of 1 file, 128/128 blocks.
```

Example 4. Loading FRONTWARDS Data Using CONVERT

```
$CONVERT/NOSORT/STATISTICS/FDL=FRONT1  FRONTWARDS.DAT INDXFRONT.DAT

CONVERT Statistics
Number of Files Processed:          1
Total Records Processed:         1000    Buffered I/O Count:          10
Total Exception Records:            0    Direct I/O Count:            72
Total Valid Records:             1000    Page Faults:                161
Elapsed Time:       0 00:00:03.33    CPU Time:          0 00:00:02.20
```

```
$DIR/FULL INDXFRONT.DAT

Directory DISK$INSTRUCTOR:[WOODS.RMS.COURSE]

INDXFRONT.DAT;1               File ID:  (31155,15,0)
Size:          128/128        Owner:    [VMS,WOODS]
Created:   1-JAN-1986 20:30   Revised:   1-JAN-1986 20:30 (1)
Expires:   <None specified>   Backup:    <No backup done>
File organization:  Indexed, Prolog: 3, Using 1 key
                            In 2 areas
File attributes:    Allocation: 128, Extend: 30, Maximum bucket size: 3,
                         Global buffer count: 0, No version limit
                    Contiguous best try
Record format:      Fixed length 50 byte records
Record attributes:  Carriage return carriage control
File protection:    System:R, Owner:RWED, Group:R, World:
Access Cntrl List:  None

Total of 1 file, 128/128 blocks.
```

Example 5. Loading BACKWARDS Data Using RMS_Puts in Program Control

```
$SH STATUS

  Status on   1-JAN-1986 20:46:30.46     Elapsed CPU :   0 00:16:36.68
  Buff. I/O :     41497    Cur. ws. :     450    Open files :         2
  Dir. I/O :      36682    Phys. Mem. :   251    Page Faults :   125478

$RUN LOADINDX

Enter name of INPUT SEQUENTIAL file:  BACKWARDS.DAT       } RMS PUTS
Enter name of FDL file for OUTPUT:  BACK1.FDL

 ELAPSED: 00:01:01.49  CPU: 0:00:13.89  BUFIO: 15  DIRIO: 1868  FAULTS: 110

 VS  FAST_CONVERT              0:04.82          29          221          264

$SH STATUS

  Status on   1-JAN-1986 20:47:34.54     Elapsed CPU :   0 00:16:51.09
  Buff. I/O :     41520    Cur. ws. :     300    Open files :         2
  Dir. I/O :      38573    Phys. Mem. :   243    Page Faults :   125667

$DIR/FULL INDXBACK.DAT

Directory DISK$INSTRUCTOR:[WOODS.RMS.COURSE]

INDXBACK.DAT;2                File ID:  (31581,39,0)
Size:          368/368        Owner:    [VMS,WOODS]
Created:   1-JAN-1986 20:46   Revised:   1-JAN-1986 20:47 (1)
Expires:   <None specified>   Backup:    <No backup done>
File organization:  Indexed, Prolog: 3, Using 1 key
                    In 2 areas
File attributes:    Allocation: 368, Extend: 45, Maximum bucket size: 3,
                    Global buffer count: 0, No version limit
                    Contiguous best try
Record format:      Fixed length 50 byte records
Record attributes:  Carriage return carriage control
File protection:    System:R, Owner:RWED, Group:R, World:
Access Cntrl List:  None
```

Example 6. Analyze Statistics for BACKWARDS RMS_Puts Data Load

(Sheet 1 of 2)

```
$ANALYZE/RMS_FILE/STATISTICS/OUT=INDXBACK.ANL  INDXBACK.DAT
------------------------------------------------------------------------------
RMS File Statistics                              1-JAN-1986 20:52:05.27
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXBACK.DAT;2              Page 1

FILE HEADER

        File Spec: DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXBACK.DAT;2
        File ID: (31581,39,0)
        Owner UIC: [010,007]
        Protection:  System: R, Owner: RWED, Group: R, World:
        Creation Date:    1-JAN-1986 20:46:31.95
        Revision Date:    1-JAN-1986 20:47:23.70, Number: 1
        Expiration Date: none specified
        Backup Date:     none posted
        Contiguity Options:  contiguous-best-try
        Performance Options: none
        Reliability Options: none
        Journaling Enabled:  none


RMS FILE ATTRIBUTES

        File Organization: indexed
        Record Format: fixed
        Record Attributes:   carriage-return
        Maximum Record Size: 50
        Longest Record: 50
        Blocks Allocated: 368, Default Extend Size: 45
        Bucket Size: 3
        Global Buffer Count: 0


FIXED PROLOG

        Number of Areas: 2, VBN of First Descriptor: 2
        Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 2, offset %X'0000')

        Bucket Size: 3
        Reclaimed Bucket VBN: 0
        Current Extent Start: 323, Blocks: 46, Used: 30, Next: 353
        Default Extend Quantity: 45
        Total Allocation: 364

STATISTICS FOR AREA #0

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #1 (VBN 2, offset %X'0040')

        Bucket Size: 3
        Reclaimed Bucket VBN: 0
        Current Extent Start: 181, Blocks: 4, Used: 3, Next: 184
        Default Extend Quantity: 3
        Total Allocation: 4

STATISTICS FOR AREA #1
```

Example 6 (Sheet 2 of 2)

```
RMS File Statistics                                1-JAN-1986 20:52:05.41
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXBACK.DAT;2            Page   2


        Count of Reclaimed Blocks:                  0

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

        Index Area: 1, Level 1 Index Area: 1, Data Area: 0
        Root Level: 1
        Index Bucket Size: 3, Data Bucket Size: 3
        Root VBN: 181
        Key Flags:
                (0)  KEY$V_DUPKEYS     0
                (3)  KEY$V_IDX_COMPR   0
                (4)  KEY$V_INITIDX     0
                (6)  KEY$V_KEY_COMPR   0
                (7)  KEY$V_REC_COMPR   0
        Key Segments: 1
        Key Size: 5
        Minimum Record Size: 5
        Index Fill Quantity: 1536, Data Fill Quantity: 1536
        Segment Positions:        0
        Segment Sizes:            5
        Data Type: string
        Name: "SEQ_NO"
        First Data Bucket VBN: 3

STATISTICS FOR KEY #0

        Number of Index Levels:                     1
        Count of Level 1 Records:                 108
        Mean Length of Index Entry:                 7
        Count of Index Blocks:                      3
        Mean Index Bucket Fill:                   50%
        Mean Index Entry Compression:              0%

        Count of Data Records:                   1000
        Mean Length of Data Record:                50
        Count of Data Blocks:                     342
        Mean Data Bucket Fill:                    39%
        Mean Data Key Compression:                 0%
        Mean Data Record Compression:              0%

        Overall Space Efficiency:                 26%

The analysis uncovered NO errors.

ANALYZE/RMS_FILE/STATISTICS/OUT=INDXBACK.ANL INDXBACK.DAT
```

(No. records * rec. len) / Total allocation

Example 7. Analyze Statistics for FRONTWARDS RMS_Puts Data Load

(Sheet 1 of 2)

```
$ANALYZE/RMS_FILE/STATISTICS/OUT=INDXFRONT.ANL  INDXFRONT.DAT
------------------------------------------------------------------------
RMS File Statistics                                1-JAN-1986 20:50:03.56
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXFRONT.DAT;2            Page  1


FILE HEADER

        File Spec: DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXFRONT.DAT;2
        File ID: (31365,15,0)
        Owner UIC: [010,007]
        Protection:  System: R, Owner: RWED, Group: R, World:
        Creation Date:     1-JAN-1986 20:44:32.52
        Revision Date:     1-JAN-1986 20:45:01.50, Number: 1
        Expiration Date: none specified
        Backup Date:      none posted
        Contiguity Options:  contiguous-best-try
        Performance Options: none
        Reliability Options: none
        Journaling Enabled:  none


RMS FILE ATTRIBUTES

        File Organization: indexed
        Record Format: fixed
        Record Attributes:   carriage-return
        Maximum Record Size: 50
        Longest Record: 50
        Blocks Allocated: 140, Default Extend Size: 33
        Bucket Size: 3
        Global Buffer Count: 0


FIXED PROLOG

        Number of Areas: 2, VBN of First Descriptor: 2
        Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 2, offset %X'0000')

        Bucket Size: 3
        Reclaimed Bucket VBN: 0
        Current Extent Start: 1, Blocks: 136, Used: 122, Next: 123
        Default Extend Quantity: 33
        Total Allocation: 136

STATISTICS FOR AREA #0

        Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #1 (VBN 2, offset %X'0040')

        Bucket Size: 3
        Reclaimed Bucket VBN: 0
        Current Extent Start: 137, Blocks: 4, Used: 3, Next: 140
        Default Extend Quantity: 3
        Total Allocation: 4

STATISTICS FOR AREA #1
```

Example 7 (Sheet 2 of 2)

```
RMS File Statistics                          1-JAN-1986 20:50:03.82
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDXFRONT.DAT;2              Page  2

        Count of Reclaimed Blocks:              0

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

        Index Area: 1, Level 1 Index Area: 1, Data Area: 0
        Root Level: 1
        Index Bucket Size: 3, Data Bucket Size: 3
        Root VBN: 137
        Key Flags:
                (0)  KEY$V_DUPKEYS     0
                (3)  KEY$V_IDX_COMPR   0
                (4)  KEY$V_INITIDX     0
                (6)  KEY$V_KEY_COMPR   0
                (7)  KEY$V_REC_COMPR   0
        Key Segments: 1
        Key Size: 5
        Minimum Record Size: 5
        Index Fill Quantity: 1536, Data Fill Quantity: 1536
        Segment Positions:        0
        Segment Sizes:            5
        Data Type: string
        Name: "SEQ_NO"
        First Data Bucket VBN: 3

STATISTICS FOR KEY #0

        Number of Index Levels:                  1
        Count of Level 1 Records:               40
        Mean Length of Index Entry:              7
        Count of Index Blocks:                   3
        Mean Index Bucket Fill:                 19%
        Mean Index Entry Compression:            0%

        Count of Data Records:                1000
        Mean Length of Data Record:             50
        Count of Data Blocks:                  120
        Mean Data Bucket Fill:                  97%
        Mean Data Key Compression:               0%
        Mean Data Record Compression:            0%

        Overall Space Efficiency:               69%


The analysis uncovered NO errors.

ANALYZE/RMS_FILE/STATISTICS/OUT=INDXFRONT.ANL INDXFRONT.DAT
```

Example 8. Comparison of Statistics for RMS_Puts Data Load -- BACKWARDS versus FRONTWARDS RMS_Puts Data Load

ANALYZE STATISTICS FOR KEY #0

	INDXBACK.DAT	INDXFRONT.DAT
Number of Index Levels:	1	1
Count of Level 1 Records:	108	40
Mean Length of Index Entry:	7	7
Count of Index Blocks:	3	3
Mean Index Bucket Fill:	50%	19%
Mean Index Entry Compression:	0%	0%
Count of Data Records:	1000	1000
Mean Length of Data Record:	50	50
Count of Data Blocks:	342	120
Mean Data Bucket Fill:	39%	97%
Mean Data Key Compression:	0%	0%
Mean Data Record Compression:	0%	0%
Overall Space Efficiency:	26%	69%

ANALYZE FILE SUMMARY

Bucket size	3	3
Number blocks allocated	368	140
Default extent	45	33
Area 0 - Number unused	16	14
1 - Number unused	1	1

RTL LIB$SHOW_TIMER STATISTICS for LOADINDX:

	INDXBACK		INDXFRONT	
	No DFW*	DFW*	No DFW*	DFW*
Elapsed time	01:01.49	0:46:54	0:30:12	0:22.34
CPU time	0:13.89	0:11.94	0:10.16	0:04.50
Buffered I-O	15	17	7	13
Direct I-O	1868	978	1364	395
Page faults	110	112	112	119

* Deferred write

Percent
Difference = (Poorer way - Better way)/Poorer way * 100.0

Calculation of percent difference in DIO between INDXBACK_DFW and INDXFRONT_DFW

$$\frac{(978 - 395)}{978} * 100.0 = 78.85\% \text{ reduction in DIO}$$

59.61

Example 9. Real-Data ANALYZE/RMS/STAT Output Using Data Compression Option

(Sheet 1 of 2)

```
RMS File Statistics
Page  1

FILE HEADER

   File Spec: DISKXYZ:[USERA]REALDATA2.DAT;2
   File ID: (31365,15,0)
   Owner UIC: [120,007]
   Protection:  System: R, Owner: RWED, Group: , World:
   Creation Date:     1-JAN-1986 20:44:32.52
   Revision Date:     31-JAN-1986 20:45:01.50, Number: 161
   Expiration Date: none specified
   Backup Date:      none posted
   Contiguity Options:  contiguous-best-try
   Performance Options: none
   Reliability Options: none
   Journaling Enabled:  none


RMS FILE ATTRIBUTES

   File Organization: indexed
   Record Format: fixed
   Record Attributes:   carriage-return
   Maximum Record Size: 140
   Longest Record: 140
   Blocks Allocated: 3003, Default Extend Size: 729
   Bucket Size: 3
   Global Buffer Count: 0


FIXED PROLOG

   Number of Areas: 2, VBN of First Descriptor: 2
   Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 2, offset %X'0000')

   Bucket Size: 3
   Reclaimed Bucket VBN: 0
   Current Extent Start: 1, Blocks: 2922, Used: 1010, Next: 1011
   Default Extend Quantity: 729
   Total Allocation: 2922

STATISTICS FOR AREA #0

   Count of Reclaimed Blocks:              0

AREA DESCRIPTOR #1 (VBN 2, offset %X'0040')

   Bucket Size: 3
   Reclaimed Bucket VBN: 0
   Current Extent Start: 2923, Blocks: 81, Used: 48, Next: 2971
   Default Extend Quantity: 21
   Total Allocation: 81
```

Example 9 (Sheet 2 of 2)

```
RMS File Statistics
Page  2

STATISTICS FOR AREA #1

   Count of Reclaimed Blocks:              0


KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

   Index Area: 1, Level 1 Index Area: 1, Data Area: 0
   Root Level: 2
   Index Bucket Size: 3, Data Bucket Size: 3
   Root VBN: 2089
   Key Flags:
           (0)  KEY$V_DUPKEYS    0
           (3)  KEY$V_IDX_COMPR  1
           (4)  KEY$V_INITIDX    0
           (6)  KEY$V_KEY_COMPR  1
           (7)  KEY$V_REC_COMPR  1
   Key Segments: 1
   Key Size: 60
   Minimum Record Size: 60
   Index Fill Quantity: 1536, Data Fill Quantity: 1536
   Segment Positions:       0
   Segment Sizes:           60
   Data Type: string
   Name: "MY_VERY_OWN_KEY"
   First Data Bucket VBN: 3

STATISTICS FOR KEY #0

   Number of Index Levels:                 2
   Count of Level 1 Records:             336
   Mean Length of Index Entry:            62
   Count of Index Blocks:                 48
   Mean Index Bucket Fill:                75%
   Mean Index Entry Compression:          16%

   Count of Data Records:                8763
   Mean Length of Data Record:            140
   Count of Data Blocks:                 1008
   Mean Data Bucket Fill:                 94%
   Mean Data Key Compression:             38%
   Mean Data Record Compression:          90%

   Overall Space Efficiency:              79%

   The analysis uncovered NO errors.


   ANALYZE/RMS_FILE/STATISTICS/OUT=REALDATA2.ANL  REALDATA2.DAT
```

MODULE 7
FILE SHARING AND RECORD/BUCKET LOCKING: SEQUENTIAL, RELATIVE, AND INDEXED FILES

Major Topics

- File sharing
- Record locking
- Alternative record locking controlling options

Source

Guide to VAX/VMS File Applications — Chapter 7 (Sections 7.1-7.2)

FILE SHARING

File sharing for READs (GETs) is supported for all RMS file organizations without restriction.

As of VAX/VMS Version 4.4, WRITE (PUTs, UPDATEs, DELETEs) sharing is also supported for all RMS file organizations without restriction. Prior to Version 4.4, write sharing for sequential files was restricted to fixed-length 512-byte records.

The combination of values specified for file sharing and file access by the initial accessor of the file determines the type of file access that will be allowed for subsequent users. In addition to the comparison of the file access values that subsequent accessors specify with the file sharing values of the initial accessor, the values specified for file sharing by subsequent accessors must be compatible with the values specified for file access by the initial accessor.

Initial File Access and Subsequent File Sharing

Initial Accessor Access	Subsequent Accessor Sharing
ACCESS GET*	SHARING GET*
ACCESS DELETE	SHARING DELETE
ACCESS PUT	SHARING PUT
ACCESS UPDATE	SHARING UPDATE
ACCESS TRUNCATE**	No access allowed

* May be implied by default
** Specifying ACCESS TRUNCATE disables file sharing

Because the initial accessor can specify multiple access values, a subsequent accessor whose sharing values match all of the initial accessor's access values is allowed access; when the subsequent accessor specifies a sharing value that the initial accessor did not specify as an access value (an exception is ACCESS GET, which is implied), access will be denied.

File Access and Sharing Options

	File Access Open Keywords	File Sharing Open Keywords
BASIC	ACCESS READ WRITE MODIFY SCRATCH APPEND (sequential) DEFAULT - MODIFY	ALLOW READ WRITE (locks against delete or scratch) MODIFY (unlimited access) NONE DEFAULT If access READ - READ and other access - NONE
COBOL	INPUT OUTPUT EXTEND I-O DEFAULT No default - must specify one of above keywords.	ALLOWING READERS WRITERS UPDATERS ALL NO OTHERS DEFAULT If input - READERS all other modes - NO OTHERS
FORTRAN	READ ONLY DEFAULT If above keyword not present in OPEN - read-write	SHARED DEFAULT If above keyword omitted, no sharing
PASCAL	History OLD NEW UNKNOWN READONLY DEFAULT - NEW (except if external file opened using RESET or EXTEND - OLD)	Sharing READONLY READWRITE NONE DEFAULT If History = READONLY - READONLY If History ≠ READONLY - NONE

```
          File Access                    File Sharing
          FAB$B_FAC                      FAB$B_SHR

MACRO     FAB$V                          FAB$V
          FAB$M  _GET                    FAB$M  _SHRGET
                 _PUT                           _SHRPUT
                 _DEL                           _SHRDEL
                 _UPD                           _SHRUPD
                 _TRN (sequential)              _MSE    multi-streaming
                 _BIO Block I/O                 _UPI    user-interlock
                 _BRO Block I/O                 _NIL

          DEFAULT                        DEFAULT
                    $V                             $V
                    $M_GET               If $B_FAC = FAB$M_GET
                                                     _SHRGET

                                                   $V          $V
                                         If $B_FAC = FAB$M _PUT then $M_NIL
                                                     _DEL
                                                     _UPD
                                                     _TRN
```

RECORD LOCKING

RMS provides a record-locking capability for relative files, indexed files, and sequential files. This capability affords control over operations when two or more streams or processes are accessing the file simultaneously. Record locking ensures that when a program is adding, deleting, or modifying a record on a given stream, another stream or process cannot access the same record.

Shared sequential files have one capability not common to shared relative or indexed files. This capability is associated with Append operations. If more than one process is connected to the end-of-file of a shared sequential file and is sequentially putting records to the end of the file, RMS guarantees the records will be appended in temporal order, even in a VAXcluster environment.

To prevent simultaneous updates of the same records, RMS uses the VAX/VMS lock manager to lock a record that has been read and will be modified later by the same program.

Whether or not record locking can occur on a file depends on the file access and file sharing specified by the initial accessor and whether another user has successfully opened the file for shared access.

- RMS automatically locks records when one or more of the processes with SHARED FILE ACCESS has opened the file for access other than read (for example, WRITE/PUT, MODIFY/UPDATE, DELETE).

- RMS handles automatic locking of an entire bucket for the short period of time required to access the record initially. This automatic locking also occurs later when the contents of the bucket are modified. In the interim, the record remains locked, but other records in the bucket can be accessed and modified as required. See Appendix A for a summary of the specific points in RMS-coded instructions for index files when locking is done at record, bucket, or f le level.

- In the case of shared sequential files, buck t locking is done on a 'virtual' bucket, the size of which is determined by the first accessor's multiblock count (MBC). A common buffer size has to be found to be used over all the processes sharing access to a particular file. Since there is nothing in the file header for a sequential file which could be used for this purpose, it is set equal to the first accessor's MBC. The typical system default for MBC is 16 blocks. When VMS 4.4 and 4.3 act as partners in accessing a shared sequential file on a VAXcluster, a multiblock count of one will be assumed by RMS.

NOTE

Shared sequential files may not be used in a VAXcluster environment in which VMS 4.2 or earlier is being run on any of the nodes. They may be accessed concurrently from nodes running VMS 4.3 and any other node running VMS 4.4 on the same cluster, but for this purpose VMS 4.4 will operate in a 4.3 fallback mode.

- The default RMS record locking actions can be modified to varying extents on a per operation basis, as described in the next section.

- Records can be locked automatically or manually. RMS handles automatic record locking transparently. The default is automatic record locking, which is appropriate when you are dealing with a lock on a single record at a time. Manual record locking requires additional effort on the programmer's part. Use it when dealing with locks on multiple records at one time.

Automatic Record Locking (RMS Default)

For a process which opened the file specifying WRITE ACCESS, each GET or FIND operation locks a record. RMS assumes an UPDATE or DELETE operation may follow. Lock is released when:

- the next record is accessed (FIND, GET or PUT),
- the current record is updated or deleted,
- an I/O error occurs,
- the record stream is disconnected,
- the file is closed, or
- a Free or Release service (or its equivalent) is called.

For a process which opened the file specifying READONLY access but sharing WRITE, each read (GET or FIND) operation will still incur some locking activities but to a lesser extent than if the process had specified WRITE access. A query lock will be used for each read operation. The query lock is briefly taken out in order to find out whether the record is locked against the process, and is then immediately released.* The query lock call to the lock manager requests "concurrent read" access rather than "exclusive" access.

* The only case in which a readonly process will hold onto a record lock is if the special locking option RAB$V_REA was set in the RAB$L_ROP field (see the next section).

Manual Record Unlocking

When enabled, any record that would have automatically been unlocked will remain locked until:

- a Free or Release service is called,
- the record stream is disconnected, or
- the file is closed.

Some examples of when manual control over unlocking of records may be useful are as follows:

- Multiple records must be modified in a single transaction. The programmer does not want any of the updates to be done unless all the updates are successfully completed. The programmer is responsible for restoring the original contents of any record already updated if the update for any other record within the transaction is unsuccessful.
- While the updates are being done to several related records, the programmer does not want a other user to be able to access any of the records in the related set.

There are three ways manual record unlocking can be enabled.

1. Set option directly in RAB$L_ROP.

```
   $V
RAB$M_ULK
```

2. Set FDL CONNECT MANUAL_UNLOCKING and call FDL$PARSE within the program before opening the file.

3. Some higher-level languages have a keyword available in the OPEN statement to set this option.

BASIC - UNLOCK EXPLICIT

COBOL - I-O CONTROL
LOCK-HOLDING

FORTRAN - N/A

PASCAL - N/A

MACRO - RAB$L_ROP = RAB$V_ULK

ALTERNATIVE RECORD LOCKING CONTROLLING OPTIONS

There are several record locking options available that can be set in the RAB$L_RDP field (or its FDL equivalent). These provide the user with varying degrees of control over the default RMS record locking actions.

As described above for the readonly/sharing write case, none of these special record locking options will turn off completely record locking activities. There is no way to avoid RMS calls to the lock manager to request a lock and then release it. The locking control these options provide is that they can cut down the length of time a lock will be held, and the access mode requested of the lock manager. As described for the readonly process in the previous section, a query lock will be briefly taken out and then released for any of these special locking options. Even if the NLK (do not lock records) option is enabled, a query lock is requested in order to find out whether the record is locked against the process.

RAB$V_NLK
Do not lock record; specifies that the record accessed through a Get or Find service is not to be locked. The RAB$V_NLK option takes precedence over the RAB$V_ULK option.

This option corresponds to the FDL attribute CONNECT NOLOCK.

RAB$V_NXR
Nonexistent record processing; specifies that if the record randomly accessed through a Get or Find service does not exist (was never inserted into the file or was deleted), the service is to be performed anyway. This option applies to relative files only. For a Get service, the previous contents of a deleted record are returned. The processing of a deleted record returns a completion status code of RMS$_OK_DEL, and the processing of a record that never existed returns RMS$_OK_RNF.

This option corresponds to the FDL attribute CONNECT NONEXISTENT_RECORD.

RAB$V_REA
Lock record for read; specifies that the record is to be locked for a read operation for this process, while allowing other accessors to read (but not to modify) the record. Use this option only when you do not want the file to be modified by any subsequent activities. Use the RAB$V_RLK option to allow possible subsequent modification of the file.

This option corresponds to the FDL attribute CONNECT LOCK_ON_READ.

RAB$V_RLK
Lock record for write; specifies that a user who locks a record for modification is allowing the locked record to be read by other accessors. If both RAB$V_RLK and RAB$V_REA bits are specified, the RAB$V_REA bit is ignored. The RAB$V_NLK bit takes precedence

over all others.

This option corresponds to the FDL attribute CONNECT LOCK_ON_WRITE.

RAB$V_RRL
Read regardless of lock; read the record even if another stream has locked the record. This option allows the reader some control over access. If a record is locked against all access and if a Put or Get service is requested, then the record will be returned anyway (if the RAB$V_RRL option is specified), with the alternate status RMS$_OK_RRL.

This option corresponds to the FDL attribute CONNECT READ_REGARDLESS.

RAB$V_TMO
Timeout; specifies that if the RAB$V_WAT option was specified, the RAB$B_TMO field contains the maximum time value, in seconds, to be allowed for a record input wait caused by a locked record. If the timeout period expires and the record is still locked, RMS will abort the record operation with the RMS$_TMO completion status. Note that the maximum time allowed for a timeout is 255 seconds. Other functions of the RAB$V_TMO option are listed under "Miscellaneous Options."

This option corresponds to the FDL attribute CONNECT TIMEOUT_ENABLE.

This option is not supported for DECnet operations; it is ignored.

RAB$V_ULK
Manual unlocking; specifies that RMS will not automatically unlock records. Instead, once locked (through a Get, Find or Put service), a record must be specifically unlocked by a Free or Release service. The RAB$V_NLK option takes precedence over the RAB$V_ULK option.

This option corresponds to the FDL attribute CONNECT MANUAL_UNLOCKING.

RAB$V_WAT
Wait; if the record is currently locked by another stream, wait until it is available. This option may be used with the RAB$V_TMO option to limit waiting periods to a specified time interval.

This option corresponds to the FDL attribute CONNECT WAIT_FOR_RECORD.

MODULE 8
BUFFER MANAGEMENT: SEQUENTIAL, RELATIVE, AND INDEXED FILES

Major Topics

- Interaction of several RMS options with buffers:
  - Read-ahead and write-behind
  - Deferred write
  - Synchronous or asynchronous
- Local buffers — indexed file example
- Size and number of buffers
- Global buffers and index caching
  - Single node
  - VAXcluster
- Calculating number of buffers needed to cache index

Source

Guide to VAX/VMS File Applications — Chapter 7 (Section 7.3)
Chapter 3 (Section 3.6)

TYPES — (a) Local (in P1 region) — size → relative / ~~[illegible]~~ — BUCKUP

(b) Global

sequential — show RMS (Multiblock count = 16)

Calculation (a) how many

(b) size

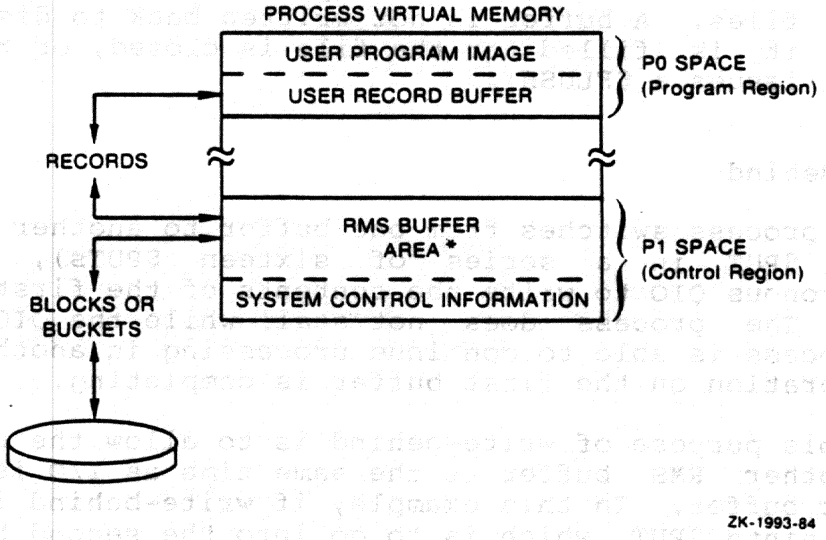
INTERACTION OF RMS OPTIONS WITH BUFFERS

The following RMS options affect buffer flushing.

- Read-ahead/write-behind
- Deferred write
- Asynchronous/synchronous

These options will be explained using an example that assumes eight records can fit into one buffer and a multibuffer count of two buffers.

RMS Buffers and the User Program



* RMS buffers for image-specific files begin in P1 but may overflow into P0 space, unless the user image was linked using the option IOSEGMENT= NOPOBUFS. This latter option is rarely used.

Read-Ahead/Write-Behind

- Applies only to nonshared sequential files
- Set with RAB$M_RAH and RAB$M_WBH bits in RAB$L_ROP
- If either or both of these bits are set, RMS will use two buffers by default
- By default provides asynchronous I/O

NOTE

Even if the write-behind option were not enabled, deferred write is always enabled for sequential files. A buffer is not written back to disk until it is filled (or the file is closed, or the user issues a $FLUSH).

Write-Behind

When a process switches from one buffer to another (for example, a ninth $PUT in a series of sixteen $PUTs), RMS issues an asynchronous QIO to write the contents of the first buffer out to disk. The process does not stall while the QIO is completing. The process is able to continue processing in another buffer while the operation on the first buffer is completing.

The whole purpose of write-behind is to allow the user to make use of another RMS buffer at the same time as I/O is in progress on another buffer. In this example, if write-behind is not enabled on the ninth $PUT, which is to go into the second buffer, RMS does not return control to the user until the first buffer has been written to disk (therefore, the process stalls).

Read-Ahead

The read-ahead option operates slightly different from write-behind. When the user does the first $GET, RMS issues the reads for both buffers at the same time. The process stalls until the first read is completed. The QIO associated with the second read completes asynchronously. In this example, on the seventeenth $GET, when the process turns back to the first buffer, RMS will again issue the reads for both buffers.

Asynchronous Option with Read-Ahead/Write-Behind Interaction

It is best not to use the asynchronous option at the same time as the read-ahead/write-behind (RAH/WBH) options. Except for one special case, when write-behind and read-ahead are enabled, setting the ASY bit has no effect, since asynchronous I/O takes

place anyway. The special case is where the write of an I/O buffer is not completed before the buffer needs to be reused.

In this case, if both ASY and WBH were enabled, after eight $PUTS the first buffer is full. The ninth $PUT goes into the second buffer and an asynchronous request is issued to write the first buffer out to disk. On the sixteenth $PUT, the second buffer is full; on the seventeenth $PUT, RMS issues an asynchronous QIO to write the second buffer out to disk.

Because of the speed of the I/O device or the system load, suppose the first asynchronous buffer write is not yet completed. The seventeenth record needs to go into the first buffer. The setting of the ASY bit affects what action RMS takes in this situation. If the ASY bit is set, RMS returns control to the user immediately. The burden is on the user to refrain from modifying the first buffer storage until the asynchronous I/O completes. If the ASY bit is not set, the user does not get control back from the seventeenth $PUT until it can be successfully moved into the first buffer.

Deferred Write

- Applies to relative and indexed files and, as of VMS 4.4, to shared sequential files
- Set by FAB$M_DFW bit in FAB$L_FOP
- If this bit is set, RMS will use two buffers by default
- The meaning of deferred write is slightly different for nonshared sequential files.

Without deferred write enabled, every $PUT, $UPDATE, or $DELETE to a relative, indexed, or shared sequential file results in at least one direct I/O operation. For example, $PUTs one through eight to a relative file buffer would have resulted in eight writes to the disk and one more direct I/O to bring the bucket into the buffer initially. With DFW enabled, the write to the disk is deferred until a modified buffer is needed. With deferred write turned on, it is possible to perform multiple $PUTs to one buffer and incur only one direct I/O.

One important difference between deferred write and write-behind is that control is not returned to the user while the write is being done to the disk. When the user issues the seventeenth $PUT, RMS will write-back the contents of the first buffer to the disk. Control will only be returned to the user after the write has completed.

While this may suggest that the asynchronous option should be enabled together with DFW, the same danger described above for write-behind is inherent. If both ASY and DFW were enabled, then when the user issues the seventeenth $PUT, RMS would return control immediately to the user. It is possible that the first buffer local storage area could be modified before the write-back to disk has completed. Again, as in the asynchronous write-behind sequential file example, the burden is on the user to refrain from modifying local buffer storage until the write-back to disk has been completed.

In the case of multiple $PUTs which are clustered, deferred write can result in substantial performance gains. There are other factors, however, that have to be taken into account:

- In the case of a system crash, data not written back to disk may be lost. Also, if index buckets are cached, the modified index buckets may not have been written back to disk.

- In the case of shared files, there will be some performance degradation due to blocking AST activity. This will be discussed in the section on global buffers.

NOTE

Not all operations on indexed files can be deferred. Any operation that causes a bucket split will force the write-back of the modified buckets to disk. This forced write-back decreases the chances of lost information should a system failure occur.

Asynchronous I/O

- Applies to all file organizations
- Set by RAB$M_ASY bit in the RAB$L_ROP

Setting this bit allows the user to get control back immediately from RMS, rather than RMS waiting for I/O completion before returning control to the user.

Setting the ASY bit gives the user the opportunity to perform some operations totally unrelated to a particular record stream, such as computations or I/O to other files, while I/O on a particular file is in progress.

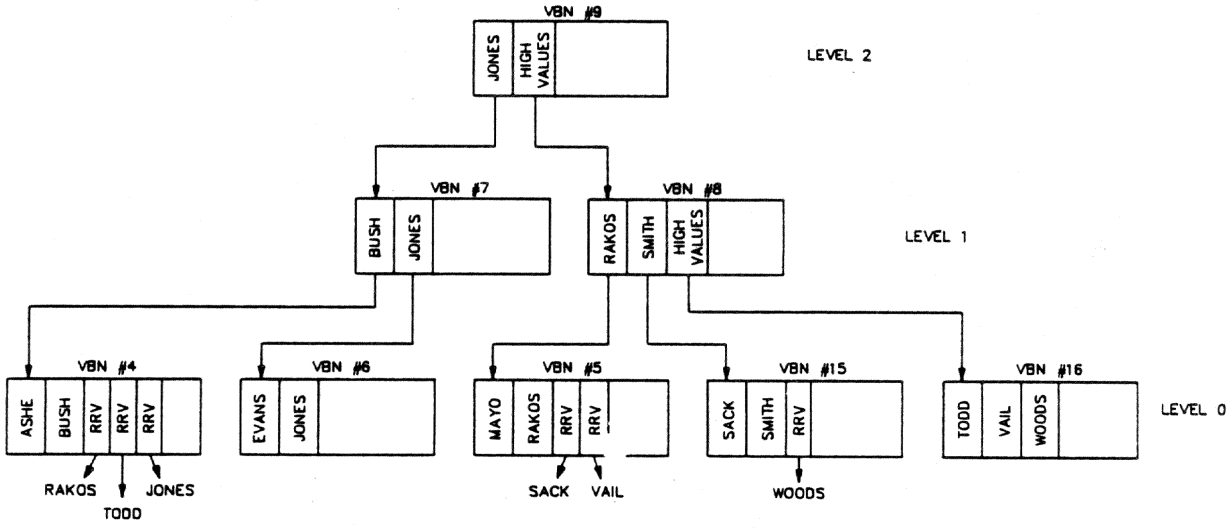
LOCAL BUFFERS

Management of local buffers by RMS will be illustrated using the simulated indexed file data example introduced in Part 2 of Module 5, Indexed File Organization. The illustration will involve an update to an existing indexed file in which new records will be inserted.

Example 1. Local Buffer Illustration

- Keyed $PUTs on primary key
- Multibuffer count = 3
- Exclusive use of file (or sharing read). Sharing write will be discussed in the section on global buffers.
- Synchronous, deferred write (assumes some of the insertions may be clustered)

Primary Index Tree

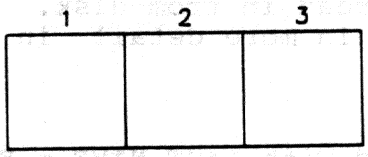


RMS sets up an internal table for local buffers (local list) with the number of cells equal to the number of local buffers.

Among the information maintained in this table are:

- What VBN # the bucket begins with
- Weighting factor (essentially this is the level in the tree structure associated with the bucket)

Local Buffer Internal Table



BU-2470

Update 1. $PUT Record With LAST_NAME = DOG

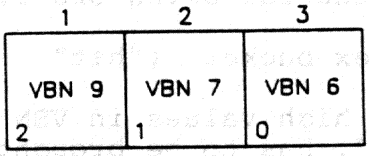
This $PUT involves following buckets to be brought into the local buffer from the disk in order listed:

VBN 9 - ROOT index bucket

VBN 7 - VBN that JONES in VBN 9 is pointing to

VBN 6 - VBN that JONES in VBN 7 is pointing to

Internal Table at End of $PUT DOG



BU-2471

General Steps for Each Bucket Accessed

1. Local list (internal table) is sequentially scanned to determine if the bucket needed is already in the buffer.

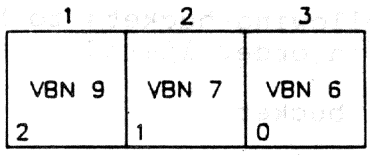
2. If there is no hit in Step 1, the local list is scanned to identify the buffer that will be used to bring in the bucket from the disk. The search routine used will be described later in this module.

3. If the file is write shared, then a request is made to the lock manager for exclusive access to the bucket. Once granted, the access mode is degraded by the lock manager and the bucket is read in from disk. (Write sharing procedures will be described in more detail in the section on global buffers.)

Update 2. $PUT Record With LAST_NAME = FOX

This $PUT involves the same VBNs as the $PUT for DOG. In each case the scan of the local list will lead to a hit.

Internal Table at End of $PUT FOX



BU-2472

Update 3. $PUT Record with LAST_NAME = PIG

This $PUT involves following buck ts to be brought into local buffer from disk in the following order.

VBN 9 - ROOT index bucket ("hit" already in buffer)

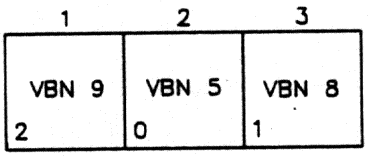
VBN 8 - VBN that high values in VBN 9 is pointing to (no "hit", has to be brought in)

VBN 5 - VBN that RAKOS in VBN 8 is pointing to (no "hit", has to be brought in)

The search routine used to identify which local buffer should be re-used will be illustrated by bringing in VBN 8 and VBN 5 from disk.

- The local list is scanned backwards to identify the first unused buffer with the lowest weight. In scanning through the list, a pointer is maintained to the cell with the lowest weight.
- As an optimization feature, if any unused buffer has an associated weight of zero, the sequential scan stops there, and the bucket will be brought in from disk to this buffer.
- If no unused buffer has an associated weight of zero, the total local list will be sequentially scanned. The bucket will be brought in from disk to the buffer associated with the lowest weight (the first one identified in the backwards scan in case of a tie).
- The scan always begins with the last cell in the local list.
- Unlike global buffers, no weight decrementing is done.

Internal Table at End of $PUT PIG



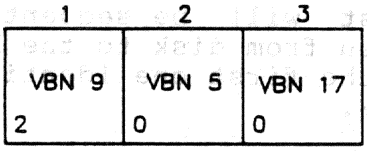
BU-2473

Update 4. $PUT record with LAST_NAME = MOUSE

This $PUT involves the same VBNs as the $PUT for PIG. In each case the scan of the local list will lead to a hit.

This update will involve a bucket split. In the case of a bucket split, the write-back to the disk of any modified buckets is not deferred. Modified buckets are immediately written back out to disk.

Internal Table at End of $PUT MOUSE



BU-2474

SIZE AND NUMBER OF BUFFERS

The size of the buffer is equal to the I/O unit of transfer from disk to memory.

	Unit of Transfer	
Sequential	Multi-blocks	
Relative	Bucket	maximum size = 63 blocks
Indexed	Bucket	

Sequential

Size
The number of blocks to be transferred to or from the disk can be varied each time the file is processed. If not specified by the process, the system default will be used. This is generally 16 blocks (SHOW RMS_DEFAULT).

Buffers
A maximum of two buffers is all that is needed for any sequential file. If either (or both) the read-ahead or write-behind option is enabled, RMS defaults to two buffers.

Relative

Size
Whatever the bucket size is set at when the file is created (or reorganized) will be the fixed size of the buffer. FDL uses the following rules in calculating bucket size for relative files.

Type of Processing	# Records per Bucket
Random	4
"Clustered" access or sequential	16

Buffers
The maximum number ever needed for a relative file is two buffers. If the deferred write option is enabled, RMS will default to two buffers.

Indexed

Size
Whatever the bucket size is set at when the file is created (or reorganized) will be the fixed size of the buffer. The size of each buffer used for a file will be a constant size for all buffers allocated to an indexed file. If areas were used to specify different bucket sizes for index and data buckets, the size of all buffers allocated to that file will be the largest bucket size.

Choosing Data Bucket Size for Indexed Files

- Larger data buckets yield fewer index buckets, which result in fewer DIOs, but longer search times (CPU) at the data level.
- Smaller data buckets yield more index buckets, which result in more DIOs, but shorter search times (CPU) at the data level.

For indexed files, it is the bucket size more than anything else that determines the shape and size of the index. A small bucket size relative to record size will result in an index with many levels, while a large bucket size relative to record size will result in a flatter index. Since each level will result in an additional disk access for each I/O, it is generally desirable to make the index as flat as practical. The flattest index will, of course, result when all the key values can fit in one index bucket (the root bucket). To accomplish this for larger file applications a very large bucket size (maximum of 63 blocks) may be required. However, there are five factors that can cause a large bucket size to adversely affect performance.

1. Data transfer time

   The I/O time required to do one direct I/O is made up of four components:

   I/O TIME = SEEK + LATENCY + DATA TRANSFER + BUCKET SEARCH

   seek time = average time required for the disk head to be positioned over the desired track (28 ms for an RA81).

   latency = average time required for the desired record to pass under the disk head after it has been positioned. This is about the time required for one half a rotation of the platter (8.3 ms for an RA81).

   data transfer = time required to transfer the number of bytes in the bucket, which equals the number of bytes to be transferred (512 times the bucket size) divided by the

transfer rate (2.2 Mbytes/sec for an RA81).

bucket search = average time required for the CPU to search through the bucket for the desired record once it is in memory.

Most of I/O time is consumed by mechanical motion (seek + latency). This is a property of the hardware and cannot be changed by the programmers. The four components of I/O are present for every I/O operation (except, in some cases, bucket search time). A major objective of tuning is to reduce the number of seeks required, since this is the largest single component of I/O.

However, the data transfer time can be controlled to some extent by the programmer since it is directly proportional to the bucket size. The calculations that follow (based on an RA81) demonstrate this relationship across a range of bucket sizes.

Relationship of Bucket Size to Data Transfer Time Based on RA81

Bucket Size	Seek Time	Rotational Latency	Data Transfer	Total	% Data Transfer of Total
1	28 ms	8.3 ms	.2 ms	36.5 ms	0.6%
10	28 ms	8.3 ms	2.3 ms*	38.6 ms**	6.0%***
15	28 ms	8.3 ms	3.5 ms	39.8 ms	8.8%
20	28 ms	8.3 ms	4.7 ms	41.0 ms	11.5%
32	28 ms	8.3 ms	7.4 ms	43.7 ms	16.9%
63	28 ms	8.3 ms	14.7 ms	51.0 ms	28.8%

* Data transfer = (10 x 512)/2200

** Total = 28.0 + 8.3 + 2.3

*** % Data transfer = 2.3/38.6 * 100

Source: VAX Hardware Handbook (Vol. 1, 1986)

The last column of this table reports the percentage of the total time which is represented by data transfer. For small buckets, this is insignificant. However, as the bucket size is increased, the time spent actually transferring the data becomes more and more significant. For sequential access, the

higher data transfer time may be balanced by a smaller number of direct I/Os. In this case, the high data transfer time per I/O is not important because most of the bytes that are transferred will be used by the CPU, whether in one large access or many smaller ones. For random access, only one record in the bucket will be accessed (unless there are multiple buffers set up for caching). Therefore, most of the data transfer work is pure overhead. As a general rule of thumb, it is wise to limit the data transfer time to between ten and fifteen percent of the total.

2. Data bucket search time

Data bucket search time is only significant in random access applications. Sequential access will cause every record in the bucket to be read, while with pseudo-random access, only the first record in a group of records will have to be located in the bucket. From that point on, it will look like sequential access. For large bucket sizes, decreased I/O resulting from a flatter index must be balanced against the time required to search through the buffer for the desired record. The bucket search time is a function of the CPU type and the number of records in the bucket. The larger the number of records, the greater the copy time, on average, to locate the record sequentially within the data bucket.

3. Memory constraints

Buffers are pages of memory in your working set. Each block (512 bytes) is a page of memory. A large bucket size will require more pages of memory. If more pages are required to support the size and number of buffers allocated than the working set supports, excessive paging can result.

4. Bucket locking

In a shared file application, an additional consideration becomes important. As described in the preceding module, buckets are locked intermittently during the period of time a record is locked. While the period of time during which the bucket is locked is small, large bucket size means that a portion of the file could potentially be locked out for some period of time. If there is a great deal of contention in a file sharing environment, these bucket locks could adversely affect performance.

5. Data loss in case of system crash

If the deferred write option is enabled, consideration must be given to how much data could potentially be lost. The number of records that can fit into the bucket is the number of records that potentially can be lost in case of system failure.

Summary -- Indexed Data Bucket Size

The above considerations yield tradeoffs which must be evaluated in the context of the application before a choice for bucket size can be made. If the application is I/O bound, the amount of I/O can be reduced at the expense of a greater load on the CPU. If the application is CPU bound, the load on the CPU can be reduced at the expense of I/O. If the application is both I/O and CPU intensive, the tradeoffs must be carefully studied and a compromise reached.

General Rule -- The data bucket size chosen should allow at least five records (near maximum size, if variable-length records) to fit in a bucket. The size should also be a multiple of the disk volume cluster size. The worst single thing that can be done to performance is to use one tightly fitting record per bucket.

In the final analysis, it may be necessary to experiment to determine the optimal bucket size. Often the best way to find the best buffering strategy for a particular application is to test various combinations of the number of buffers and the buffer size. One approach is to time each combination and measure the number of I/O operations that take place, and then choose the one that improved application performance the most considering the amount of memory used.

Number of Buffers

The number of buffers (CONNECT MULTIBUFFER_COUNT, RAB$B_MBF) is specified at run-time and recommended values can vary greatly for different applications when accessing indexed files. The following suggestions on the use of buffers apply to the type of record access to be performed.

- Completely random processing--When records are processed randomly, the use of as many buffers as your process working set can support is recommended to cache as many index buckets as possible.

- Sequential processing--When records will be accessed sequentially, even after locating the first record randomly, the use of a small multibuffer count, such as the default of two buffers, is sufficient.

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

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

NOTE

The general idea of using buffers is to use a buffer size and number of buffers that improves application performance without exhausting the virtual memory resources of your process or system. Keep in mind the tradeoffs between file I/O performance and exhausting memory resources. The buffers used by a process are charged against the process's working set. By default, buffers are locked in to the process's working set. You should avoid allocating so many buffers that the CPU spends excessive processing time paging and swapping. For performance-critical applications, consider increasing the size of the process working set and adding additional memory.

GLOBAL BUFFERS AND INDEX CACHING

Two types of buffer caches are available using RMS: local and global.

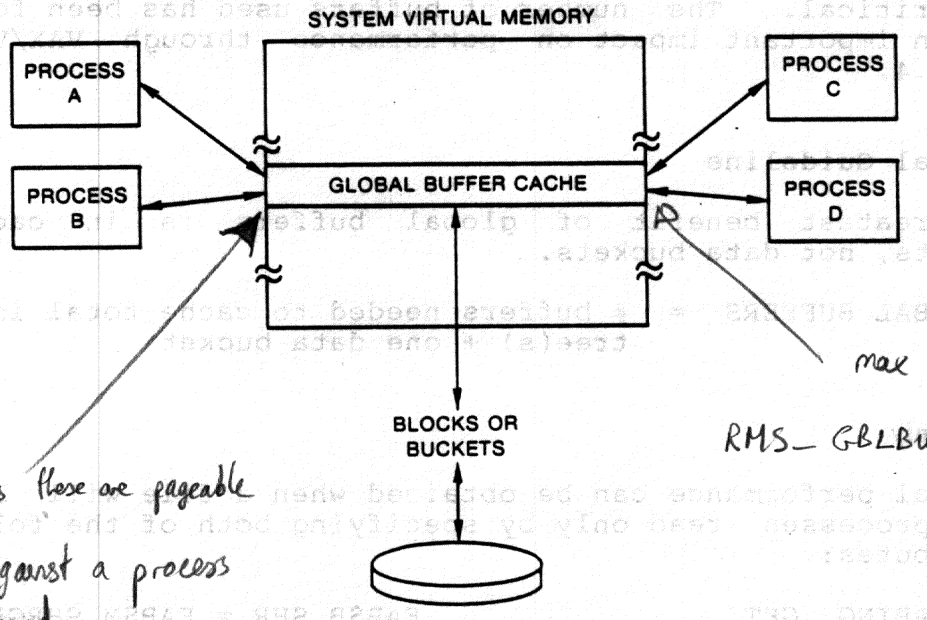
Local buffers reside within process (program) memory space and are not shared among processes, even if multiple processes are accessing the same file and reading the same records. Global buffers, which are designed for applications that access the same file and may even access the same records, do not reside in process memory space (but are charged to each process's working set).

If several processes will share an indexed file, global buffers should be considered. A global buffer is an I/O buffer that two or more processes can access in conjunction with file sharing.

This section is divided into two parts:

1. Global buffers on a single node
2. Global buffers on a VAXcluster

Using Global Buffers for a Shared File



* Unlike local buffers these are pageable
* These are charged against a process when they are mapped
* The first process may either:
  (a) Take the default number of global buffers
  (b) Specify a value

Second & subsequent processes may either:
  (a) Ignore global buffers
  (b) Accept what the 1st process has declared

SINGLE NODE

The greatest benefit of global buffers usually is found with an indexed file that is shared by multiple readers (file is opened by all processes read-only) and has a high locality of reference.

- Use of global buffers should be considered only if:
  - Several processes will be accessing the indexed file concurrently.
  - The processes will be accessing the file randomly, and there is a good probability of a high locality of reference (at a particular point in time, buckets in memory are being asked for by more than one process). The probability associated with index buckets is greater than with data buckets.

- Before implementing the use of global buffers as a general practice for a particular file, benchmarks should be done with and without global buffers.

  Global buffers not only do not always improve performance but may also degrade performance.

  In conducting benchmarks, the number of buffers specified is critical. The number of buffers used has been found to have an important impact on performance through VAX/VMS Version 4.4.

General Guideline

The greatest benefit of global buffers is in caching index buckets, not data buckets.

```
# GLOBAL BUFFERS  =  # buffers needed to cache total index
                     tree(s) + one data bucket
```

Read Only

Optimal performance can be obtained when a file will be open by all processes read only by specifying both of the following file attributes:

```
SHARING  GET                    FAB$B_SHR = FAB$M_SHRGET

                                            +

SHARING MULTISTREAMING                      FAB$M_MSE
```

This will improve performance by eliminating certain internal operations, such as the maintenance of bucket locks in the global buffers.

Not Restricted to Read Only

The search strategy used by RMS for global buffers will be illustrated using the simulated data example introduced in Module 5, Part 2 (see the Primary Index Tree and the Alternate Index Tree). The following illustration assumes that all processes sharing this file are on a single node, which may or may not be true in a VAXcluster.

Example 2. Global Buffer Single Node Illustration

Process A opens the file for shared access write with:

```
ACCESS                  WRITE
SHARING                 WRITE

# local buffers         =  2
# global buffers        =  6
```

Calculation of global buffers:

```
Primary index             3
Primary level 0           1
Alternate key 1 index     1
Alternate level 0         1
                         ----

      Total global        6
```

DEFERRED WRITE enabled

NOTE

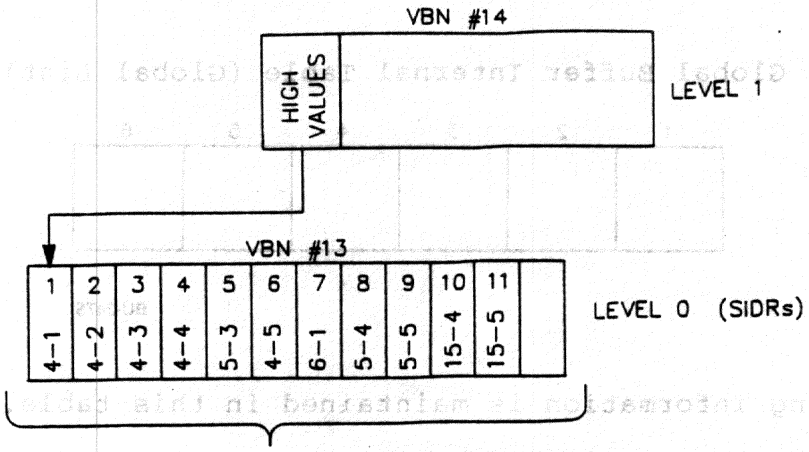
The deferred write option was enabled in this example for illustration purposes. In general, it is best not to use deferred write when global buffers are used. (Incurs extra lock management & buffer management)

Key of reference = 0 (primary key)

Process B opens file requesting read-only access for the user but allowing shared write access by others. Process B does not specify global buffers (or override global buffers by specifying in program control FAB$W_GBC = 0) so it defaults to global buffers set up by Process A. It also defaults to one local buffer.

```
Access                  Read-Only
Sharing                 Write
# global buffers       = 6  (by default)
# local buffers        = 1  (by default)

key of reference        = 1  (Alternate key 1)
```

Alternate Index Tree



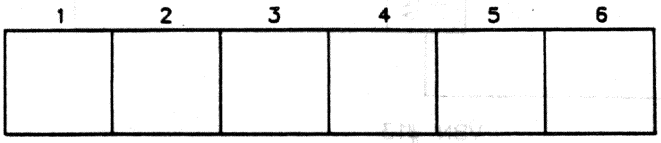
RFA's*	SEQ-NO	NAME
4-1	1	RAKOS
4-2	2	ASHE
4-3	3	TODD
4-4	4	JONES
5-3	5	VAIL
4-5	6	BUSH
6-1	7	EVANS
5-4	8	SACK
5-5	9	MAYO
15-4	10	WOODS
15-5	11	SMITH

*RFA = VBN # - ID #

BU-2469

RMS sets up an internal table for global buffers (global list) with number of cells equal to the number of global buffers.

Global Buffer Internal Table (Global List)



BU-2475

The following information is maintained in this table.

- What VBN # the bucket begins with
- Weighting factor (essentially this is initially the level in the tree structure associated with the bucket)
- Count of users touching (using) this specific buffer
- Sequence number (essentially the number of times a copy of this bucket has been written back to disk)
- What lock ID the system lock for this buffer resource is held with (used by distributed lock manager)

VAXCLUSTER relevance

Retrieval 1

Process A issues FIND ** JONES ** followed by a GET.

General Steps for Retrieval of VBN 9

1. Global list (internal table) is locked (Lock Manager) for scan to determine whether VBN 9 is in global buffer cache.

2. Sequential scan of list. In this case, no hit for VBN 9. If there were a hit, the "touched-by" count associated with this buffer is incremented so that if this process goes into wait state before gaining ownership of this VBN through the lock manager, another process in the interim cannot reuse this buffer to bring in another bucket.

3. Lock on global list is released.

NOTE

If process during list scan ever stalls, the lock on list is released so another process can get in.

4. List for local buffer is checked. In this case, no hit for VBN 9. If there had been a hit in Step 2, this step would be omitted.

5. If no hit in Step 2 or 4, global list is locked for scan to reserve buffer to which VBN 9 will be written. The search routine used will be described when all buffers in the global cache are filled in this illustration. In this case the global buffer associated with cell 1 will have it's "touched-by" count incremented.

6. If no hit in Step 2 or 4, the global list lock is released.

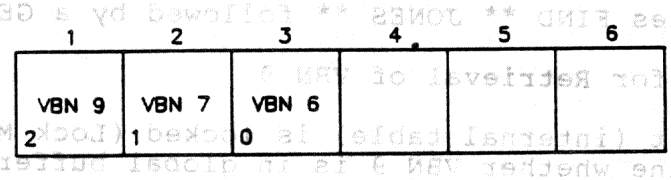
7. Request is made to the lock manager for exclusive access to VBN 9. If request is granted, the lock manager will degrade access mode, and VBN 9 will be written from disk to global buffer.

8. VBN 9 is searched to find the next VBN in the tree leading to the data bucket where JONES record is located.

   The above steps are repeated for VBN 7, and then again for VBN 6.

9. The data bucket (VBN 6) is brought into global buffer. This step will be expanded upon when an actual update is made in this example.

Internal Table at the End of Retrieval 1 (JONES)

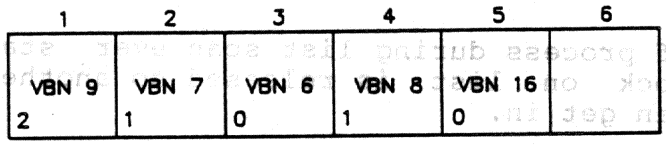


BU-2476

Retrieval 2

Process A issues FIND ** WOODS ** followed by a GET.

Internal Table at the End of Retrieval 2 (WOODS)

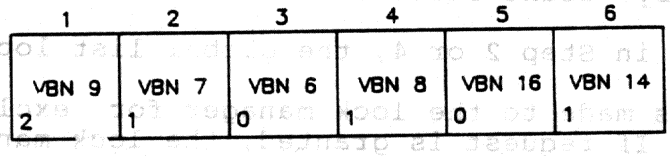


BU-2477

Retrieval 3

Process B issues FIND ** SEQ_NO = 9 (MAYO) ** followed by GET.

Internal Table After the Root Alternate Key Bucket Brought In



BU-2478

After VBN 14 has been brought in, VBN 13 is identified as the SIDR bucket containing the RFA for SEQ_NO = 9. The scan of the global buffers for VBN 13 and the local buffers had no hit.

Step 5 Expanded

Step 5 in Retrieval 1 will now be expanded upon to describe the search routine used to identify which buffer should be re-used.

1. The list is locked for search.

2. A total of eight unused cells in any such search is scanned (this example is atypical because it is limited to only six cells). A pointer is maintained to the last cell scanned in searching to identify which buffer will be recycled by any process. This will be the first cell in the next search performed for any process.

3. In scanning through the eight cells, a pointer is maintained to the cell with the lowest weight.

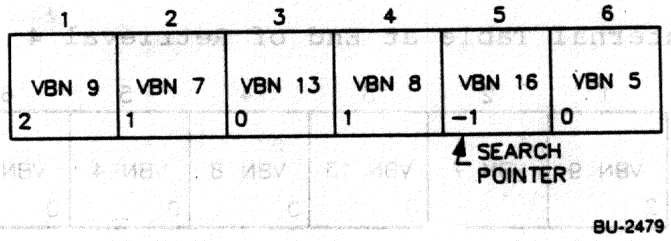
4. The last cell in the scan has its weight decremented by one. If someone uses the bucket in this cell before it is written over, the original weight (its level in the index tree) is restored.

5. The cell identified has its "touched by" count incremented by one in order to reserve it.

6. The lock on the list is released.

7. The bucket is brought in to the global buffer associated with the identified cell.

Internal Table at the End of Retrieval 3 (MAYO)



Retrieval 4

Process A issues FIND ** BUSH ** followed by UPDATE.

Step 9 Expanded

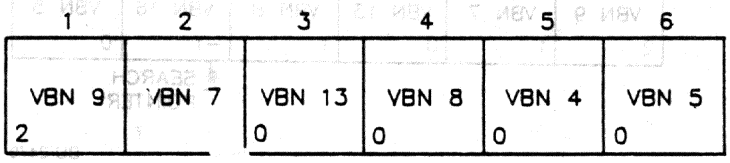
Step 9 in Retrieval 1 will now be expanded upon using the update done to VBN 4 in this example.

In case of no bucket split, the update made by Process A is made to the data record in bucket VBN 4 in the global buffer. When the lock manager is requested to release Process A's ownership of VBN 4, the lock manager will see that the deferred write option was specified. The lock manager must abide by the rule that a bucket that has not been written back to disk cannot be unowned. The lock manager will have ISAM copy the bucket in the global buffer to one of Process A's local buffers and then the lock manager can release ownership of the global buffer. The lock manager still has a lock on VBN 4 but on the local copy. Note the extra overhead involved due to the deferred write option being enabled.

At the end of this step, there is a valid copy of VBN 4 in a local cache and an equally valid copy in the global cache but an invalid copy out on disk. Process A owns the local copy and no one owns the global copy. Retrieval 5 describes what happens when someone wants to access the global copy.

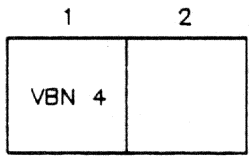
In case of bucket split, an update is made to the bucket in the global cache and the local buffer is used for the new bucket created. The deferred write option becomes inoperative. The buckets involved are immediately written back out to disk.

Internal Table at End of Retrieval 4 (BUSH)



BU-2480

Process A's Local Buffers at the End of Retrieval 4 (No Bucket Split)



BU-2481

Retrieval 5

Process B issues FIND ** SEQ_NO = 2 (ASHE) ** followed by GET.

Step 2 Expanded

In this case, the sequential scan of the internal list will disclose VBN 4 is in the global cache. The scan of the list only discloses that it is in the global cache. At this point, the "touched by" count is incremented in the list so that while Process B is trying to get ownership, if it goes into a wait state, no other process will be able to use this buffer for recycling purposes.

Step 7 Expanded

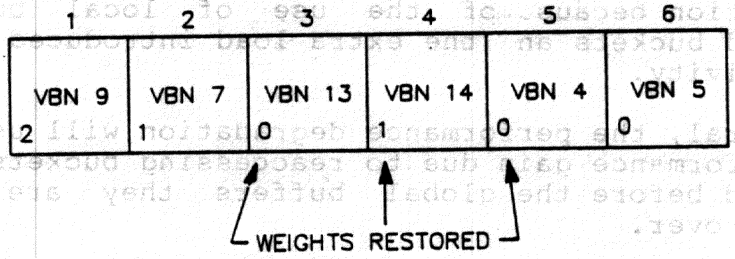
When a request for a lock on VBN 4 for Process B is made to the lock manager, the lock manager will see that a copy of VBN 4 is owned locally. The lock manager will initiate an AST for the local copy to be written out to disk. Once it is copied to disk, the lock manager will release Process A's ownership of it and give ownership of the copy in the global cache to Process B.

NOTE

If the deferred option had not been enabled, a local copy would not have been made and performance would not be degraded by the extra load introduced involving the AST activity. It is for these reasons that performance is usually better if deferred write is turned off when global buffers are used.

Since a valid copy of VBN 4 was already found in the global cache, it will not have to be recopied from disk.

Internal Table at the End of Retrieval 5 (ASHE)



BU-2482

Summary – Global Buffer Performance (as of VAX/VMS Version 4.4) – problems in releases prior to this

1. Each time the global list table is scanned, the entire global list (not buckets) is locked.

2. The scan of the global list is done sequentially. If the number of global buffers gets too large, the search time to scan the list may exceed the time it would take to do direct I/Os.

   Some users have run into very poor performance with global buffers when they try to cache a large number of data buckets and the number of buffers in the global cache becomes very large.

3. General recommendation:

   For a shared file, if you can use global buffers to cache the entire index structure (not data buckets), then everybody wins.

   If you cache the entire index structure locally, then the process may win at the expense of other processes (using more memory). This would be an appropriate strategy only for a nonshared file.

   NOTE

   The argument for caching all or a lot of the index structure falls apart for sequential access, where a small number of buffers (such as two) is plenty.

4. It is usually best that the deferred write option be turned off when global buffers are used. In a high-contention environment where frequent concurrent updating is occurring, deferred write enabled can actually cause performance degradation because of the use of local buffers for the modified buckets and the extra load introduced by the blocking AST activity.

   In general, the performance degradation will usually outweigh any performance gain due to reaccessing buckets that have been modified before the global buffers they are in have been written over.

- Global buffers reside in physical memory so each VAX node in a cluster has its own global buffer cache. There is no performance difference between a single-node system and a VAXcluster if the file sharing takes place on a single node of the cluster.

- There is no performance difference between a single-mode system and a VAXcluster if the file sharing allows read-only.

- Once a file is opened on more than a single node in a VAXcluster with sharing allowing write, the distributed lock manager is invoked.

Example 3. Global Buffers VAXcluster Illustration

- Process A opens PERSONNEL.DAT on NODE ALPHA with a global buffer count (GBC) of 50.

- Process B opens PERSONNEL.DAT on NODE BETA with a GBC of 35.

- There will be two separate global caches in physical memory for PERSONNEL.DAT -- a 50-bucket one on NODE ALPHA and a 35-bucket one on NODE BETA.

One concrete illustration of the overhead added on to the VAXcluster shared file write case:

- Process A updates VBN 4. It doesn't matter whether Process A specified the deferred write option or not.

- The global cache in NODE BETA happens to already have a copy of VBN 4 in its cache from some previous read operation. Process B then asks for VBN 4.

- The scan of the global list on NODE BETA will disclose that VBN 4 is in the global cache.

  If it were to give Process B access to that copy, Process B would be using a stale copy of VBN 4.

  A lot of distributed lock manager overhead is added on to detect when a copy in one of the global caches on the VAXcluster is no longer valid and a new copy must be brought in from the disk.

  This is essentially accomplished through a lock value block maintained by the distributed lock manager for each bucket in a global buffer.

Lock Value Block

- For all bucket locks, the lock value block contains a sequence number, which is the number of times the bucket has been written out to disk. Every time a bucket is written out to disk, the sequence number is bumped and the lock value block written back to the lock manager.

- When the first bucket gets locked, one of the pieces of information gathered about the bucket is its sequence number. Every bucket on the system, in local or global cache, has a sequence number associated with it. The sequence number is also among the pieces of information kept by RMS in the internal tables maintained for global buffers and local buffers.

  When a request is made to the lock manager for a VBN which is already in memory in a global or local buffer, the lock manager checks to see whether the sequence number in the lock value block for that VBN resource matches the sequence number in the internal table. If they don't match, the bucket is read in again from the disk.

  When a bucket in the global buffer is updated and written back to disk, the sequence number maintained for it in the internal global list table is also updated.

- Unfortunately, if no one owns the bucket in the global buffer (no one has a lock on the bucket), the resource managed by the lock manager would go away. Normally no one owns global buffers.

  In order to know whether the bucket in a global buffer is being used by anyone currently, or is still valid, the resource must remain so that the lock value block associated with it is still available.

  In order to accomplish this, there is a system lock maintained on each global buffer. The entire reason for this system lock is to make sure the resource associated with the VBN stays around with its accompanying lock value block. Behind every global buffer there is an invisible system lock.

Performance Recommendations for VAXcluster Global Buffers

The following alternatives to write sharing a large data file on a VAXcluster should be considered.

- File sharing on more than one node of a VAXcluster should be restricted to READ ONLY SHARING if at all possible.

- Processing a file with exclusive access gives better performance than with shared write access on more than one node of a VAXcluster.

- If your application requires write sharing, if possible confine the activity to a single CPU. If sufficient CPU resources and I/O capacity are available, your application will perform faster than if it were spread over many nodes.

Example 4. Real-Data ANALYZE/RMS/STAT Output
(Sheet 1 of 2)

```
RMS File Statistics
Page  1

FILE HEADER

    File Spec: DISKXYZ:[USERA]REALDATA1.DAT;2
    File ID: (31365,15,0)
    Owner UIC: [120,007]
    Protection:  System: R, Owner: RWED, Group: R, World:
    Creation Date:     1-JAN-1986 20:44:32.52
    Revision Date:     31-JAN-1986 20:45:01.50, Number: 170
    Expiration Date: none specified
    Backup Date:      none posted
    Contiguity Options:  contiguous-best-try
    Performance Options: none
    Reliability Options: none
    Journaling Enabled:  none


RMS FILE ATTRIBUTES

    File Organization: indexed
    Record Format: fixed
    Record Attributes:    carriage-return
    Maximum Record Size: 64
    Longest Record: 64
    Blocks Allocated: 2148, Default Extend Size: 522
    Bucket Size: 3
    Global Buffer Count: 0


FIXED PROLOG

    Number of Areas: 2, VBN of First Descriptor: 2
    Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 2, offset %X'0000')

    Bucket Size: 3
    Reclaimed Bucket VBN: 0
    Current Extent Start: 1, Blocks: 2085, Used: 1025, Next: 1026
    Default Extend Quantity: 522
    Total Allocation: 2085

STATISTICS FOR AREA #0

    Count of Reclaimed Blocks:                0

AREA DESCRIPTOR #1 (VBN 2, offset %X'0040')

    Bucket Size: 3
    Reclaimed Bucket VBN: 0
    Current Extent Start: 2086, Blocks: 63, Used: 48, Next: 2134
    Default Extend Quantity: 15
    Total Allocation: 63
```

Example 4 (Sheet 2 of 2)

```
RMS File Statistics
Page  2

STATISTICS FOR AREA #1

   Count of Reclaimed Blocks:          0


KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

   Index Area: 1, Level 1 Index Area: 1, Data Area: 0
   Root Level: 2
   Index Bucket Size: 3, Data Bucket Size: 3
   Root VBN: 2089
   Key Flags:
          (0)  KEY$V_DUPKEYS     0
          (3)  KEY$V_IDX_COMPR   1
          (4)  KEY$V_INITIDX     0
          (6)  KEY$V_KEY_COMPR   1
          (7)  KEY$V_REC_COMPR   1
   Key Segments: 1
   Key Size: 58
   Minimum Record Size: 58
   Index Fill Quantity: 1536, Data Fill Quantity: 1536
   Segment Positions:        0
   Segment Sizes:           58
   Data Type: string
   Name: "SPECIAL_KEY"
   First Data Bucket VBN: 3

STATISTICS FOR KEY #0

   Number of Index Levels:                2
   Count of Level 1 Records:            341
   Mean Length of Index Entry:           60
   Count of Index Blocks:                48
   Mean Index Bucket Fill:              74%
   Mean Index Entry Compression:        15%

   Count of Data Records:              8944
   Mean Length of Data Record:           64
   Count of Data Blocks:               1023
   Mean Data Bucket Fill:               92%
   Mean Data Key Compression:           37%
   Mean Data Record Compression:         0%

   Overall Space Efficiency:            52%

   The analysis uncovered NO errors.


   ANALYZE/RMS_FILE/STATISTICS/OUT=REALDATA1.ANL  REALDATA1.DAT
```

3

48

$1 + \left(\frac{48}{3}\right) = 17$ buffers to cache entire index for primary key

allows 1 data bucket

Example 5. Run-Time Statistics for REALDATA1.DAT
Varying the Number of Buffers

# BUFFERS	ELAPSED	CPU	BIO	DIO	PG FAULTS
1	7:13.23	4:20.67	3	2742	2
3	4:49.64	4.11.32	3	1637	20
6	4:40.88	4.08.60	3	1397	26
9	4:31.11	4.03.99	3	582	25
20	4:18.30	4:02.27	3	138	30
30	4:06.69	4:00.21	3	137	49
40	4:05.36	4:00.76	3	137	95

Break-even somewhere round here. Corresponds to previously calculated figure of 17

Reduction in Direct I/O between buffer = 1 and buffer = 20

$$\frac{(2742 - 138)}{2742} = 95.0\% \text{ DIO reduction}$$

Reduction in Elapsed Time between buffer = 1 and buffer = 20

$$\frac{(433.23 - 258.30)}{433.23} = 40.4\% \text{ reduction in Elapsed Time}$$

MODULE 9
RMS UTILITIES

Major Topics

Part 5. Optimizing and reorganizing files

- FDL Optimizing function
- Reorganizing files
  - CONVERT
  - CONVERT/RECLAIM

Source

Guide to VAX/VMS File Applications — Chapter 10

MODULE 9
RMS UTILITIES

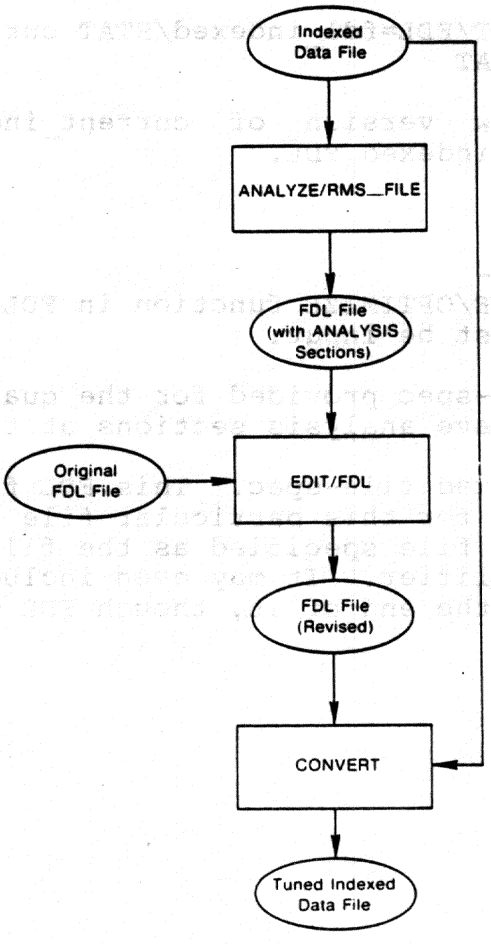
PART 5. OPTIMIZING AND REDESIGNING FILES

To maintain files properly, you must occasionally tune them. Tuning involves adjusting and readjusting the characteristics of the file, generally to make the file run faster or more efficiently, and then reorganizing the file to reflect those changes.

There are basically two ways to tune files. You can redesign your FDL file to change file characteristics or parameters. You can change these characteristics either interactively with EDIT/FDL (the preferred method) or by using a text editor. With the redesigned FDL file you can create a new data file.

You can also optimize your data file by using ANALYZE/RMS_FILE with the /FDL qualifier. This method, rather than actually redesigning your FDL file, produces an FDL file containing certain statistics about the file's use that you can then use to tune your existing data file.

The RMS Tuning Cycle



FDL Optimizing Function

To periodically optimize an indexed file, use the following steps:

1. $ANALYZE/RMS/FDL current_indexed.DAT

   Creates current_indexed.FDL which includes both a regular FDL specification and an analysis section.

2. $EDIT/FDL/ANALYSIS=current_indexed.FDL fdl_indexed.FDL

   Invokes FDL interactively. Choose the OPTIMIZE option* within the INVOKE menu. The default values provided for the interactive session will be taken from fdl_indexed file. Changes in bucket size, allocation size, etc. will be suggested by FDL on the basis of the analysis sections provided in the current_indexed file. A revised FDL file will be output with the name of the fdl_indexed file-spec.

   or

   $EDIT/FDL/ANALYSIS=current_indexed.FDL/NOINTERACTIVE -
   $_fdl_indexed.FDL

3. $CONVERT/NOSORT/FDL=fdl_indexed/STAT current_indexed.DAT -
   $_newindexed.DAT

   Creates a new version of current_indexed.DAT using the optimized fdl_indexed.FDL.

---

* To use the INVOKE/OPTIMIZE function in FDL, two FDL files must be input:

1. The FDL file-spec provided for the qualifier ANALYSIS which must have analysis sections at the end of it.

2. The fdl_indexed file-spec. This FDL file may be an old FDL file for this particular file or it may be the same FDL file specified as the file-spec for the ANALYSIS qualifier. It may even include analysis sections at the end of it, though FDL will not use them.

Reorganizing Files

- CONVERT
  - Bucket splits/RRVs disappear
  - Buckets of deleted records are reclaimed
  - RFAs are not preserved

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

- CONVERT/RECLAIM (Prolog 3 files only)

CONVERT/RECLAIM makes available for reuse those buckets that have been completely emptied by record deletions. The reclaiming is done in place, so no additional space is necessary. if there are severe space and time constraints, this is a useful feature, since the file size is kept to manageable levels. However, this results in negligible performance improvement, since bucket splits are not cleaned up. If performance is a critical issue, then a full CONVERT should be performed as often as possible.

  - Records will retain their original RFAs
  - Reclaims data buckets completely emptied by deletions

```
$ CONVERT/RECLAIM file-spec
```

- Other CONVERT options:
  - /EXCEPTIONS_FILE=file-spec
  - /PAD=value
  - /TRUNCATE
  - /STATISTICS
  - /MERGE=file-spec

- CONVERT
  - Bucket split/RMS disappear
  - Buckets of deleted records are reclaimed
  - PFAs are not preserved
  - $ CONVERT/BOI=fdl-file-spec input-file-spec output-file-spec

- CONVERT/RECLAIM (period 3 files only)

  CONVERT/RECLAIM makes available for reuse those buckets that have been completely emptied by record deletions. The reclaiming is done in place, so no additional space is necessary. If there are severe space and time constraints, this is a useful feature, since the file size is kept to manageable levels. However, this results in negligible performance improvement, since bucket splits are not cleaned up. If performance is a critical issue, then a full CONVERT should be performed as often as possible.

  - Records will retain their original PFAs
  - Reclaims data buckets completely emptied by deletions
  - $ CONVERT/RECLAIM file-spec

- Other CONVERT options:
  - /EXCEPTIONS_FILE=file-spec
  - /PAD=value
  - /TRUNCATE
  - /STATISTICS
  - /WORKB=file-spec

MODULE 10
OPTIMIZING FILE PERFORMANCE: DESIGN AND TUNING SUMMARY

Major Topics

- Design: file creation parameters
- Tuning: run-time parameters

Source

Guide to VAX/VMS File Applications — Chapter 4 (Sections 4.1-4.5)

MODULE 10
OPTIMIZING FILE PERFORMANCE: DESIGN AND TUNING SUMMARY

Major Topics

- Design: file creation parameters
- Tuning: run-time parameters

Source

Guide to VAX/VMS File Applications — Chapter 4 (Sections 4.1–4.5)

DESIGN -- FILE CREATION PARAMETERS

SEQ	REL	INDX	
X	X	X	1. Contiguous disk allocation
			-- Initial file allocation
			-- Extend size
X			2. Block spanning (sequential files)
	X	X	3. Bucketsize
		X	4. Primary Key
			-- Unique value
			-- Position 0
		X	5. Number of alternate keys
		X	6. Multiple areas
		X	7. Bucket fill factor

8. Compression - index (more important)
- data

TUNING -- RUN-TIME PARAMETERS

SEQ	REL	INDX		
X			1.	Buffer size (sequential only)
X	X	X	2.	Number of buffers
X			3.	READ-AHEAD or WRITE-BEHIND (sequential only)
	X	X	4.	DEFERRED WRITE { 1. Per volume, 2. Per image }
X	X	X	5.	Window size
	X	X	6.	Global buffers
			7.	Record locking options
			8.	Fast delete

should not be important - frequent file & disk restructuring to avoid fragmentation is required

SDA> show process /rms

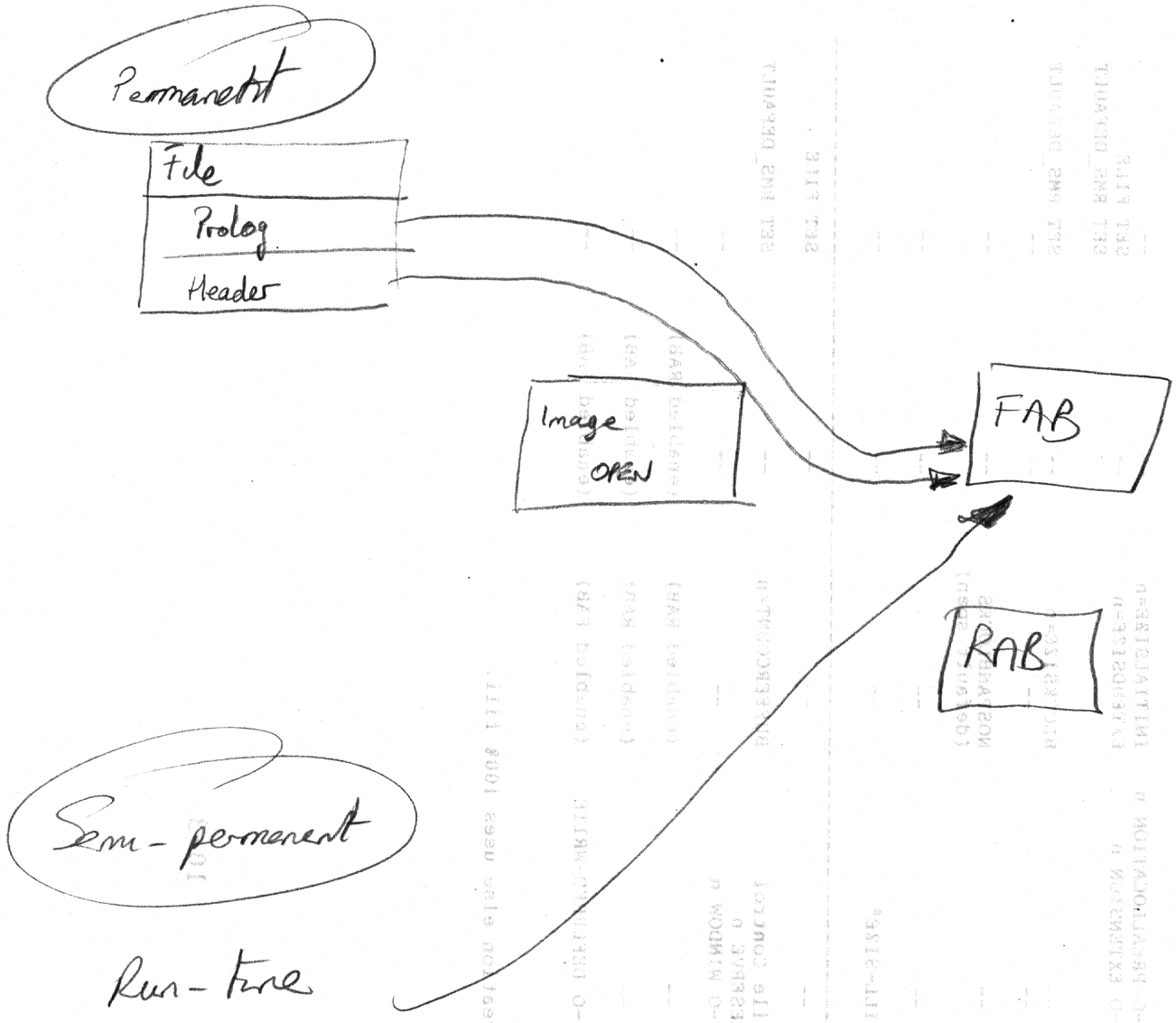
static display - show local & global buffers

Existing Keywords Available in Higher-Level Languages for Implementing Optimizing Features

Optimizing Features	BASIC	COBOL	FORTRAN	PASCAL	DCL
Alloc.		I-O Control			
Best-try-contiguous	Contiguous	Contiguous-best-try Contiguous	--	--	--
Contiguous					
Initial alloc.	FILESIZE n	I-O PREALLOCATION n	INITIALSIZE=n	--	--
Extend	EXTENDSIZE n	I-O EXTENSION n	EXTENDSIZE=n	--	SET FILE SET RMS_DEFAULT
Buffer size					
Multiblocks (sequential)	--	--	BLOCKSIZE=n	--	SET RMS_DEFAULT
Bucket size (rel., indx)	BUCKETSIZE n	--	--	--	--
Blockspan (sequential)	NOSPAN, SPAN	--	NOSPANBLOCKS (default span)	--	--
Multiple areas	--	--	--	--	--
Fill factor	--	FILL-SIZE*	--	--	--
Global buffers	--	--	--	--	SET FILE
Number buffers	BUFFER n	File Control RESERVE n	BUFFERCOUNT=n	--	SET RMS_DEFAULT
Window size	WINDOWSIZE n	I-O WINDOW n	--	--	--
Read-ahead (sequential)	--	--	(enabled RAB)	(enabled RAB)	--
Write-behind (sequential)	--	--	(enabled RAB)	(enabled RAB)	--
Deferred write (relative, index)	(enabled FAB)**	I-O DEFERRED-WRITE	(enabled FAB)	(enabled FAB)	--

* Causes it to use fill size specified at file creation else uses 100% fill.

** Enabled if ALLOW not equal to WRITE or MODIFY.


Permanent
File
Prolog
Header
Image
OPEN
FAB
RAB
Semi-permanent
Run-time

MODULE 11
PERMANENT FILE ATTRIBUTES VERSUS RUN-TIME FILE CHARACTERISTICS

Major Topics

- Permanent file attributes
- Run-time file characteristics
- Default settings for RMS control blocks for higher-level language OPENs
- DCL commands for implementing run-time features
  - SET FILE
  - SET RMS_DEFAULT
- Specifying run-time options

Source

Guide to VAX/VMS File Applications — Chapter 9

RMS Reference — Chapter 5 (FAB), Chapter 7 (RAB)

VAX/VMS File Definition Language Utility Reference

MODULE 11
PERMANENT FILE ATTRIBUTES VERSUS RUN-TIME FILE CHARACTERISTICS

Major Topics

- Permanent file attributes
- Run-time file characteristics
- Default settings for RMS control blocks for higher-level language OPENs
- DCL commands for implementing run-time features
  - SET FILE
  - SET RMS_DEFAULT
- Specifying run-time options

Source

Guide to VAX/VMS File Applications — Chapter 9
RMS Reference — Chapter 3 (FAB), Chapter 7 (RAB)
VAX/VMS File Definition Language Utility Reference

PERMANENT FILE ATTRIBUTES

Certain attributes are assigned to a file at file creation time and cannot be changed without creating a new file and transferring the records to it. Use the DCL commands ANALYZE/RMS, EDIT/FDL, CREATE/FDL, and CONVERT to do this.

- File organization
  - Sequential
  - Relative
  - Indexed sequential

- Record type
  - Fixed-length
  - Variable-length
  - Variable with fixed control (VFC)
  - Stream

- Maximum record length

- Area definitions
  - Bucket size
  - Fill size
  - Contiguity options
  - Initial allocation of disk space

- Key attributes
  - Key number
  - Key name
  - Data type for key field
  - Position of key field (can be up to eight segments)
  - Lengths of key segments
  - Duplicates allowed?
  - Changes allowed to (alternate) key fields on updates?
  - Null key character (alternate string keys only)

- Prolog version number

- Compression options for Prolog V3 files:
  - Data record compression (except primary key)
  - Primary key compression in data records
  - Key compressions in index records

- Block-spanning for sequential files

- Carriage-control record attributes

The following attributes are assigned at file creation time and can be changed at any time without making a new copy of the file. The following attributes may be changed with the DCL command SET FILE (or within program control by modifying RMS control blocks).

- Access control list
- Backup action
- Reliability options (read-check and/or write-check)
- Delete action
- Expiration date
- File extend quantity
- Global buffer count
- Owner UIC
- Protection code
- Version limit

RUN-TIME FILE CHARACTERISTICS

Some of the options concerning file use can be dynamically specified at run time. This is usually done when opening the file or by changing the record stream context within the program.

File Open Options

- Global buffer count — # can be set as file attribute
- Number of local buffers — # not 'file-oper' option but CONNECT option
- Retrieval window size — # also DCL
- File sharing options — # volume attribute
- File processing options which can be set in FAB$L_FOP (for example, deferred write)
- Default extension quantities

Record Connect Options

- Access mode (sequential, keyed, or RFA)
- Record locking and processing options that can be set in RAB$L_ROP (for example, fast delete)

Run-time file open and record connection options that apply to file performance are summarized below.

Option	Description
Asynchronous record processing*	Specifies that record I/O for this record stream will be done asynchronously. FDL: CONNECT ASYNCHRONOUS RMS: RAB$L_ROP RAB$V_ASY
Deferred write*	Allows records to be accumulated in a buffer and written only when the buffer is needed or when the file is closed. FDL: FILE DEFERRED_WRITE RMS: FAB$L_FOP FAB$V_DFW
Default extension quantity	Specifies the number of blocks to be allocated to a file when more space is needed. FDL: FILE EXTENSION RMS: RAB$W_DEQ
Fast delete*	Postpones certain internal operations associated with deleting indexed file records until the record is accessed again. This allows records to be deleted rapidly, but may affect the performance of subsequent accessors reading the file until its next-scheduled convert. FDL: CONNECT FAST_DELETE RMS: RAB$L_ROP RAB$V_FDL
Global buffer count	Specifies whether global buffers will be used and the number to be used if the record stream is the first to connect to the file. FDL: CONNECT GLOBAL_BUFFER_COUNT RMS: FAB$W_GBC
Locate mode*	Allows the use of locate mode, not move mode, when reading records. FDL: CONNECT LOCATE_MODE RMS: RAB$L_ROP RAB$V_LOC

Multiblock count	Allows multiple blocks to be transferred into memory during a single I/O operation; for sequential files only. FDL: CONNECT MULTIBLOCK_COUNT RMS: RAB$B_MBC
Number of buffers	Enables the use of multiple buffers. FDL: CONNECT MULTIBUFFER_COUNT RMS: RAB$B_MBF
Read-ahead*	Alternates buffer use between two buffers; as of VMS 4.4 for nonshared sequential files only. FDL: CONNECT READ_AHEAD RMS: RAB$L_ROP RAB$V_RAH
Retrieval window size	Specifies the number of entries in memory for retrieval windows, which corresponds to the number of extents for a file. FDL: FILE WINDOW_SIZE RMS: RAB$B_RTV
Sequential access only	Indicates that the file will be accessed sequentially only; for sequential files only. FDL: FILE SEQUENTIAL_ONLY RMS: FAB$L_FOP FAB$V_SQO
Write-behind*	Alternates buffer use between two buffers; as of VMS 4.4 for nonshared sequential files only. FDL: CONNECT WRITE_BEHIND RMS: RAB$L_ROP RAB$V_WBH

* Indicates on option that can be specified for each record-processing operation.

Existing Keywords Available in Higher-Level Languages Available for Implementing Optimizing Features

Optimizing Features	BASIC	COBOL	FORTRAN	PASCAL	DCL
Alloc.		I-O Control			
Best-try-contiguous	Contiguous	Contiguous-best-try Contiguous	--	--	--
Contiguous					
Initial alloc.	FILESIZE n	I-O PREALLOCATION n	INITIALSIZE=n	--	--
Extend	EXTENDSIZE n	I-O EXTENSION n	EXTENDSIZE=n	--	SET FILE SET RMS_DEFAULT
Buffer size					
Multiblocks (sequential)	--	--	BLOCKSIZE=n	--	SET RMS_DEFAULT
Bucket size (rel., indx)	BUCKETSIZE n	--	--	--	--
Blockspan (sequential)	NOSPAN, SPAN	--	NOSPANBLOCKS (default span)	--	--
Multiple areas	--	--	--	--	--
Fill factor	--	FILL-SIZE*	--	--	--
Global buffers	--	--	--	--	SET FILE
Number buffers	BUFFER n	File Control RESERVE n	BUFFERCOUNT=n	--	SET RMS_DEFAULT
Window size	WINDOWSIZE n	I-O WINDOW n	--	--	--
Read-ahead (sequential)	--	--	(enabled RAB)	(enabled RAB)	--
Write-behind (sequential)	--	--	(enabled RAB)	(enabled RAB)	--
Deferred write (relative, index)	(enabled FAB)**	I-O DEFERRED-WRITE	(enabled FAB)	(enabled FAB)	--

* Causes it to use fill size specified at file creation else uses 100% fill.

** Enabled if ALLOW not equal to WRITE or MODIFY.

DEFAULT SETTINGS FOR RMS CONTROL BLOCKS FOR HIGHER-LEVEL LANGUAGES

FAB Default Settings

Settings of VAX RMS FAB Fields by Higher-Level Language Compilers by Default at FILE OPEN

Field	Name	BASIC OPEN Keyword and Value	FORTRAN OPEN Keyword and Value	PASCAL OPEN Parameters and Value
FAB$L_ALQ	Allocation quantity	As specified by FILESIZE	n if INITIALSIZE = n	Initialized to zero
FAB$B_BKS	Bucket size (# blocks)	(see Footnote 1.)	MIN(((x + 511)/512),32) where x = MAX(RECL + 24, BLOCKSIZE)	Initialized to zero
FAB$W_BLS	Block size	Magtape BLOCKSIZE	n if BLOCKSIZE = n	Initialized to zero
FAB$L_CTX	Context	Initialized to zero	Initialized to zero	Initialized to zero
FAB$W_DEQ	Default file extension quantity	As specified by EXTENDSIZE	n if EXTENDSIZE = n	Initialized to zero
FAB$L_DEV	Device characteristics	Returned by RMS	Returned by RMS	Returned by RMS
FAB$L_DNA	Default file specification string address	DEFAULTNAME file specification address	UNIT = nnn set to FORnnn.DAT or FOREAD.DAT, FORACCEPT.DAT, FORTYPE.DAT, or FORPRINT.DAT or DEFAULTFILE	'.DAT'
FAB$B_DNS	Default file specification string size	DEFAULTNAME file specification size	Set to length of default file specification string	Set to length of default file name string
FAB$B_FAC	File access options			
FAB$V_DEL	Allow deletions	1 if ACCESS MODIFY, Default	1 if not READONLY	1 if not HISTORY:=READONLY
FAB$V_GET	Allow reads	1 if ACCESS READ, MODIFY, SCRATCH, Default	1	1
FAB$V_PUT	Allow writes	1 if ACCESS WRITE, MODIFY, SCRATCH, APPEND, Default	1 if not READONLY	1 if not HISTORY:=READONLY
FAB$V_TRN	Allow truncates	1 if ACCESS SCRATCH	1 if not READONLY	1 if not HISTORY:=READONLY
FAB$V_UPD	Allow updates	1 if ACCESS MODIFY, SCRATCH Default	1 if not READONLY	1 if not HISTORY:=READONLY
FAB$L_FNA	File specification string address	File name	FILE = filnam if FILE present else to FORnnn, FOR$READ, FOR$ACCEPT, FOR$TYPE, FOR$PRINT, SYS$INPUT, or SYS$OUTPUT	FILE_NAME if specified, else name of file variable if external file, else zero
FAB$B_FNS	File specification string size	Set to length of file name	Set to length of file specification string	Set to length of file name string
FAB$L_FOP	File processing options			
FAB$V_CBT	Contiguous best try	1 if CONTIGUOUS	1 if INITIALSIZE = n	Initialized to zero
FAB$V_CIF	Create if nonexistent	1 if not FOR INPUT/OUTPUT	1 if STATUS = UNKNOWN	1 if HISTORY:=UNKNOWN
FAB$V_CTG	Contiguous allocation	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_DFW	Deferred write	1 if ALLOW not equal WRITE or MODIFY else 0	1	1

Settings of VAX RMS FAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

FAB$V_DLT	Delete on close service	Initialized to zero	Set at FORTRAN CLOSE, depending upon keywords DISP (OPEN or CLOSE) or STATUS (CLOSE)	Set when file is closed, depending on DISPOSITION
FAB$V_ESC	Escape, nonstandard processing	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_INP	Input, make this SYS$INPUT	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_KFO	Known file open	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_MXV	Maximize version number	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_NAM	Name block inputs	Initialized to zero	Initialized to zero	1 if terminal file reopened to enable prompting
FAB$V_NEF	Not positioned at end of file	0 if ACCESS APPEND else 1	0 if APPEND else 1	Initialized to zero
FAB$V_NFS	Not file structured	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_OFP	Output file parse	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_POS	Current position (after closed file)	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_PPF	Process permanent file	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_RCK	Read check	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_RWC	Rewind on close service	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_RWO	Rewind on open service	0 if NOREWIND else 1	Initialized to zero	Initialized to zero
FAB$V_SCF	Submit command file (when closed)	Initialized to zero	Set at FORTRAN CLOSE, depending on keywords DISP (OPEN or CLOSE) or STATUS (CLOSE)	Set when file is closed, depending on DISPOSITION
FAB$V_SPL	Spool to printer	Initialized to zero	Set at FORTRAN CLOSE, depending on keywords DISP (OPEN or CLOSE) or STATUS (CLOSE)	Initialized to zero
FAB$V_SQO	Sequential only	1 if terminal format file	1 if SEQUENTIAL or APPEND, else 0	Initialized to zero
FAB$V_SUP	Supersede	1 if FOR OUTPUT	Initialized to zero	Initialized to zero
FAB$V_TEF	Truncate at end-of-file	Initialized to zero	Initialized to zero	Initialized to zero, set to 1 after REWRITE or TRUNCATE of a sequential organization file

Settings of VAX RMS FAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

FAB$V_TMD	Temporary (marked for deletion)	1 if TEMPORARY	1 if STATUS = SCRATCH else 0	1 if nonexternal file with no FILE_NAME specified and DISPOSITION:=DELETE specified or implied
FAB$V_TMP	Temporary (file with no directory entry)	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_UFM	User file mode	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_UFO	User file open or create file only	Initialized to zero	Initialized to zero	Initialized to zero
FAB$V_WCK	Write check	Initialized to zero	Initialized to zero	Initialized to zero
FAB$B_FSZ	Fixed control area size	Initialized to zero	Initialized to zero	2 if terminal file enabled for prompting
FAB$W_IFI	Internal file identifier	Returned by RMS		
FAB$L_MRN	Maximum record number	Initialized to zero	Returned by RMS n if MAXREC = n	Returned by RMS
FAB$W_MRS	Maximum record size	Value of RECORDSIZE/MAP if given, else: 132 if terminal-format file or ORGANIZATION SEQUENTIAL 512 if ORGANIZATION VIRTUAL	If RECORDTYPE = FIXED or ORGANIZATION = RELATIVE then n; else 0	Initialized to zero RECORD_LENGTH if specified, or file component size if ORGANIZATION is not SEQUENTIAL
FAB$L_NAM	Name block address	Set to address of name block (the expanded and resultant string areas are set up, but the related file name string is not)	Set to address of name block (the expanded and resultant string areas are set up, but the related file name string is not)	Set to address of name block (the expanded and resultant string areas are set up, but the related file name string is not)
FAB$B_ORG	File organization	FAB$C_REL if RELATIVE FAB$C_IDX if INDEXED FAB$C_SEQ if SEQUENTIAL, VIRTUAL, UNDEFINED, or omitted	FAB$C_REL if RELATIVE FAB$C_IDX if INDEXED FAB$C_SEQ if SEQUENTIAL or omitted	FAB$C_REL if ORGANIZATION:=RELATIVE FAB$C_IDX if ORGANIZATION:=INDEXED else FAB$C_SEQ.
FAB$B_RAT	Record attributes			
FAB$V_FTN	FORTRAN carriage control	1 if RECORDTYPE FORTRAN	1 if FORTRAN	1 if CARRIAGE_CONTROL:=FORTRAN
FAB$V_CR	Add LF and CR	1 if default, ANY, or LIST	1 if LIST	1 if CARRIAGE_CONTROL:=LIST (default for text files)
FAB$V_BLK	Do not span blocks	1 if NOSPAN	1 if NOSPANBLOCKS	
FAB$V_PRN	Print file format	1 if terminal device	Initialized to zero	1 if terminal file enabled for prompting
FAB$B_RFM	Record format	FAB$C_FIX if FIXED or ORGANIZATION VIRTUAL FAB$C_VAR if VARIABLE or omitted FAB$C_STM if STREAM FAB$C_VFC if VFC	FAB$C_FIX if FIXED FAB$C_VAR if VARIABLE or SEGMENTED (see Footnote 2.)	FAB$C_FIX if RECORD_TYPE:=FIXED or file component type is of fixed size FAB$C_VAR if RECORD_TYPE:=VARIABLE or file is VARYING or TEXT FAB$C_VFC if terminal file enabled for prompting

Settings of VAX RMS FAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

FAB$B_RTV	Retrieval window size	Set to WINDOWSIZE	Initialized to zero	Initialized to zero
FAB$L_SDC	Spooling device characteristics	Returned by RMS	Returned by RMS	Returned by RMS
FAB$B_SHR	File sharing	As specified by ALLOW		
FAB$V_PUT	Allow other PUTs	1 if MODIFY or WRITE	1 if SHARED	1 if SHARING:=READWRITE
FAB$V_GET	Allow other GETs	1 if MODIFY or READ	1 if SHARED	1 if SHARING:=READWRITE or READONLY
FAB$V_DEL	Allow other DELETEs	1 if MODIFY	1 if SHARED	1 if SHARING:=READWRITE
FAB$V_UPD	Allow other UPDATEs	1 if MODIFY	1 if SHARED	1 if SHARING:=READWRITE
FAB$V_NIL	Allow no other operations	NONE or SCRATCH	Initialized to zero	1 if SHARING:=NONE, default if not HISTORY:=READONLY
FAB$V_UPI	User-provided interlock	Initialized to zero	1 if SHARED and SEQUENTIAL	1 if SHARING:=READWRITE and ORGANIZATION:=SEQUENTIAL
FAB$V_MSE	Multistream allowed	1 if INDEXED else 0	Initialized to zero	Initialized to zero
FAB$L_STS	Completion Status Code	Returned by RMS	Returned by RMS	Returned by RMS
FAB$L_XAB	Extended Attribute Block Address	Set to File Header Characteristics (FHC) extended attribute block address to get the longest record length. If keys are specified, FAB$ XAB points to the XABKEYs, and the last key points to XABFHC. DIGITAL may add XABs after the FHC XAB. Your USEROPEN procedure can insert XABs anywhere in the chain as long as the requirements of RMS regarding XAB ordering are satisfied.	The XAB chain has a File Header Characteristics (FHC) extended attribute block to get the longest record length (XAB$W_LRL). If the KEY = keyword is specified, KEY index definition blocks will be present. DIGITAL may add XABs in the future. Your USEROPEN procedure can insert XABs anywhere in the chain as long as the requirements of RMS regarding XAB ordering are satisfied.	The XAB chain has a File Header Characteristics (FHC) and Summary (SUM) XABs linked in for all files. If the file record declaration contains KEY attribute specifiers, one or more XABs will also be present. DIGITAL may add XABs in the future. Your USER_ACTION routine can insert XABs anywhere in the chain as long as the requirements of RMS regarding XAB ordering are satisfied.

Footnotes:

1. BASIC
   MIN (255,BKS)
   where BKS = (511 + (bucketsize * recordsize + X) + Y)/512
   where X represents any byte allowance made for record (variable format, relative control byte, or indexed header)
   Y represents bucket header control byte if indexed file.

2. FORTRAN
   FAB$C_STM if RECORDTYPE = 'STREAM'
   FAB$C_STMCR if RECORDTYPE = 'STREAM_CR'
   FAB$C_STMLF if RECORDTYPE = 'STREAM_LF'

* Source: VMS V3.0 Language Support Reference Manual, updated to VMS V4.2

Setting of VAX RMS RAB Fields by Higher-Level Language Compilers by Default at FILE OPEN

Field	Name	BASIC OPEN Keyword and Value	FORTRAN OPEN Keyword and Value	PASCAL OPEN Parameter and Value
RAB$L_BKT	Bucket code	Initialized to zero	Initialized to zero	Initialized to zero
RAB$L_CTX	Context	Initialized to zero	Initialized to zero	Initialized to zero
RAB$L_FAB	FAB address	Set to address of FAB	Set to address of FAB	Set to addres of FAB
RAB$W_ISI	Internal stream identifier	Returned by RMS	Returned by RMS	Returned by RMS
RAB$L_KBF	Key buffer address	Address of location containing current or next logical record number for sequential access segmented files	Address of longword containing a 1 if ACCESS = 'DIRECT'	May be modified for individual file operations after the file is opened
RAB$B_KRF	Key of reference	Initialized to zero	Initialized to zero	May be modified for individual file operations after the file is opened
RAB$B_KSZ	Key size	4 (changed for INDEX.)	Initialized to zero	May be modified for individual file operations after the file is opened
RAB$B_MBC	Multiblock count	Initialized to zero	If blocksize = n use (n + 511)/512	Initialized to zero
RAB$B_MBF	Multibuffer count	As specified by BUFFER	n if BUFFERCOUNT = n	Initialized to zero
RAB$L_PBF	Prompt buffer address	Initialized to zero	Initialized to zero	Initialized to zero
RAB$B_PSZ	Prompt buffer size	Initialized to zero	Initialized to zero	Initialized to zero
RAB$B_RAC	Record access mode	Initialized to zero	RAB$K_KEY if DIRECT or KEYED RAB$K_SEQ if SEQUENTIAL, APPEND, or ACCESS omitted	May be modified for individual file operations after the file is opened
RAB$L_RBF	Record address	Initialized to zero	Set later	May be modified for individual file operations after the file is opened
RAB$L_RHB	Record header buffer	Initialized to zero	Initialized to zero	Set to address of two-byte carriage-control information for terminal files enabled for prompting
RAB$L_ROP	Record processing options			

Setting of VAX RMS RAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

RAB$V_ASY	Asynchronous	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_BIO	Block I/O	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_CCO	Cancel CTRL/O	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_CVT	Convert to uppercase	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_EOF	End-of-file	1 if ACCESS APPEND	1 if APPEND	Initialized to zero
RAB$V_FDL	Fast delete	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_KGE	Key > or =	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_KGT	Key >	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_LIM	Limit	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_LOC	Locate mode	Initialized to zero	1	Initialized to zero
RAB$V_NLK	No lock	Initialized to zero	Initialized to zero	May be modified for individual file operations after the file is opened
RAB$V_NXR	Nonexistent record	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_PMT	Prompt	1 if terminal device	Initialized to zero	Initialized to zero
RAB$V_PTA	Purge type-ahead	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_RAH	Read-ahead	Initialized to zero	1	1
RAB$V_RLK	Read locked record allowed	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_RNE	Read no echo	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_RNF	Read no filter	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_TMO	Timeout	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_TPT	Truncate file after PUT	Initialized to zero	1	May be modified for individual file operations after the file is opened
RAB$V_UIF	Update if record exists	1 if ORGANIZATION VIRTUAL	1 if ACCESS = 'DIRECT'	1 if ACCESS_METHOD:=DIRECT
RAB$V_ULK	Manual unlocking	Initialized to zero	Initialized to zero	Initialized to zero
RAB$V_WBH	Write-behind	Initialized to zero	1	1
RAB$V_RSZ	Record size	Initialized to zero	Initialized to zero	May be modified for individual file operations after the file is opened

Settings of VAX RMS RAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

RAB$L_STS	Completion status code	Returned by RMS	Returned by RMS	Returned by RMS
RAB$B_TMO	Timeout period	Initialized to zero	Initialized to zero	Initialized to zero
RAB$L_UBF	User record area address	Set to address of buffer	Initialized to zero	Set to buffer address after file is opened
RAB$L_USZ	User record area size	Same as FAB$W_MRS	Initialized to zero	Set to size of buffer after file is opened

XAB Defaults

Setting of VAX-11 RMS Extended Attribute Block (XAB) Fields

Field	Name	BASIC OPEN Keyword and Value	FORTRAN and PASCAL OPEN Keyword and Value
XABKEY			
XAB$B_DTP	Data type of the key	Set to data type of the key	Set to data type of the key
XAB$B_FLG	Key options flag		
XAB$V_CHG	Changes allowed	As specified by CHANGES	0 if key is 0, else 1
XAB$V_DUP	Duplicates allowed	As specified by DUPLICATES	0 if key is 0, else 1
XAB$W_POS0	Key position	Position of key in indexed file record	Position of key in indexed file record
XAB$B_REF	Key of reference	Primary key is zero, first alternate key is one, second alternate key is two, and so on...	Primary key is zero, first alternate key is one, second alternate key is two, and so on...
XAB$B_SIZ0	Key size	Size of key	Size of key
XABFHC		Initialized	Initialized
XABSUM		Initialized	Initialized
NAM block		Initialized	Initialized

DCL COMMANDS FOR IMPLEMENTING RUN-TIME FEATURES AND SEMI-PERMANENT ATTRIBUTES

SET FILE

The SET FILE command allows various capabilities for changing the semi-permanent attributes of a file. The following commands are of particular interest.

- SET FILE/ACL allows modification of the access control list associated with a file.
- SET FILE/BACKUP determines whether the data in a file will be copied by the BACKUP utility.
- SET FILE/DATA_CHECK specifies the reliability options (read check and/or write check).
- SET FILE/ERASE_ON_DELETE ensures erasure of confidential data when a file is deleted.
- SET FILE/EXPIRATION_DATE changes the expiration date of a file (if any).
- SET FILE/EXTENSION specifies the extend quantity, or allows a change to the existing one. This can save recreating a file if you have made a minor error of judgment. Note that this figure is only used for areas which do not have an explicit extend quantity.
- SET FILE/GLOBAL_BUFFER specifies the number of global buffers to be associated with a file.
- SET FILE/OWNER_UIC changes file ownership.
- SET FILE/PROTECTION changes the protection code of a file.
- SET FILE/TRUNCATE truncates a sequential file at the ~~end-of-file marker~~. This enables you to reclaim disk space if too many blocks have been allocated and the file content has become static.

last cluster

- SET FILE/VERSION_LIMIT changes the number of versions to be retained.
- SET FILE/ENTER=new-file-spec assigns an additional name to a file. The file now has a second name, or alias, but both the original name and the alias reference the same file. For this reason, care should be taken when deleting files which have an alias. In order to keep the file but remove one of its names, use the /REMOVE qualifier with the SET FILE command.

NB $copy is selative on copy of file attributes
$ backup always copies all attributes

RMS uses a number of defaults which can either be set system-wide or for each process. Use the SHOW RMS_DEFAULT command to find out the current defaults, and then use the SET RMS_DEFAULT command to change them.

SHOW RMS_DEFAULT

```
$ SHOW RMS_DEFAULT

          MULTI-  |                MULTIBUFFER COUNTS                  | NETWORK
          BLOCK   | Indexed  Relative              Sequential          |  BLOCK
          COUNT   |                      Disk  Magtape  Unit Record   |  COUNT
Process     0     |    0        0        0       0         0           |    0
System     16     |    0        0        0       0         0           |    8

          Prolog  Extend Quantity
Process     0           0
System      0           0
```

The SHOW RMS_DEFAULT command displays the current process and system default multiblock and multibuffer counts for all types of files. It also displays the current process and system prolog level, extend quantity, and network transfer size.

Note that all of these are defaults. They are overridden by what is specified in the file attributes, or by what is specified at file open time (or later, where possible) in a program. The "default defaults" are taken from the RMS system parameters. They may be overridden for the whole system by using the SET RMS_DEFAULT/SYSTEM command (CMKRNL privilege required).

- SET RMS_DEFAULT/BLOCK_COUNT specifies the number of blocks transferred to or from a sequential file in one I/O operation. The specified count, representing the number of blocks to be allocated for each I/O buffer, can range from 0 through 127. If you specify 0, RMS uses the process default value. If this value is 0, RMS then uses the system default value. If the system default value is also 0, then RMS uses a value of 1.

  The /BLOCK_COUNT qualifier applies only to record I/O operations, not to block I/O operations.

- SET RMS_DEFAULT/BUFFER_COUNT specifies the number of buffers to be used for files of the organization and device type indicated using additional qualifiers. The specified count, representing the number of buffers to be allocated, can range from 0 through 127. ~~A positive value indicates that the specified number of buffers must be locked in the process's working set for the I/O operation.~~ A negative value indicates that the specified number of buffers must be allocated but do not have to be locked. If you specify 0, RMS uses the process default value. If this value is 0, RMS then uses the system default value. If the system default value is also 0, then RMS uses a value of 1.

  /INDEXED qualifier indicates that the specified multibuffer default is to be applied to indexed file operations.

  /RELATIVE qualifier indicates that the specified multibuffer default is to be applied to file operations on relative files.

  /SEQUENTIAL qualifier indicates that the specified multibuffer default is to be applied to all sequential file operations, including operations on disk, magnetic tape, and unit record devices.

  /SEQUENTIAL qualifier is the default if you do not specify either /RELATIVE or /INDEXED.

- SET RMS_DEFAULT/EXTEND_QUANTITY specifies the number of blocks (0 to 65535) by which files should be extended, if not already specified.

- SET RMS_DEFAULT/NETWORK_BLOCK_COUNT specifies a maximum block count for network operations on all file organizations.

- SET RMS_DEFAULT/PROLOGUE specifies the prolog version number for file creation.

Examples

```
$ SET RMS_DEFAULT/BLOCK_COUNT=24
$ SHOW RMS
```

	MULTI-BLOCK COUNT	MULTIBUFFER COUNTS					NETWORK BLOCK COUNT
		Indexed	Relative	Sequential			
				Disk	Magtape	Unit Record	
Process	24	0	0	0	0	0	0
System	16	0	0	0	0	0	8

	Prolog	Extend Quantity
Process	0	0
System	0	0

```
$ SET RMS_DEFAULT/EXTEND=50/INDEXED/BUFFER_COUNT=24
$ SHOW RMS_DEFAULT
```

	MULTI-BLOCK COUNT	MULTIBUFFER COUNTS					NETWORK BLOCK COUNT
		Indexed	Relative	Sequential			
				Disk	Magtape	Unit Record	
Process	24	24	0	0	0	0	0
System	16	0	0	0	0	0	0

	Prolog	Extend Quantity
Process	0	50
System	0	0

SPECIFYING RUN-TIME OPTIONS

All RMS options are set within the RMS control blocks that are maintained in the user P0 space. There are at least three alternative ways these options can be set from higher-level languages.

1. Some run-time options can be preset using keyword values (or defaults) available in their language OPEN statement.

2. For options not set by default by their language compiler, and not available using the OPEN keywords, languages that have a USEROPEN function in their language OPEN statement are able to directly access the VMS control blocks and set any options as part of the OPEN performed by RMS.

3. Many of the RMS run-time options are available in EDIT/FDL and can be added to an FDL file. The FDL run-time options can be implemented within program control at run time by calling the FDL$PARSE routine. This routine also returns to higher-level languages the address of the record access block (RAB) to allow a program to subsequently change RAB values. Certain RAB options are not available in FDL and can be set only by direct manipulation of RAB fields and subfields at run time.

RMS RUN-TIME OPTIONS AVAILABLE THROUGH THE FDL ADD FUNCTION

```
(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function            (Keyword)[Help] : ?

                VAX-11 FDL Editor

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

Main Editor Function            (Keyword)[Help] : ADD

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary           (Keyword)[FILE] : ACC

(Type "?" for list of Keywords)
Enter ACCESS Attribute          (Keyword)[-]    : ?

                Legal ACCESS Secondary Attributes

BLOCK_IO                yes/no
DELETE                  yes/no
GET                     yes/no
PUT                     yes/no
RECORD_IO               yes/no
TRUNCATE                yes/no
UPDATE                  yes/no

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary            Keyword)[ACCESS] : AREA 0

(Type "?" for list of Keywords)
Enter AREA 0 Attribute          (Keyword)[-]      : ?
```

Legal AREA 0 Secondary Attributes

```
ALLOCATION                  number
BEST_TRY_CONTIGUOUS         yes/no
BUCKET_SIZE                 number
CONTIGUOUS                  yes/no
EXACT_POSITIONING           yes/no
EXTENSION                   number
POSITION     qualifier      number
VOLUME                      number
```

not worth considering on public volume but may be useful for dedicated drives

```
(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary           (Keyword)[AREA 0] : CONNECT

(Type "?" for list of Keywords)
Enter CONNECT Attribute         (Keyword)[-]      : ?
```

Legal CONNECT Secondary Attributes

```
ASYNCHRONOUS                yes/no   NOLOCK                    yes/no
BLOCK_IO                    yes/no   NONEXISTENT_RECORD        yes/no
BUCKET_CODE                 number   READ_AHEAD               yes/no
CONTEXT                     number   READ_REGARDLESS          yes/no
END_OF_FILE                 yes/no   TIMEOUT_ENABLE           yes/no
FAST_DELETE                 yes/no   TIMEOUT_PERIOD           number
FILL_BUCKETS                yes/no   TRUNCATE_ON_PUT          yes/no
KEY_GREATER_EQUAL           yes/no   TT_CANCEL_CONTROL_O       yes/no
KEY_GREATER_THAN            yes/no   TT_PROMPT                yes/no
KEY_LIMIT                   yes/no   TT_PURGE_TYPE_AHEAD       yes/no
KEY_OF_REFERENCE            number   TT_READ_NOECHO           yes/no
LOCATE_MODE                 yes/no   TT_READ_NOFILTER          yes/no
LOCK_ON_READ                yes/no   TT_UPCASE_INPUT           yes/no
LOCK_ON_WRITE               yes/no   UPDATE_IF                yes/no
MANUAL_UNLOCKING            yes/no   WAIT_FOR_RECORD          yes/no
MULTIBLOCK_COUNT            number   WRITE_BEHIND             yes/no
MULTIBUFFER_COUNT           number
```

```
(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary           (Keyword)[CONNECT] : DATE

(Type "?" for list of Keywords)
Enter DATE Attribute            (Keyword)[-]      : ?
```

```
                    Legal DATE Secondary Attributes

BACKUP                          string
CREATION                        string
EXPIRATION                      string
REVISION                        string

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary           (Keyword)[DATE] : FILE

(Type "?" for list of Keywords)
Enter FILE Attribute            (Keyword)[-]    : ?

                    Legal FILE Secondary Attributes

ALLOCATION              number  MT_PROTECTION           char/num
BEST_TRY_CONTIGUOUS     yes/no  NAME                    string
BUCKET_SIZE             number  NOBACKUP                yes/no
CLUSTER_SIZE            number  NON_FILE_STRUCTURED     yes/no
CONTEXT                 number  ORGANIZATION            keyword
CONTIGUOUS              yes/no  OUTPUT_FILE_PARSE      yes/no
CREATE_IF               yes/no  OWNER                   uic
DEFAULT_NAME            string  PRINT_ON_CLOSE          yes/no
DEFERRED_WRITE          yes/no  PROTECTION              yes/no
DELETE_ON_CLOSE         yes/no  READ_CHECK              yes/no
DIRECTORY_ENTRY         yes/no  REVISION                number
EXTENSION               number  SEQUENTIAL_ONLY         yes/no
GLOBAL_BUFFER_COUNT     number  SUBMIT_ON_CLOSE         yes/no
MAX_RECORD_NUMBER       number  SUPERSEDE               yes/no
MAXIMIZE_VERSION        yes/no  TEMPORARY              yes/no
MT_BLOCK_SIZE           number  TRUNCATE_ON_CLOSE       yes/no
MT_CLOSE_REWIND         yes/no  USER_FILE_OPEN          yes/no
MT_CURRENT_POSITION     yes/no  WINDOW_SIZE             number
MT_NOT_EOF              yes/no  WRITE_CHECK             yes/no

Enter Desired Primary           (Keyword)[FILE] : KEY 0

(Type "?" for list of Keywords)
Enter KEY  0 Attribute          (Keyword)[-]    : ?
```

always 'N' for key & even if set to 'Y'

```
                Legal KEY  0 Secondary Attributes

CHANGES                       yes/no  LEVEL1_INDEX_AREA          number
DATA_AREA                     number  NAME                       string
DATA_FILL                     number  NULL_KEY                   yes/no
DATA_KEY_COMPRESSION          yes/no  NULL_VALUE                 char/num
DATA_RECORD_COMPRESSION       yes/no  POSITION                   number
DUPLICATES                    yes/no  PROLOG                     number
INDEX_AREA                    number  TYPE                       keyword
INDEX_COMPRESSION             yes/no  SEGn_LENGTH                number
INDEX_FILL                    number  SEGn_POSITION              number
LENGTH                        number

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)

Enter Desired Primary              (Keyword)[KEY 0] : REC

(Type "?" for list of Keywords)
Enter RECORD Attribute             (Keyword)[-]     : ?

                Legal RECORD Secondary Attributes

BLOCK_SPAN                    yes/no
CARRIAGE_CONTROL              keyword
CONTROL_FIELD_SIZE            number
FORMAT                        keyword
SIZE                          number

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary              (Keyword)[RECORD] : SH

(Type "?" for list of Keywords)
Enter SHARING Attribute            (Keyword)[-]     : ?
```

```
                    Legal SHARING Secondary Attributes

DELETE                      yes/no
GET                         yes/no
MULTISTREAM                 yes/no
PROHIBIT                    yes/no
PUT                         yes/no
UPDATE                      yes/no
USER_INTERLOCK              yes/no

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary              (Keyword)[SHARING] : SY

(Type "?" for list of Keywords)
Enter SYSTEM Attribute             (Keyword)[-]     : ?

                    Legal SYSTEM Secondary Attributes

DEVICE                      string
SOURCE                      keyword
TARGET                      keyword

(ACCESS AREA CONNECT DATE FILE JOURNAL
 KEY RECORD SHARING SYSTEM TITLE)
Enter Desired Primary              (Keyword)[SYSTEM] : TI

TITLE ""
Replace this existing secondary (Yes/No)[No]     :
```

Example 1. FDL Session Adding Connect Run-Time Option

```
                    VAX-11 FDL Editor
Add      to insert one or more lines into the FDL definition
Delete   to remove one or more lines from the FDL definition
Exit     to leave the FDL Editor after creating the FDL file
Help     to obtain information about the FDL Editor
Invoke   to initiate a script of related questions
Modify   to change existing line(s) in the FDL definition
Quit     to abort the FDL Editor with no FDL file creation
Set      to specify FDL Editor characteristics
View     to display the current FDL Definition


Main Editor Function              (Keyword)[Help] : ADD

                  Legal Primary Attributes

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


Enter Desired Primary             (Keyword)[FILE]: CO

(Type ? for list of keywords)
Enter CONNECT Attribute           (Keyword)[-]:    ?

                Legal CONNECT Secondary Attributes

ASYNCHRONOUS              yes/no  NOLOCK                  yes/no
BLOCK_IO                  yes/no  NONEXISTENT_RECORD     yes/no
BUCKET_CODE               number  READ_AHEAD             yes/no
CONTEXT                   number  READ_REGARDLESS        yes/no
END_OF_FILE               yes/no  TIMEOUT_ENABLE         yes/no
FAST_DELETE               yes/no  TIMEOUT_PERIOD         number
FILL_BUCKETS              yes/no  TRUNCATE_ON_PUT        yes/no
KEY_GREATER_EQUAL         yes/no  TT_CANCEL_CONTROL_O    yes/no
KEY_GREATER_THAN          yes/no  TT_PROMPT              yes/no
KEY_LIMIT                 yes/no  TT_PURGE_TYPE_AHEAD    yes/no
KEY_OF_REFERENCE          number  TT_READ_NOECHO         yes/no
LOCATE_MODE               yes/no  TT_READ_NOFILTER       yes/no
LOCK_ON_READ              yes/no  TT_UPCASE_INPUT        yes/no
LOCK_ON_WRITE             yes/no  UPDATE_IF              yes/no
MANUAL_UNLOCKING          yes/no  WAIT_FOR_RECORD        yes/no
```

```
MULTIBLOCK_COUNT          number  WRITE_BEHIND          yes/no
MULTIBUFFER_COUNT         number

Enter CONNECT Attribute              (Keyword){-}     : LOCK_ON_W

CONNECT
        LOCK_ON_WRITE
Enter value for this Secondary    (Yes/No)[-]     : Y

                  Resulting Primary Section

CONNECT
         LOCK_ON_WRITE                yes
Press RETURN to continue (^Z for Main Menu)

(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                 (Keyword)[Help] : ADD

(Access Area x Connect Date File Journal
 Key y Record Sharing System Title)
Enter Desired Primary                (Keyword)[CONNECT] : CO

(Type ? for list of keywords)
Enter CONNECT Attribute              (Keyword)[-]   : FAST

CONNECT
         FAST_DELETE
Enter value for this Secondary    (yes/no)[-] : Y

                    Resulting Primary Section
CONNECT
         FAST_DELETE      yes
         LOCK_ON_WRITE    yes
Press RETURN to continue (^Z for Main Menu)

(Add Delete Exit Help Invoke Modify Quit Set View)
Main Editor Function                 (Keyword)[Help] : EX
```

MODULE 12
CALLING RMS SERVICES DIRECTLY FROM MACRO AND HIGHER-LEVEL LANGUAGES

Major Topics

- VAX/VMS procedure calling standard
- Reporting success or failure of call
  - Calling as a function
  - RMS completion status codes
  - Testing completion status
- Passing arguments to RMS procedures

Source

BASIC — User's Guide, Chapter 9

COBOL — User's Guide, Part IV

FORTRAN — User's Guide, Chapter 4

PASCAL — User's Guide, Chapter 4

MACRO — RMS Reference, Chapters 3 and 4

THE VAX/VMS PROCEDURE CALLING STANDARD

To eliminate duplication of programming and debugging, VAX/VMS provides a set of rules specifying the interface between a calling program and a called procedure. These rules are known as the VAX/VMS Procedure Calling Standard. If this standard is followed, a procedure written in any VAX/VMS native-mode language can be called from a program written in any other native-mode language. These languages include FORTRAN, MACRO, BASIC, COBOL, and PASCAL.

The VAX/VMS Procedure Calling Standard is followed by all VAX high-level languages. It permits interlanguage calls among all VAX languages. The VAX/VMS Procedure Calling Standard specifies:

1. how to save the state of the calling program and transfer control to the called procedure,

2. how to restore the state of the calling program and return control to it,

3. how to pass arguments between the calling program and the called procedure,

4. how to report the success or failure of the called procedure to the calling program.

REPORTING SUCCESS OR FAILURE OF A CALL

Calling as a Function

- The VAX/VMS Procedure Calling Standard requires that a procedure report success or failure to the calling program in a status longword.

- For most system-supplied procedures, the program must test the status longword to detect an error. The status longword is returned to the program in two ways.

  1. As the function result in high-level languages, if the procedure is called as a function.

  2. In RO in MACRO.

- When calling RMS procedures (or any system service), even if a fatal error is encountered, it will return by default to the next instruction in the user code.

- Before returning to user program from a file or record operation, RMS also writes the condition value in the completion status code field (_STS) of the associated control block (FAB or RAB). For certain completion values, RMS returns additional information in the status value field (_STV) of the control block. The description of the RMS completion status codes in Appendix A of the RMS Reference Manual indicates which codes return information in the STV field.

RMS Completion Status Codes

The following symbolic names are associated with the numeric constant:

RMS$_fff[...f]

Refer to the RMS Reference Manual for more information. Appendix A, Table A-2 lists the symbolic names for the completion codes alphabetically. Appendix A, Table A-1 lists the completion codes by number.

Example

```
RMS$_EOF          0001827A  (Hex)
                    98938   (Decimal)
```

The symbolic names and associated numeric values are defined in an $RMSDEF module, the location of which varies among the programming languages.

Location $RMSDEF Module

Language	SYS$LIBRARY:	External Reference to Symbolic Name Resolved at
BASIC	STARLET.OLB(1)	Link — for V3.0 and above use: SYS$LIBRARY:BASIC$STARLET.TLB
COBOL	STARLET.OLB(1)	Link
FORTRAN	FORSYSDEF.TLB(2)	Compile
PASCAL	STARLET.PEN(3)	Compile
MACRO	STARLET.MLB(1)	Assemble

Use the following DCL library utilities to extract a listing from one of these libraries.

(1) $LIBRARY/EXTRACT = $RMSDEF/OUT=filespec -

SYS$LIBRARY:STARLET.MLB

(2) $LIBRARY/EXTRACT=$RMSDEF/OUT=filespec -

SYS$LIBRARY;FORSYSDEF.TLB

(3) The source for the PASCAL environment file is in SYS$LIBRARY:STARLET.PAS. This is not a library output file. Use EDIT to locate and extract module $RMSDEF.

```
$LIBRARY/EXTRACT=$RMSDEF/OUT=RMSDEF.DEF   SYS$LIBRARY:STARLET.MLB


                .MACRO  $RMSDEF,$GBL
                $DEFINI RMS,$GBL
;                       ADD RMS$_BUSY
;                       ADD RMS$_FILEPURGED
$EQU            RMS$_FACILITY       1
$EQU            RMS$V_STVSTATUS     14
$EQU            RMS$_SUC            65537
$EQU            RMS$_NORMAL         65537
$EQU            RMS$_STALL          98305
$EQU            RMS$_PENDING        98313
$EQU            RMS$_OK_DUP         98321
$EQU            RMS$_OK_IDX         98329
$EQU            RMS$_OK_RLK         98337
$EQU            RSM$_OK_RRL         98345
; OK_RRV
$EQU            RMS$_KFF            98353
$EQU            RMS$_OK_ALK         98361
$EQU            RMS$_OK_DEL         98369
$EQU            RMS$_OK_RNF         98377
$EQU            RMS$_OK_LIM         98385
$EQU            RMS$_OK_NOP         98393
$EQU            RMS$_OK_WAT         98401
$EQU            RMS$_CRE_STM        98409
$EQU            RMS$_OK_RULK        98417
$EQU            RMS$_CONTROLC       67153
$EQU            RMS$_CONTROLO       67081
$EQU            RMS$_CONTROLY       67089
$EQU            RMS$_CREATED        67097
$EQU            RMS$_SUPERSEDE      67121
$EQU            RMS$_OVRDSKQUOTA    67177
$EQU            RMS$_FILEPURGED     67193
$EQU            RMS$_BOF            98712
$EQU            RMS$_RNL            98720
$EQU            RMS$_RTB            98728
$EQU            RMS$_TMO            98736
$EQU            RMS$_TNS            98744
$EQU            RMS$_BES            98752
$EQU            RMS$_PES            98760
$EQU            RMS$_ACT            98906
$EQU            RMS$_DEL            98914
$EQU            RMS$_INCOMPSHR      98922
$EQU            RMS$_DNR            98930
$EQU            RMS$_EOF            98938       <-----
$EQU            RMS$_FEX            98946
$EQU            RMS$_FLK            98954
$EQU            RMS$_FNF            98962
$EQU            RMS$_PRV            98970
$EQU            RMS$_REX            98978
$EQU            RMS$_RLK            98986
$EQU            RMS$_RNF            98994
$EQU            RMS$_WLK            99002
```

```
$LIBRARY/EXTRACT=$RMSDEF/OUT=RMSDEF.DEF   SYS$LIBRARY:FORSYSDEF.TLB

!*** MODULE $RMSDEF ***
!
!  This SDL File Generated by VAX-11 Message V04-00 on 15-SEP-1984 22:53:50.83
!
!       .TITLE  RMSDEF -RMS COMPLETION CODES                                  *
! *                                                                           *
! *  COPYRIGHT (C) 1978, 1980, 1982, 1984 BY                                  *
! *  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                   *
! *  ALL RIGHTS RESERVED.                                                     *
! *

  . . .

! --
        PARAMETER RMS$_FACILITY = '00000001'X
        PARAMETER RMS$V_STVSTATUS = '0000000E'X ! MOVE TO BIT 14 OF THE
!  STATUS CODE IT INDICATES
!  THAT STV CONTAINS A SECONDARY
!  STATUS CODE.
        PARAMETER RMS$_SUC = '00010001'X
        PARAMETER RMS$_NORMAL = '00010001'X

! +
!
!       SUCCESS CODES
!
! -
!  BIT 16 = BIT 15 = 1
        PARAMETER RMS$_STALL = '00018001'X
!  (NOTE: USER NEVER RECEIVES THIS CODE)
        PARAMETER RMS$_PENDING = '00018009'X
        PARAMETER RMS$_OK_DUP = '00018011'X
        PARAMETER RMS$_OK_IDX = '00018019'X

 . . .

! +
!
!       ERROR CODES - WITHOUT STV
!
! -
!  BIT 16 = BIT 15 = 1, BIT 14 = 0
        PARAMETER RMS$_ACT = '0001825A'X
        PARAMETER RMS$_DEL = '00018262'X
        PARAMETER RMS$_INCOMPSHR = '0001826A'X
        PARAMETER RMS$_DNR = '00018272'X
        PARAMETER RMS$_EOF = '0001827A'X          <-----
        PARAMETER RMS$_FEX = '00018282'X
        PARAMETER RMS$_FLK = '0001828A'X
        PARAMETER RMS$_FNF = '00018292'X
        PARAMETER RMS$_PRV = '0001829A'X
        PARAMETER RMS$_REX = '000182A2'X
        PARAMETER RMS$_RLK = '000182AA'X
        PARAMETER RMS$_RNF = '000182B2'X
!  (RECORD NEVER WAS IN FILE, OR HAS BEEN DELETED.)
        PARAMETER RMS$_WLK = '000182BA'X

 . . .
```

Excerpt Extracted From SYS$LIBRARY:STARLET.PAS Using EDIT

```
(*** MODULE $RMSDEF ***)

(*                                                                              *)
(* This SDL File Generated by VAX-11 Message V04-00 on 15-SEP-1984 22:53:50.83
(*                                                                              *)
(*      .TITLE  RMSDEF -RMS COMPLETION CODES                                    *)
(******************************************************************************)
(**                                                                            **)
(**  COPYRIGHT (C) 1978, 1980, 1982, 1984 BY                                   **)
(**  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                    **)
(**  ALL RIGHTS RESERVED.                                                      **)
(**                                                                            **)

 . . .

CONST   RMS$_FACILITY = 1;
        RMS$V_STVSTATUS = 14;           (* MOVE TO BIT 14 OF THE               *)

(*  STATUS CODE IT INDICATES                                                    *)
(*  THAT STV CONTAINS A SECONDARY                                               *)
(*  STATUS CODE.                                                                *)

CONST   RMS$_SUC = 65537;
        RMS$_NORMAL = 65537;

(*+                                                                             *)
(*                                                                             *)
(*      SUCCESS CODES                                                          *)
(*                                                                             *)
(*-                                                                            *)
(*  BIT 16 = BIT 15 = 1                                                        *)

CONST   RMS$_STALL = 98305;

(*  (NOTE: USER NEVER RECEIVES THIS CODE                                       *)

CONST   RMS$_PENDING = 98313;
        RMS_OK_DUP = 98321;
        RMS_OK_IDX = 98329;

 . . .

(*      ERROR CODES - W1 HOU~ STV                                              *)
(*                                                                             *)
(*-                                                                            *)
(*  BIT 16 = BIT 15 = 1, BIT 14 = 0

CONST   RMS$_ACT = 98906;
        RMS$_DEL = 98914;
        RMS$_INCOMPSHR = 98922;
        RMS$_DNR = 98930;
        RMS$_EOF = 98938;           <-----
        RMS$_FEX = 98946;
        RMS$_FLK = 98954;
        RMS$_FNF = 98962;
        RMS$_PRV = 98970;
        RMS$_REX = 98978;
        RMS$_RLK = 98986;
        RMS$_RNF = 98994;

(*  (RECORD NEVER WAS IN FILE, OR HAS BEEN DELETED.)                           *)

 . . .
```

Testing Completion Status -- Calling RMS Procedures

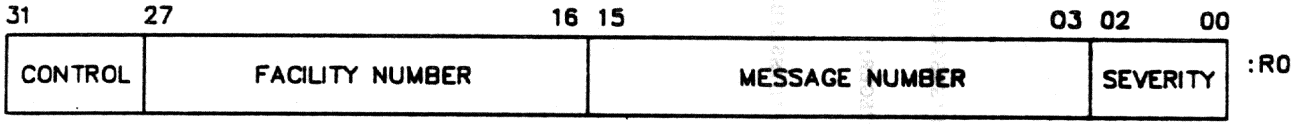
	BASIC	COBOL	FORTRAN	PASCAL
1. Call procedure as	Function	Function	Function	Function
2. Declare function name	EXTERNAL LONG function name	--	INTEGER*4 name	[INHERIT ('SYS$LIBRARY:STARLET')]
3. Declare variable to receive status code as function result	LONG STAT	01 STAT COMP PIC S9(9)	INTEGER*4 STAT	STAT : INTEGER;
4. If you wish to test for specific status code (optional)	EXTERNAL LONG CONSTANT symbol_name	01 symbol_name PIC S9(9) COMP VALUE IS EXTERNAL symbol_name	INCLUDE '($RMSDEF)'	[INHERIT ('SYS$LIBRARY:STARLET')]

A programmer can test for:

1. Success or failure <--- overall test -- binomial evaluation

2. Specific conditions <--- test of specific condition codes (symbolic names)

Format of Status Value



BU-2483

Severity Codes

Severity	Meaning
0	Warning
1	Success
2	Error
3	Information
4	Severe Error
5-7	Reserved for Future Use

Example 1. Testing Specific Condition Codes

(Sheet 1 of 2)

BASIC

```
01      EXTERNAL LONG FUNCTION SYS$GET
02      EXTERNAL LONG CONSTANT RMS$_EOF
03      DECLARE LONG STAT
04      .
05      .
06      STAT = SYS$GET(...)
07      handle error IF (STAT = RMS$_EOF)
08      .
09      .
```

COBOL

```
10      DATA DIVISION.
11      WORKING_STORAGE SECTION.
12        01  STAT       PIC S9(9)       COMP.
13        01  RMS$_EOF   PIC S9(9)       COMP    VALUE EXTERNAL RMS$_EOF.
14      PROCEDURE DIVISION.
15      BEGIN.
16      .
17      .
18      CALL 'SYS$GET' USING ... GIVING STAT.
19      IF STAT IS EQUAL TO RMS$_EOF THEN  handle error.
20      ...
```

FORTRAN

```
21      INTEGER*4         SYS$GET, STAT
22      INCLUDE           '($RMSDEF)'
23      .
24      .
25      STAT = SYS$GET(...)
26      IF (STAT .EQ. RMS$_EOF) THEN
27              handle error
28      END IF
29      .
30      .
```

Example 1 (Sheet 2 of 2)

PASCAL

```
31      [INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM GO_GET_IT;
32      VAR
33         STAT : INTEGER;
34      ...
35      BEGIN
36      ...
37      STAT := $GET(...);
38      IF (STAT = RMS$_EOF) THEN handle error;
39      ...
40      END.
```

MACRO

```
41      $GET    ...
42      CMPL    #RMS$_EOF, R0
43      BEQLU   handle_error
44      ...
```

Example 2. Testing Overall Error -- Test of Low-Order Bit in STATUS Returned

(Sheet 1 of 2)

BASIC

```
01      EXTERNAL LONG FUNCTION SYS$GET
02      .
03      DECLARE LONG STAT
04      .
05      .
06      STAT = SYS$GET(...)
07      handle error IF (STAT AND 1%) = 0%
08      .
09      .
```

COBOL

```
10      DATA DIVISION.
11      WORKING-STORAGE SECTION.
12        01  STAT       PIC S9(9)        COMP.
13      .
14      PROCEDURE DIVISION.
15      BEGIN.
16      .
17      .
18      CALL 'SYS$GET' USING ... GIVING STAT.
19      IF STAT IS FAILURE THEN  handle error.
20      ...
```

FORTRAN

```
21      INTEGER*4        SYS$GET, STAT
22      .
23      .
24      .
25      STAT = SYS$GET(...)
26      IF (.NOT.STAT) THEN
27              handle error
28      END IF
29      .
30      .
```

Example 2 (Sheet 2 of 2)

PASCAL

```
31      [INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM GO_GET_IT;
32      VAR
33         STAT : INTEGER;
34      ...
35      BEGIN
36      ...
37      STAT := $GET(...);
38      IF NOT ODD (STAT) THEN handle error;
39      ...
40      END.
```

MACRO

```
41      $GET    ...
42      BLBC    R0, handle_error
43      .
44      ...
```

RMS services are considered system services for the purpose of generating system service exceptions on errors. You can choose whether to test and handle errors in your program or set the system service failure exception mode (using the Set System Failure Exception Mode system service, SYS$SETSFM) to have failures automatically signaled. For most applications, especially if a high-level language is used, testing and handling errors in your program are the preferred method. If you test for error conditions in your program, be sure to disable any unwanted system service exception generation.

Three general SS$_ condition codes may be encountered.

SS$_INSFARG	Insufficient # arguments in call
SS$_ILLSER	Nonexistent service called
SS$_ACCVIO	Argument list cannot be read

Note that if the FAB or RAB is invalid or inaccessible, then the error completion routine will not attempt to store the error code in the STS field of an invalid control block structure. The following errors can be detected only by testing the completion code returned by the function call (or by enabling system service failure exception mode), following the completion of an RMS operation (even if an error completion AST has been specified):

RMS$_BLN	Invalid block length field (either FAB or RAB)
RMS$_BUSY	User structure (FAB/RAB) still in use
RMS$_FAB	FAB not writeable or invalid block ID field
RMS$_RAB	RAB not writeable or invalid block ID field
RMS$_STR	User structure (FAB/RAB) became invalid during operation

These completion codes indicate that the FAB or RAB is invalid or inaccessible. These completion codes are usually rare and, if they occur at all, would most likely occur during initial program debugging and testing. Examine the completion value in R0 or returned by the function call (instead of the STS field of the FAB or RAB) for the completion codes listed above.

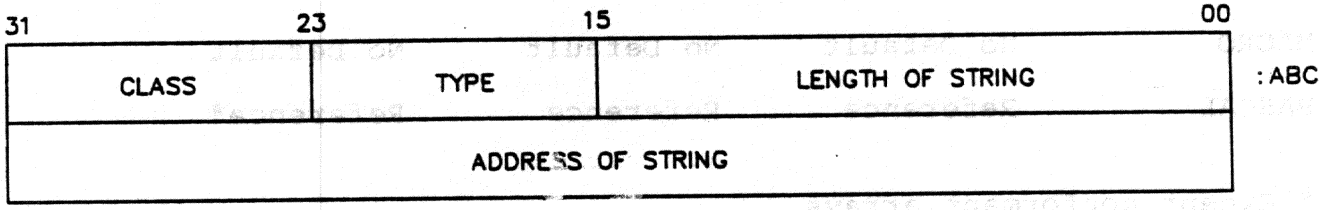
When signaling RMS errors, both the STS and STV fields of the appropriate structure (FAB or RAB) should be supplied. This will cause all relevant information to be displayed in the error message text, including additional information regarding the error status in the STV field. For the file processing and file naming services, use the STS and STV fields of the specified FAB (use the old FAB for the Rename service). For record processing and block I/O processing services, use the STS and STV fields of the corresponding RAB.

The recommended way to signal RMS errors is to provide both the STS and STV fields of the RAB or FAB as arguments to the Run-Time Library (RTL) routine LIB$SIGNAL (or LIB$STOP). Certain VAX languages provide a built-in means of signaling errors, such as by providing a system-defined function. For a more detailed explanation of condition signaling and invoking RTL routines, see the VAX/VMS Run-Time Library Routines Reference Manual.

PASSING ARGUMENTS TO PROCEDURES

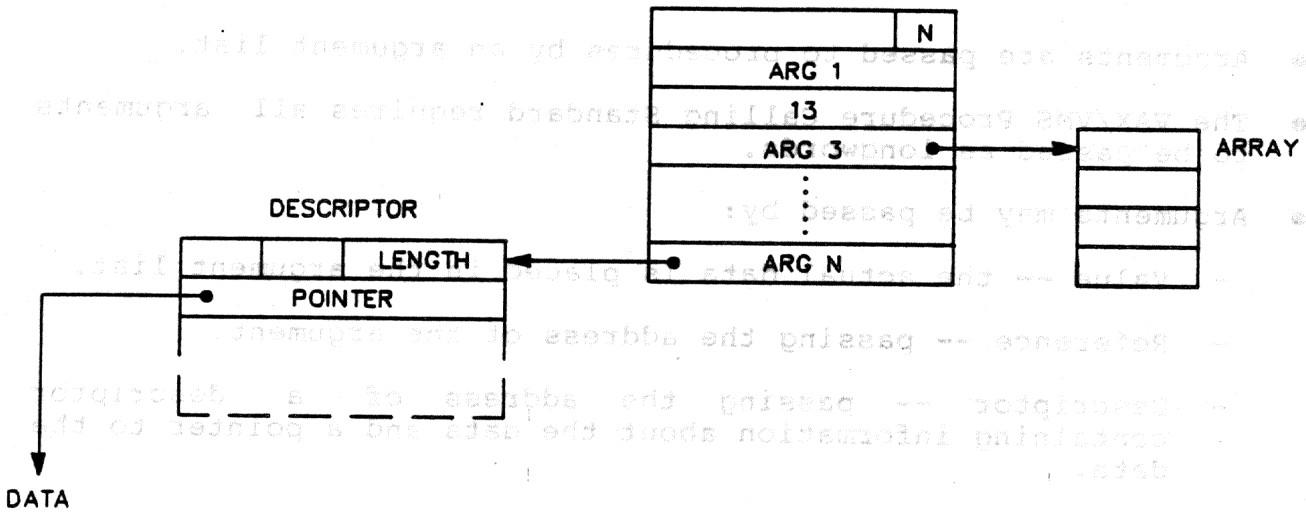
- Arguments are passed to procedures by an argument list.
- The VAX/VMS Procedure Calling Standard requires all arguments to be passed as longwords.
- Arguments may be passed by:
  - Value -- the actual data is placed in the argument list.
  - Reference -- passing the address of the argument.
  - Descriptor -- passing the address of a descriptor containing information about the data and a pointer to the data.
- The format of passed arguments must match the format expected by the called procedure.
  - In calls to user-written procedures, arguments should match the default for the language in which the procedure is written.
  - In calls to system-supplied procedures, arguments should match the format specified by the VAX/VMS documentation.

Layout of the String Descriptor for ABC



BU-2484

Format of an Argument List



BU-2485

Default Argument Passing and Receiving Mechanisms

Language	Numeric	Character	Array
BASIC	Reference	Descriptor	Descriptor
COBOL	Reference	Reference	N/A
FORTRAN	Reference	Descriptor	Reference
MACRO	No Default	No Default	No Default
PASCAL	Reference	Reference	Reference*

* Except conformant arrays

Argument Passing Specifiers

Description	Mechanism Specifier			
	BASIC	COBOL	FORTRAN	PASCAL
Passed by Value	BY VALUE	BY VALUE	%VAL	%IMMED
Passed by Reference	BY REF	BY REFERENCE or BY CONTENT	%REF	%REF
Passed by Descriptor	BY DESC	BY DESCRIPTOR	%DESCR	%DESCR
Passed by String Descriptor	BY DESC	BY DESCRIPTOR	%DESCR	%STDESCR

Example 3. Passing BY VALUE

```
BASIC:      DECLARE LONG abc
             .
             .
             .
            abc = 35
             .
             .
             .
            stat = SUB1 (abc BY VALUE)

COBOL:      ...
            01 abc PIC 9(5) VALUE is 35.
             .
             .
             .
            CALL SUB1 USING BY VALUE abc
                       GIVING STAT.

FORTRAN:    INTEGER*4 abc/35/
             .
             .
             .
            stat = SUB1 (%VAL(abc))

PASCAL:     ...
            VAR
               abc : INTEGER := 35;
            ...
            FUNCTION SUB1 (%IMMED xyz:INTEGER):
                       INTEGER; EXTERN;
             .
             .
             .
            STAT := SUB1 (abc);

MACRO:      abc: .LONG 35

             .
             .
            PUSHL abc
            CALLS #1, SUB1
```

Example 4. Passing BY REFERENCE

BASIC:

```
DECLARE LONG abc
 .
 .
 .
abc = 35
 .
 .
 .
stat = SUB1  (abc BY REF)
                or
             (abc)
```

COBOL:

```
01 abc PIC 9(5) VALUE is 35.
 .
 .
 .
CALL SUB1 USING BY REFERENCE abc
                          or
                         abc
          GIVING STAT.
```

FORTRAN:

```
INTEGER *4  abc/35/
 .
 .
 .
STAT = SUB1 (%REF(abc))
                or
             (abc)
```

PASCAL:

```
VAR
abc : INTEGER := 35;
...
FUNCTION SUB1 (%REF xyz: INTEGER): INTEGER;
                                    EXTERN;
                      or
             (xyz : INTEGER);
 .
 .
 .
STAT := SUB1 (abc);
```

MACRO:

```
abc: .LONG 35
 .
 .
 .
PUSHAL abc
CALLS  #1, SUB1
```

Example 5. Passing BY DESCRIPTOR

```
BASIC:      DECLARE STRING abc
             .
             .
             .
            abc = '35'
             .
             .
             .
            STAT = SUB1 (abc BY DESC)
                          or
                        (abc)

COBOL:      01 abc PIC x(2) VALUE is '35'.
             .
             .
             .
            CALL SUB1 USING BY DESCRIPTOR abc
                      GIVING STAT.

FORTRAN:    CHARACTER*2 abc /'35'/
             .
             .
             .
            STAT = SUB1 (%DESCR(abc))
                          or
                        (abc)

PASCAL:     VAR
            abc : PACKED ARRAY [1..2] OF CHAR;
             .
             .
             .
            FUNCTION  SUB1 (xyz:[CLASS_S] PACKED ARRAY [1..u:
                      INTEGER] of CHAR): INTEGER; EXTERN;
             .
             .
             .
            abc  = '35';
            STAT. = SUB1 (abc);

MACRO:      abc: .ASCID /35/
             .
             .
             .
            PUSHAQ      abc
            CALLS       #1, SUB1
```

RMS routines -- The passing mechanism is identified in the second sentence of the argument description.

Passing Method	Description
Immediate Value	"The value of" or omission of the word "address"
Reference	"Address of"
Descriptor	"Address of a descriptor"

Example 6. Passing Mechanism for SYS$SETDDIR Arguments

Arguments	Passing Mechanism
new_dir_addr	_____
length_addr	_____
cur_dir_addr	_____

BASIC	Include commas as placeholders for each omitted argument.
COBOL	BY VALUE 0 for each omitted argument.
FORTRAN	Include commas as placeholders for each omitted argument.
PASCAL	Positional syntax: If default value is declared in formal declaration, then commas are not needed. Keyword syntax is allowed for PASCAL.
MACRO	For most RMS calls, MACROS exists to assist in the construction of the argument list. However, in a few cases (for example, SYS$SETDDIR), RMS procedures are called without an RMS MACRO and a zero should be passed for the optional argument. Keyword syntax is allowed for MACRO.

Example 7. Program Calling RMS Procedure SYS$SETDDIR

(Sheet 1 of 2)

BASIC

SETDDIR.BAS

```
1       10  !
2           !
3           !     This program calls the RMS procedure $SETDDIR to
4           !     change the default directory for the process.
5           !
6               EXTERNAL INTEGER FUNCTION SYS$SETDDIR
7               DECLARE LONG     stat,&
8                       STRING   dir
9               dir='[course.v4prog.bas]'
10              stat=SYS$SETDDIR(dir,,)
11              PRINT "Error" IF (stat AND 1%)=0%
12              END
```

COBOL

SETDDIR.COB

```
1       *
2       IDENTIFICATION DIVISION.
3       *
4       PROGRAM-ID. SETDDIR.
5       *
6       *   This program calls the RMS procedure $SETDDIR to change
7       *   the default directory for the process.
8       *
9       DATA DIVISION.
10      WORKING-STORAGE SECTION.
11
12      01  DIRECTORY PIC  X(17)          VALUE
13                                         '[COURSE.PROG.COB]'.
14      01  STAT      PIC S9(9)   COMP.
15
16      PROCEDURE DIVISION.
17      BEGIN.
18          CALL 'SYS$SETDDIR' USING BY DESCRIPTOR DIRECTORY
19                                   BY VALUE 0
20                                   BY VALUE 0
21                             GIVING STAT.
22          IF STAT IS FAILURE DISPLAY 'ERROR'.
23          STOP RUN.
```

FORTRAN

SETDDIR.FOR

```
1       C
2       C
3       C   This program calls the RMS procedure $SETDDIR to change
4       C   the default directory for the process.
5       C
6           IMPLICIT INTEGER (A - Z) or INTEGER SYS$SETDDIR, STAT
7           CHARACTER*17 DIR /'[COURSE.PROG.FDR]'/
8           STAT = SYS$SETDDIR (DIR,,)
9           IF (.NOT. STAT) TYPE *, 'ERROR'
10          END
```

Example 7 (Sheet 2 of 2)

PASCAL

```
 1          PROGRAM setddir( OUTPUT);
 2
 3          (*   SETDDIR.PAS                                              *)
 4          (*   This program calls the RMS procedure $SETDDIR to         *)
 5          (*   change the default directory for the process.            *)
 6
 7          TYPE word_integer = [WORD] 0..65535;
 8          VAR dir_status: INTEGER;
 9
10          FUNCTION SYS$SETDDIR(
11             new_dir: [CLASS_S] PACKED ARRAY [1..u:INTEGER] of CHAR;
12             old_dir_len: word_integer := %IMMED 0;
13             old_dir: VARYING [line2] OF CHAR := %IMMED 0):
14                 INTEGER; EXTERN;
15
16          BEGIN
17             dir_status:= SYS$SETDDIR( ('[COURSE.V4PROG.PAS]');
18             IF NOT ODD( dir_status)
19                THEN WRITELN( 'Error in SYS$SETDDIR call.')
20          END.
```

MACRO
SETDDIR.MAR

```
 1          ;
 2          ;
 3               .TITLE  SETDDIR
 4          ;
 5          ;    This program calls the RMS procedure $SETDDIR to change
 6          ;    the default directory for the process.
 7          ;
 8               .PSECT NONSHARED_DATA PIC,NOEXE,LONG
 9          DIR:        .ASCID  /[COUR[illegible].PROG.MAC]/
10               .PSECT CODE              PIC,SHR,NOWRT,LONG
11               .ENTRY START,^M<>
12               PUSHL  #0
13               PUSHL  #0
14               PUSHAQ DIR
15               CALLS  #3,SYS$SETDDIR
16               MOVL   #SS$_NORMAL,R0 or $EXIT_S
17               RET
18               .END   START
```

MODULE 13
ALTERNATE APPROACHES TO ACCESSING RMS CONTROL BLOCKS DIRECTLY — LANGUAGE EXAMPLES

Major Topics

- USEROPEN function supported by some higher-level languages
- Call FDL$PARSE to set up and initialize control blocks

Source

RMS Reference Manual — Chapter 5 (FAB), Chapter 7 (RAB)

USEROPEN FUNCTION OR REGULAR I/O

The USEROPEN function is an alternative approach to accessing RMS control blocks directly. The USEROPEN function is called as part of the OPEN command. Thereafter, regular language I/O is used.

Language	Open Keyword	Value	Language Function May Be Written In
BASIC	USEROPEN	function-name	BASIC
COBOL	not available	--	--
FORTRAN	USEROPEN	function-name	FORTRAN
PASCAL	USER_ACTION	function-name	PASCAL

The USEROPEN function has three arguments:

1. Arg 1 -- FAB address
2. Arg 2 -- RAB address

} passed to user by RMS

3. Arg 3 -- File # - BASIC
   Unit # - FORTRAN
   Filename identifier - PASCAL

} user defined

The Run-Time Library sets up the following VAX-11 RMS control structures before calling the USEROPEN function:

FAB File Access Block
RAB Record Access Block
NAM Name Block
XAB FHC Extended Attributes Block
.
.
. Any key XABs specified for index files

For BASIC and FORTRAN USEROPEN functions, only the RAB structure remains allocated after the OPEN. Therefore, do not store the addresses of the FAB, XAB, or NAM blocks for later use.

The PASCAL USER_ACTION parameter differs from the BASIC/FORTRAN USEROPEN keyword in that you may include user-written procedures with either OPEN or CLOSE or both, while USEROPEN is limited to OPEN. Further, the USER_ACTION function may be written in PASCAL and is allowed to up-level address. The RMS RAB, FAB, and NAM blocks are allocated dynamically and remain defined as long as the file is opened. Therefore, your program may store the addresses of these structures for later use.

To assign values to a field or read values in a field in the RMS control block (FAB, RAB), the user needs to be able to declare a structure in the programming language to match that of RMS.

BASIC

None provided in system library. Set up your own and 'include' it in the source file.

```
%INCLUDE 'FABRABDEF.BAS'
```

FAB$TYPE
RAB$TYPE

FORTRAN

```
INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
```

SYS$LIBRARY: FORSYSDEF.TLB

PASCAL

```
[INHERIT ('SYS$LIBRARY: STARLET')]
```

FAB$TYPE
RAB$TYPE

Now included in BASICSTARTLET.TLB

Example 1. USEROPEN Alternative (if available) or Regular I/O

BASIC (Sheet 1 of 5)

```
10 OPTION  TYPE = EXPLICIT                                    BASOPEN_INDX.BAS
    !
    !   This BASIC program uses USEROPEN function to open
    !   an indexed file.  Thereafter, all access to the file is
    !   done using regular BASIC I/O.  User is prompted for
    !   SEQ_NO of record to be retrieved randomly and given
    !   the option of deleting any record retrieved.
    !
    !   Program assumes FILE SHARING and RECORD LOCKING
    !
    ON ERROR GO TO err_check

    DECLARE SINGLE  CONSTANT        TIME_WAIT=10.0
    DECLARE STRING  CONSTANT  RIGHT_JUSTIFY = "#######"

    RECORD ACCOUNT$TYPE
            STRING  SEQ_NO=7
            STRING  LAST_NAME=15
            STRING  FIRST_NAME=10
            STRING  SOC_SEC=9
            STRING  STREET=18
            STRING  CITY=14
            STRING  STATE=2
            STRING  ZIP_CODE=5
    END RECORD ACCOUNT$TYPE

    MAP (INREC)      ACCOUNT$TYPE    IN_REC

    MAP (FILE_NAME) STRING  FILENAME=80

    MAP (KEYVALUE)  STRING  KEY_VALUE=7
    MAP (DELFLAG)   STRING  DELETE_FLAG=1

    DECLARE LONG    KEY_IN

    EXTERNAL SUB    LIB$INIT_TIMER,&
                    LIB$SHOW_TIMER,&
                    LIB$WAIT

 start:

    CALL LIB$INIT_TIMER()

    INPUT 'Enter filename';filename

    OPEN filename FOR INPUT AS FILE #1%, &
                    ORGANIZATION INDEXED, &
                    MAP INREC, &
                    ACCESS MODIFY, &
                    ALLOW MODIFY, &
                    USEROPEN OPENFILE
```

```
52
53      PRINT "*********************************************"
54      PRINT "     Hit <CR> or enter zero to stop run"
55      PRINT "*********************************************"
56
57      INPUT 'Enter SEQ_NO';KEY_IN
58
59      WHILE (KEY_IN > 0)
60
61        KEY_VALUE = FORMAT$(KEY_IN,RIGHT_JUSTIFY)
62
63  100           GET #1%, KEY #0% EQ KEY_VALUE
64
65        PRINT IN_REC::SEQ_NO;TAB(9);IN_REC::SOC_SEC;TAB(20);&
66                      IN_REC::LAST_NAME
67
68        INPUT 'Do you wish to delete this record? (Y/<CR>)'; &
69                                          DELETE_FLAG
70        IF (DELETE_FLAG = 'Y' OR DELETE_FLAG = 'y') THEN
71              DELETE #1%
72
73  150           INPUT 'Enter SEQ_NO';KEY_IN
74
75      NEXT
76
77   cleanup:
78      CLOSE #1%
79      CALL LIB$SHOW_TIMER()
80      GOTO done
81
82   err_check:
83   !
84   !  ERR = 155  <---  record not found on random get
85   !  ERR = 154  <---  record/bucket locked
86   !
87      SELECT ERR
88       CASE 155
89              PRINT "Record NOT FOUND with SEQ_NO = ";KEY_VALUE
90              RESUME 150
91       CASE 154
92              PRINT "Record currently LOCKED - ", &
93                      "will try again shortly"
94              CALL LIB$WAIT (TIME_WAIT)
95              RESUME 100
96        CASE ELSE
97              ON ERROR GO TO 0
98      END SELECT
99
100  done:      END
101
```

```
102  175        FUNCTION LONG OPENFILE (FAB$type FAB,&
103                                     RAB$type RAB,&
104                                     LONG channel)
105
106      EXTERNAL LONG FUNCTION SYS$OPEN
107      EXTERNAL LONG FUNCTION SYS$CONNECT
108
109    ! Include record definitions of FAB$type and RAB$type
110    !    and external constants
111
112      %INCLUDE 'FABRABDEF.BAS'
113
114      DECLARE LONG ret_stat
115
116     !  Extension # blocks if file is extended
117      FAB::FAB$W_DEQ = 10
118
119     !  File access desired for USER
120     !        FAB::FAB$B_FAC = (FAB::FAB$B_FAC OR FAB$M_DEL OR FAB$M_GET &
121     !                  OR FAB$M_PUT OR FAB$M_UPD)
122
123     !  File options desired
124      FAB::FAB$L_FOP = (FAB::FAB$L_FOP OR FAB$M_DFW)  !  deferred write
125
126     !  # global buffers if wish to use them or set to zero if wish to
127     !        override global and use local buffers if someone already
128     !        has file opened with global buffers enabled
129     !  The following statement must be inserted in source after
130     !  SYS$OPEN call but prior to SYS$CONNECT call.
131     !        FAB::FAB$W_GBC = ?
132
133     !  Sharing attributes - what others can do or set TO FAB$M_SHRNIL
134     !        FAB::FAB$B_SHR = (FAB::FAB$B_SHR OR FAB$M_SHRPUT &
135     !                  OR FAB$M_SHRGET OR FAB$M_SHRDEL       &
136     !                  OR FAB$M_SHRUPD)
137
138      ret_stat = SYS$OPEN(FAB,,)
139           CALL LIB$STOP(ret_stat BY VALUE) IF (ret_stat AND 1%)=0%
140
141
142    !  Specify number of local buffers you want RMS to allocate on CONNECT
143      RAB::RAB$B_MBF  =  3
144
145    !  Enable any record processing options to be used for entire run
146      RAB::RAB$L_ROP  =  (RAB::RAB$L_ROP OR RAB$M_FDL)  !  fast delete
147
148
149      ret_stat = SYS$CONNECT(RAB,,)
150      CALL LIB$STOP(ret_stat BY VALUE) IF (ret_stat AND 1%)=0%
151
152      OPENFILE = ret_stat
153
154      END FUNCTION
```

BASIC (Sheet 4 of 5)

```
1       !+
2       !       FABRABDEF.BAS
3       !
4       !       RMS Data Structures Definitions
5       !+
6
7       RECORD fab$TYPE
8
9       BYTE fab$b_bid
10      BYTE fab$b_bln
11      WORD fab$w_ifi
12      LONG fab$l_fop
13      LONG fab$l_sts
14      LONG fab$l_stv
15      LONG fab$l_alq
16      WORD fab$w_deq
17      BYTE fab$b_fac
18      BYTE fab$b_shr
19      LONG fab$l_ctx
20      BYTE fab$b_rtv
21      BYTE fab$b_org
22      BYTE fab$b_rat
23      BYTE fab$b_rfm
24      LONG fab$l_jnl
25      LONG fab$l_xab
26      LONG fab$l_nam
27      LONG fab$l_fna
28      LONG fab$l_dna
29      BYTE fab$b_fns
30      BYTE fab$b_dns
31      WORD fab$w_mrs
32      LONG fab$l_mrn
33      WORD fab$w_bls
34      BYTE fab$b_bks
35      BYTE fab$b_fsz
36      LONG fab$l_dev
37      LONG fab$l_sdc
38      WORD fab$w_gbc
39      BYTE fab$b_acm
40      BYTE fab$b_rcf
41      BYTE fill(4)
42
43      END RECORD
44
45
46      RECORD rab$TYPE
47
48      BYTE rab$b_bid
49      BYTE rab$b_bln
50      WORD rab$b_isi
51      LONG rab$l_rop
52      LONG rab$l_sts
```

```
53      LONG rab$l_stv
54      RFA  rab$w_rfa                  ! 6 bytes
55      WORD fill
56      LONG rab$l_ctx
57      WORD fill
58      BYTE rab$b_rac
59      BYTE rab$b_tmo
60      WORD rab$w_usz
61      WORD rab$w_rsz
62      LONG rab$l_ubf
63      LONG rab$l_rbf
64      LONG rab$l_rhb
65      VARIANT
66      CASE
67          LONG rab$l_kbf
68          BYTE rab$b_ksz
69      CASE
70          LONG rab$l_pbf
71          BYTE rab$b_psz
72      END VARIANT
73      BYTE rab$b_krf
74      BYTE rab$b_mbf
75      BYTE rab$b_mbc
76      VARIANT
77      CASE
78          LONG rab$l_bkt
79      CASE
80          LONG rab$l_dct
81      END VARIANT
82      LONG rab$l_fab
83      LONG rab$l_xab
84      END RECORD
85
86    ! +
87    ! declarations of FAB and RAB CONSTANTS
88    ! +
89      EXTERNAL BYTE CONSTANT  FAB$M_DEL,&
90                              FAB$M_GET,&
91                              FAB$M_PUT,&
92                              FAB$M_UPD,&
93                              FAB$M_DFW,&
94                              FAB$M_SHRPUT,&
95                              FAB$M_SHRGET,&
96                              FAB$M_SHRDEL,&
97                              FAB$M_SHRUPD,&
98                              FAB$M_MSE
99
100     EXTERNAL BYTE CONSTANT  RAB$C_KEY,&
101                             RAB$C_SEQ,&
102                             RAB$C_RFA,&
103                             RAB$M_FDL
```

```
1  *                                                    COBOPEN_INDX.COB
2  *
3  *  COBOL PROGRAM which opens an indexed file.  All access
4  *  to the file is done using regular COBOL I-O.  User
5  *  is prompted for SEQ_NO of record to be retrieved
6  *  randomly and given option of deleting any record
7  *  retrieved.
8  *
9  *  COBOL currently does not support a USEROPEN function call
10 *       from its language.
11 *
12 *  Program assumes FILE SHARING and RECORD LOCKING
13 *
14 IDENTIFICATION DIVISION.
15 PROGRAM-ID.  COBOPEN_INDX.
16 ENVIRONMENT DIVISION.
17 INPUT-OUTPUT SECTION.
18 FILE-CONTROL.
19     SELECT INDX1    ASSIGN FILENAME
20                     ORGANIZATION IS INDEXED
21                     ACCESS MODE IS RANDOM RESERVE 3
22                     RECORD KEY IS SEQ_NO
23                     ALTERNATE RECORD KEY IS LAST_NAME
24                                             WITH DUPLICATES.
25 I-O-CONTROL.
26     APPLY           DEFERRED-WRITE
27                     EXTENSION 10
28                     WINDOW 7        ON INDX1.
29 DATA DIVISION.
30 FILE SECTION.
31 FD  INDX1           VALUE OF ID IS FILENAME.
32 01  IN_REC.
33     02  SEQ_NO                PIC X(7).
34     02  LAST_NAME   PIC X(15).
35     02  FIRST_NAME PIC X(10).
36     02  SOC_SEC               PIC X(9).
37     02  STREET                PIC X(18).
38     02  CITY                  PIC X(14).
39     02  STATE                 PIC XX.
40     02  ZIP_CODE    PIC X(5).
41 WORKING-STORAGE SECTION.
42 * CONSTANTS
43 01  TIME_WAIT              USAGE COMP-1     VALUE IS 10.0.
44 *VARIABLES
45 01  KEY_VALUE              PIC X(7)         JUSTIFIED RIGHT.
46 01  DELETE_FLAG            PIC X            VALUE IS 'N'.
47 01  RET_STATUS             PIC S9(9)  COMP.
48 01  PROG_STAT              PIC 9.
49     88  NO_ERROR                    VALUE 1.
50     88  SOME_ERROR                  VALUE 2.
51     88  WAIT_READ_AGAIN             VALUE 3.
52     88  END_OF_INPUT                VALUE 4.
53 01  FILENAME               PIC X(80)        VALUE 'XXXX'.
54 01  RMS$_RNF               PIC 9(9) COMP    VALUE EXTERNAL RMS$_RNF.
```

```
01  RMS$_RLK                PIC 9(9) COMP   VALUE EXTERNAL RMS$_RLK.

PROCEDURE DIVISION.
000BEGIN.
   DISPLAY 'ENTER FILENAME: ' WITH NO ADVANCING.
   ACCEPT FILENAME.
   CALL 'LIB$INIT_TIMER'.
   OPEN I-O INDX1 ALLOWING ALL.
   DISPLAY "************************************".
   DISPLAY "    Hit CNTL+Z to STOP RUN".
   DISPLAY "************************************".
   MOVE 0 to PROG_STAT.
   PERFORM 100-CHOOSE-RECORD UNTIL END_OF_INPUT.
   PERFORM 250-CLEANUP.

100-CHOOSE-RECORD.
   IF PROG_STAT NOT = 3 THEN
           DISPLAY "Enter SEQ_NO:  " WITH NO ADVANCING
           ACCEPT KEY_VALUE WITH CONVERSION AT END
                           MOVE 4 TO PROG_STAT
   END-IF.

   IF PROG_STAT LESS THAN 4 THEN
           MOVE KEY_VALUE TO SEQ_NO
           PERFORM 150-READ-BY-PRIMARY-KEY
   END-IF.

150-READ-BY-PRIMARY-KEY.
   MOVE 1 TO PROG_STAT.
   READ INDX1
           KEY IS SEQ_NO
           INVALID KEY
              MOVE 2 TO PROG_STAT
              MOVE RMS-STS OF INDX1 TO RET_STATUS

                EVALUATE RET_STATUS

                 WHEN RMS$_RNF
                     DISPLAY "NO RECORD WITH SEQ_NO = ",
                                   KEY_VALUE

                 WHEN RMS$_RLK
                      DISPLAY "RECORD CURRENTLY LOCKED - ",
                                   "WILL TRY AGAIN SHORTLY"
                      CALL 'LIB$WAIT' USING
                                   BY REFERENCE TIME_WAIT
                      MOVE 3 TO PROG_STAT

                 WHEN OTHER
                      CALL 'LIB$STOP' USING BY VALUE RMS-STS

               END-EVALUATE.
```

```
107
108     IF PROG_STAT IS EQUAL TO 1 THEN
109             DISPLAY SEQ_NO,"  ",SOC_SEC,"  ",LAST_NAME
110             PERFORM 200-DELETE-RECORD
111     END-IF.
112
113  200-DELETE-RECORD.
114     DISPLAY "Do you wish to delete this record? (Y/<CR>):  "
115             WITH NO ADVANCING.
116     ACCEPT DELETE_FLAG.
117     IF DELETE_FLAG = "Y" OR DELETE_FLAG = "y"
118             DELETE INDX1 INVALID KEY
119                          DISPLAY "Bad DELETE".
120  250-CLEANUP.
121     CLOSE INDX1.
122     CALL 'LIB$SHOW_TIMER'.
123     STOP RUN.
```

```
                                                                FOROPEN_INDX.FOR
!
!   FORTRAN program using USEROPEN function to open an indexed
!       file.  Thereafter, all access to the file is done using
!       regular FORTRAN I-O.
!
!  User is prompted for SEQ_NO of record to be
!  retrieved randomly and given option of deleting
!  any record retrieved.
!
!   Program assumes FILE SHARING and RECORD LOCKING
!
    PROGRAM FOROPEN_INDX

    IMPLICIT          NONE

    REAL              TIME_WAIT
    PARAMETER         (TIME_WAIT = 10.0)

    STRUCTURE /ACCOUNT_STRUC/
            CHARACTER*7     SEQ_NO            ! KEY 0
            CHARACTER*15    LAST_NAME         ! KEY 1
            CHARACTER*10    FIRST_NAME
            CHARACTER*9     SOC_SEC
            CHARACTER*18    STREET
            CHARACTER*14    CITY
            CHARACTER*2     STATE
            CHARACTER*5     ZIP_CODE
    END STRUCTURE

    RECORD /ACCOUNT_STRUC/  IN_REC

    INTEGER           IUNIT/1/,
    1                 RET_STATUS,
    2                 RMS_STS,
    3                 RMS_STV,
    4                 OPENFILE,
    5                 LEN_FILENAME,
    6                 KEY_IN

    CHARACTER*80      FILENAME
    CHARACTER*7       KEY_VALUE
    CHARACTER*1       DELETE_FLAG

    INTEGER           PROG_STAT/0/      !  VALUE  1 = no error
                                        !  VALUE  2 = some error
                                        !  VALUE  3 = wait-read-again

    EXTERNAL           OPENFILE

    INCLUDE           '($RMSDEF)'

    CALL LIB$INIT_TIMER()
```

```
53
54      WRITE (6,1)
55      READ (5,2)  len_filename,FILENAME
56
57      OPEN    (UNIT=IUNIT,
58      1       FILE=FILENAME(1:len_filename),
59      2       STATUS='OLD',
60      3       ORGANIZATION='INDEXED',
61      4       ACCESS='KEYED',
62      5       FORM='UNFORMATTED',
63      6       RECORDTYPE='FIXED',
64      7       SHARED,
65      7       USEROPEN=OPENFILE)
66
67      WRITE (6,6)
68      WRITE (6,5)             !  directions to stop run
69      WRITE (6,6)
70
71      WRITE (6,3)
72      READ (5,4) KEY_IN
73
74      DO WHILE (KEY_IN .GT. 0)
75  !
76  !   Convert integer KEY_IN to right justified character string KEY_VALUE
77  !     using internal read
78  !
79      IF (PROG_STAT .NE. 3)
80      1         WRITE (UNIT=KEY_VALUE,FMT='(I7)') KEY_IN
81
82           READ (IUNIT,KEY=KEY_VALUE,KEYID=0,IOSTAT=RET_STATUS) IN_REC
83
84         IF (RET_STATUS .GT. 0) THEN
85              PROG_STAT = 2
86              CALL ERRSNS(,RMS_STS,RMS_STV,IUNIT,)
87
88              IF (RMS_STS .EQ. RMS$_RNF) THEN
89                 WRITE (6,11) KEY_VALUE
90              ELSE
91
92                IF (RMS_STS .EQ. RMS$_RLK) THEN
93                  WRITE (6,12)
9<                  CALL LIB$WAIT (TIME_WAIT)
9                   PROG_STAT = 3
96                ELSE
97                 CALL LIB$STOP(%VAL(RMS_STS),%VAL(RMS_STV))
98                END IF
99              END IF
100        ELSE
101           PROG_STAT = 1
102        END IF
103
```

```
104         IF (PROG_STAT .EQ. 1) THEN
105
106                 WRITE (6,*) IN_REC.SEQ_NO,'  ',IN_REC.SOC_SEC,'  ',
107        1             IN_REC.LAST_NAME
108
109                 WRITE (6,14)
110                 READ  (5,15) DELETE_FLAG
111
112                 IF (DELETE_FLAG .EQ. 'Y' .OR.
113        1                    DELETE_FLAG .EQ. 'y') THEN
114                     DELETE (UNIT=IUNIT)
115                 END IF
116          END IF   ! prog_stat = 1
117
118         IF (PROG_STAT .NE. 3) THEN
119
120               WRITE (6,3)
121               READ (5,4) KEY_IN
122
123         END IF    ! prog_stat .ne. 3
124
125       END DO
126
127       CLOSE (IUNIT)
128
129       CALL LIB$SHOW_TIMER()
130
131       CALL EXIT
132
133    1  FORMAT ('$Enter filename:  ')
134    2  FORMAT (Q,A)
135    3  FORMAT ('$Enter SEQ_NO:  ')
136    4  FORMAT (I)
137    5  FORMAT ('    Hit <CR> or enter zero to stop run')
138    6  FORMAT ('*********************************************')
139   11  FORMAT (' Record NOT FOUND with SEQ_NO = ',A)
140   12  FORMAT (' Record currently LOCKED ',
141       1                ' - will try again shortly')
142   14  FORMAT ('$Do you wish to delete this record? (Y/<CR>):  ')
143   15  FORMAT (A)
144
145       END
146
```

```
147      INTEGER FUNCTION OPENFILE (FAB,RAB,IUNIT)
148
149      IMPLICIT        NONE
150
151      INCLUDE         '($FABDEF)'
152      INCLUDE         '($RABDEF)'
153      INCLUDE         '($RMSDEF)'
154
155      RECORD /FABDEF/         FAB
156      RECORD /RABDEF/         RAB
157
158      INTEGER                 IUNIT,
159      1                       RET_STATUS,
160      2                       SYS$OPEN,
161      3                       SYS$CONNECT
162
163   !  Extension # blocks if file is extended
164      FAB.FAB$W_DEQ = 10
165
166   !  File access desired for USER
167      FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_DEL .OR. FAB$M_GET
168      1        .OR. FAB$M_PUT .OR. FAB$M_UPD
169
170   !  File options desired
171      FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_DFW    !  deferred write
172
173   !  # global buffers if wish to use them or set to zero if wish to
174   !  override global and use local buffers if someone already has
175   !  file opened ith global buffers enabled
176   !  The following statement must be inserted in source after
177   !  SYS$OPEN call but prior to SYS$CONNECT call.
178   !  FAB.FAB$W_GBC = ?
179
180   !  Sharing attributes - what others can do or set TO FAB$M_SHRNIL
181      FAB.FAB$B_SHR = FAB.FAB$B_SHR .OR. FAB$M_SHRPUT
182      1        .OR. FAB$M_SHRGET .OR. FAB$M_SHRDEL
183      2        .OR. FAB$M_SHRUPD
184
185      RET_STATUS = SYS$OPEN(FAB,,)
186
187   !  Specify number of local buffers you want RMS to allocate on CONNECT
188      RAB.RAB$B_MBF  =  3
189
190   !  Enable any record processing options to be used for entire run
191      RAB.RAB$L_ROP  =  RAB.RAB$L_ROP .OR. RAB$M_FDL  !  fast delete
192
193      IF (RET_STATUS) RET_STATUS = SYS$CONNECT(RAB,,)
194
195      IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))
196      OPENFILE = RET_STATUS
197
198      RETURN
199      END
200
```

MAROPEN_INDX.MAR

```
1   ;
2   ;
3   ;   MACRO program using RMS services to open an INDEX file
4   ;  and terminal INPUT and OUTPUT files.
5   ;
6   ;   All I-O is done using RMS services.
7   ;
8   ;   User is prompted for SEQ_NO of a record to be
9   ;  retrieved randomly and given option of deleting
10  ;  any record retrieved.
11  ;
12  ;   Program assumes FILE SHARING and RECORD LOCKING
13  ;
14  ;
15      .TITLE MAROPEN
16
17      .MACRO          LIB_ERROR ?NO_ERROR
18      BLBS            R0,NO_ERROR
19      PUSHL           R0
20      CALLS           #1,G^LIB$STOP
21      NO_ERROR:
22      .ENDM           LIB_ERROR
23
24      .MACRO          RMS_ERROR device, prefix,?NO_ERROR
25      BLBS            R0,NO_ERROR
26      MOVAL           device, R6
27      PUSHL           prefix'$L_STV(R6)
28      PUSHL           prefix'$L_STS(R6)
29      PUSHL           R0
30      CALLS           #3,G^LIB$STOP
31      NO_ERROR:
32      .ENDM           RMS_ERROR
33
34      .PSECT  NONSHARED_DATA  NOEXE,WRT
35
36  TIME_WAIT: .F_FLOATING      10.0    ; # seconds wait if record locked
37
38  IN_REC:
39      SEQ_NO:             .BLKB   7
40      LAST_NAME:          .BLKB   15
41      FIRST_NAME:         .BLKB   10
42      SOC_SEC             .BLKB   9
43      STREET:             .BLKB   18
44      CITY:               .BLKB   14
45      STATE:              .BLKB   2
46      ZIP_CODE:           .BLKB   5
47  INREC_LENGTH = . - IN_REC
48
49  INREC_BUFF:                     .BLKB   80
50
51  KEY_VALUE:          .BLKB   7
52  KEY_LENGTH =        . - KEY_VALUE
53
```

```
 54  IN_PROMPT1:                 .ASCII /Enter filename:  /
 55  IN_PMTSIZE1 =     . - IN_PROMPT1
 56
 57  IN_PROMPT2:                 .ASCII /Enter SEQ_NO:  /
 58  IN_PMTSIZE2 =    . - IN_PROMPT2
 59
 60  IN_PROMPT3:          .ASCII /Do you wish to DELETE this record? (Y or <CR>): /
 61  IN_PMTSIZE3 =        . - IN_PROMPT3
 62
 63  DIRECTIONS:          .ASCII /  AT SEQ_NO PROMPT -  HIT <CR> TO STOP RUN/
 64  DIRECTIONS_LENGTH = . - DIRECTIONS
 65
 66  HIGHLIGHT: .ASCII /************************************************/
 67  HIGHLIGHT_LENGTH =   . - HIGHLIGHT
 68
 69  OUT_BUFF:            .BLKB    80
 70  OUTBUFF_LENGTH    =   . - OUT_BUFF
 71
 72  IN_BUFF:             .BLKB    80
 73  INBUFF_LENGTH     =   . - IN_BUFF
 74
 75  MSG_RNF:    .ASCII /   RECORD NOT FOUND/
 76  MSGRNF_LENGTH = . - MSG_RNF
 77
 78  MSG_RLK: .ASCII /RECORD CURRENTLY LOCKED - WILL TRY AGAIN SHORTLY/
 79  MSGRLK_LENGTH = . - MSG_RLK
 80
 81  .PSECT      SHARED_DATA      PIC, NOEXE, LONG
 82
 83  INDX_INFAB:
 84     $FAB              FAC=<GET,DEL>,-
 85                       FOP=<DFW>,-              ; deferred write
 86                       ORG=<IDX>,-
 87                       RAT=<CR>,-
 88                       RFM <FIX>,-
 89                       SHR=<SHRGET,SHRPUT,SHRDEL,SHRUPD>
 90  INDX_INRAB:
 91     $RAB              FAB=INDX_INFAB,-
 92                       KBF=KEY_VALUE,-
 93                       KSZ=KEY_LENGTH,-
 94                       KRF=0,-                  ; primary key of reference
 95                       MBF=3,-                  ; multibuffer count
 96                       RAC=<KEY>,-
 97                       ROP=<FDL>,-              ; fast delete
 98                       RBF=INREC_BUFF,-
 99                       RSZ=INREC_LENGTH,-
100                       UBF=INREC_BUFF,-
101                       USZ=INREC_LENGTH
102
```

```
103  TTIN_FAB:
104     $FAB            FNM=<SYS$INPUT>,-
105                     RAT=CR,-
106                     FAC=<GET>
107  TTIN_RAB:
108     $RAB            FAB=TTIN_FAB,-
109                     UBF=IN_BUFF,-
110                     USZ=80,-
111                     ROP=PMT,-
112                     PBF=IN_PROMPT1,-
113                     PSZ=IN_PMTSIZE1
114
115  TTOUT_FAB:
116     $FAB            FNM=<SYS$OUTPUT>,-
117                     RAT=CR,-
118                     FAC=<PUT>
119  TTOUT_RAB:
120     $RAB            FAB=TTOUT_FAB,-
121                     RBF=OUT_BUFF,-
122                     RSZ=80
123
124     .PSECT  CODE    SHR, NOWRT, EXE
125
126  ;*****************************************************************************
127
128     .ENTRY  MAROPEN_INDX    ^M<>
129
130     CALLS           #0, G^LIB$INIT_TIMER
131     LIB_ERROR
132
133  ; open SYS$INPUT
134
135     $OPEN           FAB=TTIN_FAB
136     RMS_ERROR       TTIN_FAB, FAB
137     $CONNECT        RAB=TTIN_RAB
138     RMS_ERROR       TTIN_RAB, RAB
139
140  ; input name of indexed file
141
142     $GET            RAB=TTIN_RAB
143     RMS_ERROR       TTIN_RAB, RAB
144
145  ; open indexed file
146
147     $FAB_STORE      FAB=INDX_INFAB,-
148                     FNA=IN_BUFF,-                   ; filename
149                     FNS=TTIN_RAB+RAB$W_RSZ          ; filename length
150
151     $OPEN           FAB=INDX_INFAB
152     RMS_ERROR       INDX_INFAB, FAB
153     $CONNECT        RAB=INDX_INRAB
154     RMS_ERROR       INDX_INRAB, RAB
155
```

```
156  ; open SYS$OUTPUT
157
158     $OPEN           FAB=TTOUT_FAB
159     RMS_ERROR       TTOUT_FAB, FAB
160     $CONNECT        RAB=TTOUT_RAB
161     RMS_ERROR       TTOUT_RAB, RAB
162
163  ; display directions to stop run and prompt for first key value
164
165     MOVC5           #HIGHLIGHT_LENGTH,HIGHLIGHT,#^A/ /,#80,OUT_BUFF
166     $PUT            RAB=TTOUT_RAB
167     RMS_ERROR       TTOUT_RAB, RAB
168
169     MOVC5           #DIRECTIONS_LENGTH,DIRECTIONS,#^A/ /,#80,OUT_BUFF
170     $PUT            RAB=TTOUT_RAB
171     RMS_ERROR       TTOUT_RAB, RAB
172
173     MOVC5           #HIGHLIGHT_LENGTH,HIGHLIGHT,#^A/ /,#80,OUT_BUFF
174     $PUT            RAB=TTOUT_RAB
175     RMS_ERROR       TTOUT_RAB, RAB
176
177  ; input key value
178
179  KEY_IN:
180     $RAB_STORE      RAB=TTIN_RAB,-
181                     PBF=IN_PROMPT2,-
182                     PSZ=#IN_PMTSIZE2
183
184     $GET            RAB=TTIN_RAB
185     RMS_ERROR       TTIN_RAB, RAB
186
187  STOP_RUN_CHK:
188     TSTW            TTIN_RAB+RAB$W_RSZ      ; <cr> so # bytes = 0
189     BNEQ            CONV_KEY                ; NO
190     JMP             D'E                     ; YES, STOP
191
192  ; right justify key value
193
194  CONV_KEY:
195     MOVC5           #0,KEY_VALUE,#^A/ /,#KEY_LENGTH,KEY_VALUE
196     CVTWL           TTIN_RAB+RAB$W_RSZ,R6
197     SUBL3           R6,#KEY_LENGTH,R6
198     ADDL            #KEY_VALUE,R6
199     MOVC3           TTIN_RAB+RAB$W_RSZ,IN_BUFF,(R6)
200
```

```
201  ; do indexed keyed retrieval of record
202
203  READ_REC:
204     $GET            RAB=INDX_INRAB
205     BLBS            R0,SUCCESSFUL_READ      ; record found
206     CMPL            R0,#RMS$_RNF            ; record NOT found
207     BEQL            RNF
208     CMPL            R0,#RMS$_RLK            ; record LOCKED
209     BEQL            RLK
210     RMS_ERROR       INDX_INRAB,RAB
211
212  SUCCESSFUL_READ:
213     JMP             TYPEOUT_REC
214
215  ; record not found - print out message and input new key value
216
217  RNF:
218     MOVC5           #MSGRNF_LENGTH,MSG_RNF,#^A/ /,#80,OUT_BUFF
219     $PUT            RAB=TTOUT_RAB
220     RMS_ERROR       TTOUT_RAB, RAB
221     JMP             KEY_IN
222
223  ; record LOCKED - print out message, wait 10 seconds and try again
224
225  RLK:
226     MOVC5           #MSGRLK_LENGTH,MSG_RLK,#^A/ /,#80,OUT_BUFF
227     $PUT            RAB=TTOUT_RAB
228     RMS_ERROR       TTOUT_RAB, RAB
229     PUSHAL          TIME_WAIT
230     CALLS           #1,G^LIB$WAIT
231     LIB_ERROR
232
233  WAIT_OVER:
234     JMP             READ_REC
235
236  ; record found - type out record on terminal
237
238  TYPEOUT_REC:
239     MOVC3           #INREC_LENGTH,INREC_BUFF,OUT_BUFF
240     $PUT            RAB=TTOUT_RAB
241     RMS_ERROR       TTOUT_RAB, RAB
242
243  ; ask user whether he or she wishes to DELETE record displayed
244
245  DELETE_FLAG:
246     $RAB_STORE      RAB=TTIN_RAB,-
247                     PBF=IN_PROMPT3,-
248                     PSZ=#IN_PMTSIZE3
249
```

```
250     $GET            RAB=TTIN_RAB
251     TSTW            TTIN_RAB+RAB$W_RSZ      ; <CR> - don't delete
252     BNEQ            CHECK_FLAG_UC           ; not equal 0 so check whether
253                                             ;   equal to 'Y' or 'y'
254     JMP             KEY_IN                  ; equal to 0 so don't delete
255                                             ;   record -  read another key
256 CHECK_FLAG_UC:
257     CMPB            IN_BUFF,#^A/Y/          ; equal to upper case 'Y'
258     BNEQ            CHECK_FLAG_LC
259     JMP             DELETE_REC
260
261 CHECK_FLAG_LC:
262     CMPB            IN_BUFF,#^A/y/          ; equal to lower case 'y'
263     BNEQ            FLAG_NE
264     JMP             DELETE_REC
265
266 FLAG_NE:
267     JMP             KEY_IN                  ; not equal to 'Y' or 'y' -
268                                             ;   read in another key value
269
270 ;  delete record
271
272 DELETE_REC:
273     $DELETE         RAB=INDX_INRAB
274     RMS_ERROR       INDX_INRAB, RAB
275
276 GET_ANOTHER_KEY:
277     JMP             KEY_IN
278
279 DONE:
280     CALLS           #0, G^LIB$SHOW_TIMER
281     LIB_ERROR
282     $EXIT_S                                 ; EXIT to VMS
283     .END    MAROPEN_INDX
```

```
                                                        PASOPEN_INDX.PAS
{
   PASCAL program using USERACTION function to open an INDEXED
   file.  Thereafter, all access to the file is done
   using regular PASCAL I/O.

   User is prompted for seq_no of record to be
   retrieved randomly and given option of
   deleting any record retrieved.

   Program assumes FILE SHARING and RECORD LOCKING

}
   [INHERIT ('SYS$LIBRARY: STARLET')]
   PROGRAM PASOPEN_INDX(INPUT,OUTPUT,INDX1);

   CONST
          TIME_WAIT = 10.0;          { Number of seconds wait
                                       if record locked }

   TYPE

     ACCOUNT_STRUC = RECORD
          SEQ_NO      : [KEY(0)] PACKED ARRAY [1..7] OF CHAR;
          LAST_NAME   : [KEY(1)] PACKED ARRAY [1..15] OF CHAR;
          FIRST_NAME  : PACKED ARRAY [1..10] OF CHAR;
          SOC_SEC     : PACKED ARRAY [1..9] OF CHAR;
          STREET      : PACKED ARRAY [1..18] OF CHAR;
          CITY        : PACKED ARRAY [1..14] OF CHAR;
          STATE       : PACKED ARRAY [1..2] OF CHAR;
          ZIP_CODE    : PACKED ARRAY [1..5] OF CHAR;
     END;

     ACCOUNT_REC    =  FILE OF ACCOUNT_STRUC;
     UNSAFE_FILE    =  [UNSAFE] FILE OF CHAR;
     RAB_PTR        =  ^RAB$TYPE;

   VAR

     INDX1          :  ACCOUNT_REC;
     IN_REC         :  ACCOUNT_STRUC;
     RMS_STS        :  UNSIGNED;
     RET_STATUS,
     LEN_FILENAME,
     KEY_IN,
     KEYLEN         :  INTEGER;
     FILENAME       :  [VOLATILE] VARYING [80] OF CHAR;
     KEY_VALUE      :  VARYING [7] OF CHAR;
     RAB_START      :  RAB_PTR;
     DELETE_FLAG    :  CHAR;

     PROG_STAT      :  INTEGER:=0;  {  VALUE  1 =  no error
                                       VALUE  2 =  some error
                                       VALUE  3 =  wait-read-again  }
```

```
53
54      FUNCTION PAS$RAB (VAR anyname : Unsafe_file):
55                              RAB_PTR; EXTERN;
56
57      PROCEDURE LIB$WAIT (
58              num_secs     : REAL); EXTERN;
59
60      PROCEDURE LIB$STOP (
61              %IMMED cond_value : INTEGER); EXTERN;
62
63      PROCEDURE LIB$INIT_TIMER (
64              VAR HANDLER_ADR : INTEGER:= %IMMED 0); EXTERN;
65
66      PROCEDURE LIB$SHOW_TIMER (
67              HANDLER_ADR : INTEGER := %IMMED 0;
68              CODE : INTEGER := %IMMED 0;
69              [IMMEDIATE,UNBOUND] PROCEDURE ACTION_RTN (OUT_STR :
70                [CLASS_S] PACKED ARRAY [L..U:INTEGER]
71                OF CHAR):= %IMMED 0;
72              %IMMED USER_ARG : INTEGER := %IMMED 0); EXTERN;
73
74      FUNCTION OPENFILE (
75                      VAR FAB : FAB$TYPE;
76                      VAR RAB : RAB$TYPE;
77                      VAR FNAME : ACCOUNT_REC): INTEGER;
78
79      BEGIN  { openfile function }
80
81  {   Extension # blocks if file is extended }
82      FAB.FAB$W_DEQ := 10;
83
84  {   File access desired for USER }
85      FAB.FAB$B_FAC := FAB$M_DEL + FAB$M_GET + FAB$M_PUT + FAB$M_UPD;
86
87  {   File options desired  }
88      FAB.FAB$L_FOP := FAB$M_DFW;      {  deferred write }
89
90  {   # global buffers if wish to use them or set to zero if wish
91      to override global and use local buffers if someone
92      already has file opened with global buffers enabled.
93      The following statement must be inserted in source after
94  {   SYS$OPEN call but prior to SYS$CONNECT call.      }
95  {   FAB.FAB$W_GBC := #;   }
96
97
98  {   Sharing attributes - what others can do or set TO FAB$M_SHRNIL }
99      FAB.FAB$B_SHR := FAB$M_SHRPUT + FAB$M_SHRGET + FAB$M_SHRDEL
100             + FAB$M_SHRUPD;
101
102     RET_STATUS := $OPEN (FAB,,);
103
104
105 {   Specify number of local buffers you want RMS to allocate on CONNECT }
106     RAB.RAB$B_MBF  :=  3;
```

```
107
108  {  Enable any record processing options to be used for entire run }
109     RAB.RAB$L_ROP  :=  RAB$M_FDL;              {  fast delete}
110
111     IF ODD(RET_STATUS) THEN
112             RET_STATUS := $CONNECT(RAB,,)
113     ELSE
114             LIB$STOP(RET_STATUS);
115
116     OPENFILE := RET_STATUS;
117
118     END;  { function openfile }
119
120
121     BEGIN  { MAIN }
122
123       LIB$INIT_TIMER;
124
125       WRITE ('Enter filename:  ');
126       READLN  (filename);
127
128       OPEN (file_variable    := INDX1,
129           file_name          := filename,
130           organization       := INDEXED,
131           access_method      := KEYED,
132           history            := OLD,
133           user_action        := OPENFILE);
134
135       RAB_START := PAS$RAB (INDX1);
136       RESETK (INDX1,0);                 {  Retrieval by primary key }
137
138       WRITELN ('***************************************');
139       WRITELN ('      Enter ZERO to stop run');
140       WRITELN ('***************************************');
141
142       WRITE ('Enter SEQ_NO:  ');
143       READLN (KEY_IN);
144
145       WHILE (KEY_IN > 0) DO
146       BEGIN  { while loop }
147
148  {    Convert integer KEY_IN to right justified character string KEY_VALUE }
149
150          IF (PROG_STAT <> 3) THEN
151              WRITEV (KEY_VALUE,KEY_IN:7);
152
153          FINDK (INDX1,0,KEY_VALUE,EQL,ERROR:=CONTINUE);
154
```

```
155  {  NOTE:  PASCAL FINDK does not return status code RMS$_RNF to RAB$L_STS
156            Can test for this condition with UFB functon                    }
157
158          IF (STATUS(INDX1) > 0) OR UFB(INDX1) THEN
159          BEGIN
160             PROG_STAT := 2;
161             RMS_STS   := RAB_START^.RAB$L_STS;
162
163             IF UFB(INDX1) THEN
164               WRITELN (' Record NOT FOUND with SEQ_NO = ',KEY_VALUE)
165
166             ELSE
167             IF (RMS_STS = RMS$_RLK) THEN
168               BEGIN
169                 WRITELN (' Record currently LOCKED ',
170                      ' - will try again shortly');
171                 LIB$WAIT (TIME_WAIT);
172                 PROG_STAT := 3;
173               END
174               ELSE
175                  LIB$STOP (%IMMED RMS_STS);
176          END
177          ELSE PROG_STAT := 1;
178
179        IF (PROG_STAT = 1) THEN
180        BEGIN
181
182            IN_REC := INDX1^;
183            WRITELN (IN_REC.SEQ_NO,'   ',IN_REC.SOC_SEC,'   ',
184                                  IN_REC.LAST_NAME);
185
186            WRITE ('Do you wish to delete this record?',
187                                  ' (Y/any char):  ');
188            READLN (DELETE_FLAG);
189
190            IF ((DELETE_FLAG = 'Y') OR (DELETE_FLAG = 'y')) THEN
191                     DELETE (INDX1);
192         END;  { if prog_stat = 1 }
193
194       IF (PROG_STAT <> 3) THEN
195       BEGIN
196
197            WRITE ('Enter SEQ_NO:  ');
198            READLN (KEY_IN);
199
200       END { if prog_stat <> 3  }
201
202     END;  { while loop }
203
204     CLOSE (INDX1);
205
206     LIB$SHOW_TIMER;
207
208     END.  { Main program }
```

Entry Points to PASCAL Utilities

This section describes the entry points to utilities in the VAX Run-Time Library that can be called as external routines by a VAX PASCAL program. These utilities access VAX PASCAL extensions that are not directly provided by the language.

1. PAS$FAB(f)

   The PAS$FAB function returns a pointer to the RMS File Access Block (FAB) of file f. After this function has been called, the FAB can be used to get information about the file and to access RMS facilities not explicitly available in the PASCAL language.

   The component type of file f can be any type; the file must be open.

   PAS$FAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_FAB  = ^FAB$TYPE;

FUNCTION PAS$FAB
   (VAR F : Unsafe_File) : Ptr_to_FAB;
   EXTERN;
```

   This declaration allows a file of any type to be used as an actual parameter to PAS$FAB. The type FAB$TYPE is defined in the VAX PASCAL environment file STARLET.PEN, which your program or module can inherit.

2. PAS$RAB(f)

   The PAS$RAB function returns a pointer to the RMS Record Access Block (RAB) of file f. After this function has been called, the RAB can be used to get information about the file and to access RMS facilities not explicitly available in the PASCAL language.

   The component type of file f can be any type; the file must be open.

PAS$RAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_RAB  = ^RAB$TYPE;

FUNCTION PAS$RAB
   (VAX F : Unsafe_File) : Ptr_to_RAB;
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$RAB. The type RAB$TYPE is defined in the VAX PASCAL environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS RAB does not interfer with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the RAB is used, which may in turn require you to change your program.

FDL$PARSE ALTERNATIVE

Call FDL$PARSE to set up and initialize RMS control blocks and thereafter direct calls to RMS services must be used.

1. Opens up FDL file (specified as arg 1 in call).

2. Allocates necessary RMS control blocks and initializes permanent file attributes set in FDL file and any run-time characteristics indicated in the FDL file.

3. Returns address of FAB (arg 2) and address of RAB (arg 3) to the calling program. Subsequently, can be used by the calling program to access and change settings in FAB and RAB.

NOTE

- FDL$PARSE may be called from all higher-level languages including COBOL (and DIBOL).
- Once FDL$PARSE is called the rest of the I/O must be done by calling RMS services directly. There is no way to connect the file channel in the FAB obtained by FDL$PARSE with the regular higher-level language I/O. Note that this channel could be used in QIOs.
- The following warning applies to programmers used to accessing the RMS control blocks using a USEROPEN routine supported by their higher-level language.

WARNING

When you call FDL$PARSE you are responsible for enabling any file or record options you wish to use. They must be either explicitly specified in the FDL or directly set in the program. The defaults usually enabled by your compiler will not be set. Thus, if your compiler had usually enabled DEFERRED WRITE, you will now have to do it.

4. To access RMS control blocks and set them, most higher-level languages have to pass the FAB or RAB pointer to a routine. In the routine the FAB or RAB RECORD is specified as the receiving parameter.

IMPORTANT

The calling routine must pass the FAB or RAB pointer BY VALUE (even though the called routine expects to receive it by reference). We are deliberately tricking the compiler.

FDL$PARSE Routine

The FLD$PARSE routine parses an FDL specification, allocates RMS control blocks (FABs, RABs, or XABs), and fills in the relevant fields.

Format

```
FDL$PARSE fld_spec, fdl_fab_pointer
          ,fdl_rab_pointer [,flags]
          [,dflt_fdl_spc] [,stmnt_num]
```

Arguments

fdl_spec

The fdl_spec argument is the name of the FDL file or the actual FDL specification to be parsed. It is the address of a character string descriptor pointing to either the name of the FDL file or the actual FDL specification to be parsed. If the FDL$V_FDL_STRING flag is set in the flags argument, FDL$PARSE interprets this argument as an FDL specification in string form. Otherwise, FDL$PARSE interprets this argument as a file name of an FDL file.

fdl_fab_pointer

Address of an RMS file access block (FAB). The fdl_fab_pointer argument is the address of a longword which receives the address of an RMS file access block (FAB). FDL$PARSE both allocates the FAB and fills in its relevant fields.

fdl_rab_pointer

Address of an RMS record access block (RAB). The fdl_rab_pointer argument is the address of a longword which receives the address of an RMS record access block (RAB). FDL$PARSE both allocates the RAB and fills in its relevant fields.

flags

The flags (or masks) argument controls how the dflt_fdl_spc argument is interpreted and how errors are signaled. It is the address of a longword containing the control flags. If this argument is omitted or is specified as zero, no flags are set. The flags and their meanings are described below.

Flag	Description
FDL$V_DEFAULT_STRING	Interprets the dflt_fdl_spc argument as an FDL specification in string form. By default, the dflt_fdl_spc argument is interpreted as a file name of an FDL file.
FDL$V_FDL_STRING	Interprets the fdl_spec argument as an FDL specification in string form. By default, the fdl_spec argument is interpreted as a file name of an FDL file.
FDL$V_SIGNAL	Signals any error. By default, the status code is returned to the calling image.

This argument is optional. By default, an error status is returned rather than signaled.

dflt_fdl_spc

The dflt_fdl_spc argument is the name of the default FDL file or specification. It is the address of a character string descriptor pointing to either the default FDL file or the default FDL specification. If the FDL$V_DEFAULT_STRING flag is set in the flags argument, FDL$PARSE interprets this argument as an FDL specification in string form. Otherwise, FDL$PARSE interprets this argument as a file name of an FDL file.

This argument allows you to specify default FDL attributes. In other words, FDL$PARSE processes the attributes specified in this argument, unless you override them with the attributes you specify in the fdl_spec argument.

The FDL defaults can be coded directly into your program, typically with an FDL specification in string form.

This argument is optional.

stmnt_num

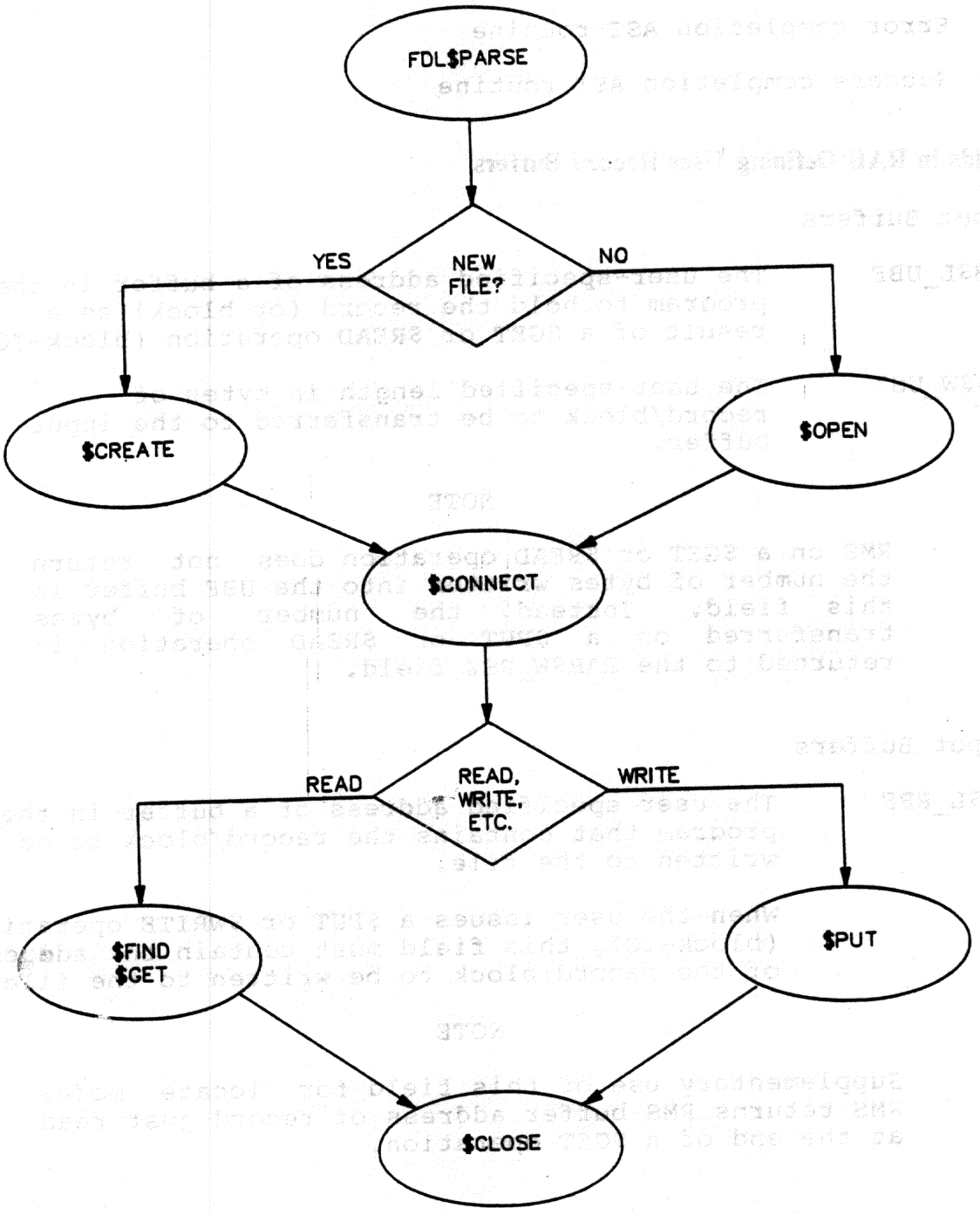
The FDL statement number. The stmnt_num argument is the address of a longword that receives the FDL statement number. If the routine completes successfully, the stmnt_num argument is the number of statements in the FDL specification. If the routine does not complete successfully, the stmnt_num argument receives the number of the statement that caused the error. In general, however, line numbers and statement numbers are not the same.

This argument is optional.

Condition Values Returned

SS$_NORMAL	Normal successful completion.
LIB$_BADBLOADR	Bad block address.
LIB$_BADBLOSIZ	Bad block size.
LIB$_INSVIRMEM	Insufficient virtual memory.
RMS$_DNF	Directory not found.
RMS$_DNR	Device not ready or mounted.
RMS$_WCC	Invalid wildcard context (WCC) value.

Typical Sequence of RMS Calls Initiated by FDL$PARSE



BU-2486

RMS services typically have three arguments:

1. RMS control block address (FAB or RAB)
2. Error completion AST routine
3. Success completion AST routine

Fields in RAB Defining User Record Buffers

Input Buffers

RAB$L_UBF — The user-specified address of a buffer in the program to hold the record (or block) as a result of a $GET or $READ operation (block-IO).

RAB$W_USZ — The user-specified length in bytes of record/block to be transferred to the input buffer.

NOTE

RMS on a $GET or $READ operation does not return the number of bytes written into the UBF buffer in this field. Instead, the number of bytes transferred on a $PUT or $READ operation is returned to the RAB$W_RSZ field.

Output Buffers

RAB$L_RBF — The user-specified address of a buffer in the program that contains the record/block to be written to the file.

When the user issues a $PUT or $WRITE operation (block-IO), this field must contain the address of the record/block to be written to the file.

NOTE

Supplementary use of this field for locate mode: RMS returns RMS buffer address of record just read at the end of a $GET operation.

RAB$W_RSZ The user-specified length, in bytes, of the output buffer.

NOTE

Supplementary use of this field for variable-length fields. RMS returns the length of the record/block transferred by a $GET or $READ operation. In the case of VFC records, RMS returns the length of the variable portion.

Specification of Internal Address

BASIC

```
DECLARE LONG X-ptr
  .
  .
  .
X-ptr = LOC (     )
```

COBOL

```
01 x-ptr     USAGE IS POINTER
                 VALUE REFERENCE _____________.
                                 variable-name
```

FORTRAN

```
INTEGER*4     x-ptr
  .
  .
  .
X-ptr = %LOC (      )
```

PASCAL

Alternative 1 (New Pascal V3.0)

```
VAX
x-ptr : INTEGER;
  .
  .
  .
x-ptr : = IADDRESS (      );
```

Alternative 2

```
TYPE          {real    }
  ptr-type = {integer;}

VAR
 x-ptr : ^ptr-type;
  .
  .
  .
x-ptr : = ADDRESS (      );
```

MACRO

```
label:
  .
  .
  .
MOVAL     label
```

VAX Language I/O Operations and RMS Services

This section consists of tables showing how each language's statements relate to RMS services.

BASIC I/O Statements and RMS Routines

BASIC Statement	RMS Routines
GET	$GET
PUT	$PUT
FIND	$FIND
DELETE	$DELETE
UPDATE	$UPDATE
RESTORE	$REWIND
SCRATCH	$TRUNCATE
FREE	$FREE
UNLOCK	$RELEASE

NOTE

1. The first PRINT or INPUT statement to channel 0 causes an $OPEN and $CONNECT operation to SYS$INPUT and SYS$OUTPUT.

2. If a $DELETE, $FIND, $FREE, $GET, $PUT, or $UPDATE operation fails because the record stream is active, the function is retried after a $WAIT.

COBOL I/O Statements and RMS Routines

COBOL Statement	RMS Routine
ACCEPT	$GET
CLOSE	$CLOSE, $DISCONNECT, $NXTVOL
DELETE	$FIND, $DELETE (See Note)
OPEN	$OPEN or $CREATE, $CONNECT
READ	$GET
REWRITE	$FIND, $UPDATE (See Note)
WRITE	$PUT
DISPLAY	$PUT
START	$FIND
UNLOCK	$RELEASE, $FREE

NOTE

$FIND is done only when the DELETE or REWRITE is being performed during random access.

FORTRAN I/O Statements and RMS Routines

FORTRAN Statement	RMS Routine	See Note Number:
ACCEPT	$GET	5
BACKSPACE	$REWIND, followed by one or more $GET operations if target record is not the first record	
CLOSE	$CLOSE	
DEFINE FILE	None	
DELETE(u)	$DELETE	5
DELETE(u, REC = r)	$FIND, $DELETE	5
ENDFILE	$PUT	2,5
FIND	$FIND	1,5
INQUIRE (by file)	$PARSE, $SEARCH, $OPEN, $CLOSE	
INQUIRE (by unit)	None	
OPEN	$OPEN or $CREATE, $CONNECT	
OPEN (with USEROPEN)	None	
OPEN (on connected unit)	$PARSE, $SEARCH, $CLOSE, $OPEN or $CREATE, $CONNECT	8
PRINT	$PUT	4,5
READ	$GET	1,5,7
READ (internal file)	None	
REWIND	$REWIND	
REWRITE	$UPDATE	5
TYPE	$PUT	4,5
UNLOCK	$FREE	5
WRITE	$PUT	2,5,7
WRITE (internal file)	None	

NOTES

1. If the unit is not already open, the first READ or FIND statement on a logical unit invokes an $OPEN and a $CONNECT.

2. If the unit is not already open, the first WRITE or ENDFILE statement on a logical unit causes a $CREATE and $CONNECT.

3. The first ACCEPT statement in a program causes an $OPEN and $CONNECT.

4. The first PRINT statement in a program and the first TYPE statement in a program each cause a $CREATE and $CONNECT.

5. If a $DELETE, $FIND, $FREE, $GET, $PUT, or $UPDATE fails because the record stream is active, the function is retried after a $WAIT.

6. If RECORDTYPE is explicitly or implicitly 'SEGMENTED', an unformatted sequential READ statement can cause more than one $GET.

7. If RECORDTYPE is explicitly or implicitly 'SEGMENTED', an unformatted sequential WRITE statement can cause more than one $PUT.

8. If the specified file name is the same as the name of the currently open file, only the BLANK= parameter is changed; otherwise, the current file is closed and the new file opened.

PASCAL I/O Statements and RMS Routines

PASCAL Statement	RMS Routine	See Note Number:
CLOSE	$CLOSE	1,3
DELETE	$DELETE	1
EOF	None	1,2
EOLN	None	1,2
FIND	$GET	1
FINDK	$GET	1
GET	$GET	1,2
LINELIMIT	None	1,2

LOCATE	None	1
OPEN	$OPEN or $CREATE, $CONNECT	4
PAGE	$PUT	1,2
PUT	$PUT	1,2
READ	$GET	1,2
READLN	$GET	1,2
RESET	$REWIND, $GET	4,5
RESETK	$REWIND, $GET	1
REWRITE	$REWIND, $GET, $TRUNCATE	4,5
STATUS	None	1,2
TRUNCATE	$TRUNCATE	1,2
UFB	None	1,2
UNLOCK	$RELEASE	1
UPDATE	$UPDATE	1
WRITE	$PUT	1,2
WRITELN	$PUT	1,2

NOTES

1. May implicitly open file INPUT or OUTPUT. See OPEN for RMS operations.

2. If delayed device access (lazy lookahead) is in progress on the file, a $GET will be done, and all open text files for which prompting is enabled may have a $PUT performed for them.

3. If the file is in Generation mode, a $PUT may also be performed.

4. If the file is a text file opened on a terminal, and if the carriage control is LIST and no USER-ACTION procedure was specified, the file is closed and reopened with two-byte VFC record type and PRN carriage control to allow prompting.

5. Opens file if it is not already open. See OPEN for RMS operations.

Summary of FAB Fields and Options

Field	Name
FAB$L_ALQ	Allocation quantity
FAB$B_BKS	Bucket size
FAB$W_BLS	Block size for tape
FAB$L_CTX	Context
FAB$W_DEQ	Default file extension quantity
FAB$L_DEV	Device characteristics
FAB$L_DNA	Default file specification string address
FAB$L_DNS	Default file specification string size
FAB$B_FAC	File access
FAB$L_FNA	File specification string address
FAB$L_FNS	File specification size
FAB$L_FOP	File processing options
FAB$V_CBT	Contiguous best try
FAB$V_CIF	Create if nonexistent
FAB$V_CTG	Contiguous allocation
FAB$V_DFW	Deferred write
FAB$V_DLT	Delete on close service
FAB$V_MXV	Maximize version number
FAB$V_NAM	Name block inputs
FAB$V_NEF	Not position at end of file (tape)
FAB$V_NFS	Not file structured
FAB$V_OFP	Output file parse
FAB$V_POS	Current position (after closed file) (tape)
FAB$V_RCK	Read check
FAB$V_RWC	Rewind on close service (tape)
FAB$V_RWO	Rewind on open service (tape)

Field	Name
FAB$L_FOP (Cont.)	
FAB$V_SCF	Submit command
FAB$V_SPL	Spool to printer
FAB$V_SQO	Sequential only
FAB$V_SUP	Supersede
FAB$V_TEF	Truncate at end-of-file
FAB$V_TMD	Temporary, marked for delete
FAB$V_TMP	Temporary (file with no directory entry)
FAB$V_UFO	User file open or create file only
FAB$V_WCK	Write check
FAB$B_FSZ	Fixed control area size
FAB$W_GBC	Global buffer count
FAB$W_IFI	Internal file identifier
FAB$L_MRN	Maximum record number
FAB$W_MRS	Maximum record size
FAB$L_NAM	Name block address
FAB$B_ORG	File organization
FAB$B_RAT	Record attributes
FAB$V_FTN	FORTRAN carriage control
FAB$V_CR	Print LF and CR
FAB$V_BLK	Do not cross block boundaries
FAB$B_RFM	Record format
FAB$B_RTV	Retrieval window size
FAB$L_SDC	Spooling device characteristics
FAB$B_SHR	File sharing
FAB$L_STS	Completion status code
FAB$L_STV	Status value (if any)
FAB$L_XAB	Extended attribute block address

Summary of RAB Fields and Options

Field	Name
RAB$L_BKT	Bucket code
RAB$L_CTX	Context
RAB$L_FAB	FAB address
RAB$W_ISI	Internal stream ID
RAB$L_KBF	Key buffer address
RAB$B_KRF	Key of reference
RAB$B_KSZ	Key size
RAB$B_MBC	Multiblock count
RAB$B_MBF	Multibuffer count
RAB$L_PBF	Prompt buffer address
RAB$B_PSZ	Prompt buffer size
RAB$B_RAC	Record access mode
RAB$C_SEQ	
RAB$C_KEY	
RAB$C_RFA	
RAB$L_RBF	Record address
RAB$W_RFA	Record's file address
RAB$L_RFA0	VBN #
RAB$W_RFA4	Byte offset or ID #
RAB$L_RHB	Record header buffer
RAB$L_ROP	Record processing options
RAB$V_ASY	Asynchronous
RAB$V_BIO	Block I/O
RAB$V_CCO	Cancel CTRL/O
RAB$V_CVT	Convert to uppercase
RAB$V_EOF	End-of-file
RAB$V_ETO	Extended terminal operation
RAB$V_FDL	Fast delete

Field	Name
RAB$L_ROP (Cont.)	
RAB$V_KGE	Key greater than or equal to
RAB$V_KGT	Key greater than
RAB$V_LIM	Limit
RAB$V_LOA	Load according to fill factor
RAB$V_LOC	Locate mode
RAB$V_NLK	No lock
RAB$V_NXR	Nonexistent record
RAB$V_PMT	Prompt
RAB$V_PTA	Purge type-ahead
RAB$V_RAH	Read-ahead
RAB$V_REA	Lock record for read (others can read)
RAB$V_RLK	Lock record for write (others can read)
RAB$V_RNE	Read no echo
RAB$V_RRL	Read regardless of lock
RAB$V_RNF	Read no filter
RAB$V_TMO	Timeout
RAB$V_TPT	Truncate put
RAB$V_UIF	Update if
RAB$V_ULK	Manual unlocking
RAB$V_WAT	Wait if record currently locked
RAB$V_WBH	Write-behind
RAB$W_RSZ	Record size
RAB$L_STS	Completion status code
RAB$L_STV	Status value (if any)
RAB$B_TMO	Timeout period
RAB$L_UBF	User record area address
RAB$W_USZ	User record area size
RAB$L_XAB	Next XAB address

Current Record Context

For each RAB connected to a FAB, RMS maintains current context information, identifying where each RAB is positioned at any given moment. RMS modifies the current context as your program performs record operations.

At any point, the current record context is represented by, at most, two records: the current record or the next record.

The context of these two records is internal to RMS; you have no direct contact with them. However, you should know what the context is to access the desired record using the appropriate RMS record service.

Record Access Stream Context

Record Operation	Record Access Mode	Current Record	Next Record
Connect	Does not apply	None	First record
Connect with RAB$L_ROP RAB$V_EOF bit set	Does not apply	None	End of file
Get, when last service was not a Find	Sequential	Old next record	New current record+1
Get, when last service was a Find	Sequential	Unchanged	Current record+1
Get	Random	New	New current record+1
Put, sequential file	Sequential	None	End of file
Put, relative file	Sequential	None	Next record position
Put, indexed file	Sequential	None	Undefined
Put	Random	None	Unchanged
Find	Sequential	Old next record	New current record+1
Find	Random	New	Unchanged
Update	Does not apply	None	Unchanged
Delete	Does not apply	None	Unchanged

Record Access Stream Context (Cont.)

Record Operation	Record Access Mode	Current Record	Next Record
Truncate	Does not apply	None	End of file
Rewind	Does not apply	None	First record
Free	Does not apply	None	Unchanged
Release	Does not apply	None	Unchanged

NOTES

1. Except for the Truncate service, RMS establishes the current record before establishing the identity of the next record.

2. The notation +1 indicates the next sequential record as determined by the file organization. For indexed files, the current key of reference is part of this determination.

3. The Connect service on an indexed file establishes the next record to be the first record in the index represented by the RAB key of reference (RAB$B__KRF) field.

4. The Connect service leaves the next record as the end of file for a magnetic tape file opened for Put services (unless the FAB$V__NEF option in the FAB$L__FOP is set).

Example 2. FDL$PARSE Alternative

BASIC (Sheet 1 of 7)

```
10 OPTION  TYPE = EXPLICIT                                    BASPARSE_INDX.BAS
   !
   !
   !   NOTE:  $LINK BASPARSE_INDX,BAS_SETRMS
   !
   !          BAS_SETRMS.BAS contains two external routines:
   !
   !                   SETFAB
   !                   SETRAB
   !
   !   This BASIC program uses FDL$PARSE to set up RMS structures
   !   for accessing an indexed file rather than using a USEROPEN
   !   function.  In addition, all access to the INDEXED file is done
   !   using RMS services rather than regular BASIC I/O.
   !   User is prompted for key value for random retrieval of
   !   records and given option to delete any record retrieved.
   !
   !   Program assumes FILE SHARING and RECORD LOCKING
   !
   ON ERROR GO TO err_check

   DECLARE SINGLE  CONSTANT  TIME_WAIT=10.0
   DECLARE STRING  CONSTANT  RIGHT_JUSTIFY = "#######"
   DECLARE WORD    CONSTANT  INREC_LENGTH=80
   DECLARE WORD    CONSTANT  KEY_VALUE_LENGTH=7

   RECORD ACCOUNT$TYPE
           STRING  SEQ_NO=7
           STRING  LAST_NAME=15
           STRING  FIRST_NAME=10
           STRING  SOC_SEC=9
           STRING  STREET=18
           STRING  CITY=14
           STRING  STATE=2
           STRING  ZIP_CODE=5
   END RECORD ACCOUNT$TYPE

   MAP (IN_REC)     ACCOUNT$TYPE    INREC

   MAP (FILE_NAME) STRING  FDL_FILE=80
   MAP (KEYVALUE)  STRING  KEY_VALUE=7
   MAP (DELFLAG)   STRING  DELETE_FLAG=1

   DECLARE LONG    FAB_PTR, &
                   RAB_PTR, &
                   RET_STATUS, &
                   INREC_PTR, &
                   KEY_IN, &
                   KEY_VALUE_PTR

   DECLARE LONG    PROG_STAT     ! VALUE  1 = no error in GET
                                 ! VALUE  2 = some error
                                 ! VALUE  3 = wait-read-again
```

```
55      EXTERNAL SUB              LIB$INIT_TIMER, &
56                                LIB$SHOW_TIMER, &
57                                LIB$WAIT
58
59      EXTERNAL LONG FUNCTION    FDL$PARSE, &
60                                SYS$OPEN, &
61                                SYS$CONNECT, &
62                                SYS$GET, &
63                                SYS$DELETE, &
64                                SYS$CLOSE
65
66      EXTERNAL LONG CONSTANT    RMS$_RNF, &
67                                RMS$_RLK
68
69    start:
70    !
71      CALL LIB$INIT_TIMER()
72    !
73      INPUT 'Enter name of FDL file ';FDL_FILE
74    !
75      RET_STATUS = FDL$PARSE(FDL_FILE,&
76                       FAB_PTR,          !  PTR to FAB returned by FDL  &
77                       RAB_PTR,,,)       !  PTR to RAB returned by FDL
78      CALL LIB$STOP(RET_STATUS BY VALUE) IF (RET_STATUS AND 1%)=0%
79
80      CALL SETFAB(FAB_PTR BY VALUE)
81
82      RET_STATUS = SYS$OPEN(FAB_PTR BY VALUE,,)
83      CALL LIB$STOP(RET_STATUS BY VALUE) IF (RET_STATUS AND 1%)=0%
84
85      INREC_PTR       = LOC(INREC)      ! internal ADDR input rec buff
86      KEY_VALUE_PTR   = LOC(KEY_VALUE)  ! internal ADDR key value
87
88      CALL SETRAB( &
89                       RAB_PTR BY VALUE,    ! PTR to RAB                &
90                       INREC_PTR,           ! internal ADDR rec buff    &
91                       INREC_LENGTH,        ! length record buffer      &
92                       KEY_VALUE_PTR,       ! internal ADDR key value   &
93                       KEY_VALUE_LENGTH)    ! length key value
94
95      RET_STATUS = SYS$CONNECT(RAB_PTR BY VALUE,,)
96      CALL LIB$STOP(RET_STATUS BY VALUE) IF (RET_STATUS AND 1%)=0%
97
98      PRINT "*********************************************"
99      PRINT "    Hit <CR> or enter zero to stop run"
100     PRINT "*********************************************"
101
102     INPUT 'Enter SEQ_NO';KEY_IN
103
104     PROG_STAT = 0
105
```

```
106     WHILE (KEY_IN > 0)
107
108       IF (PROG_STAT <> 3) THEN
109             KEY_VALUE = FORMAT$(KEY_IN,RIGHT_JUSTIFY)    ! Convert key-in
110                       ! to right justified character string key_value
111       END IF
112
113       RET_STATUS = SYS$GET(RAB_PTR BY VALUE,,)
114
115       IF (RET_STATUS AND 1%) = 0% THEN
116          PROG_STAT = 2
117          SELECT RET_STATUS
118
119          CASE RMS$_RNF
120               PRINT "Record not found with SEQ_NO = ";KEY_VALUE
121
122          CASE  RMS$_RLK
123               PRINT "Record currently locked - will try again"
124               CALL LIB$WAIT(TIME_WAIT)
125               PROG_STAT = 3
126          CASE ELSE
127               CALL LIB$STOP(RET_STATUS BY VALUE)
128          END SELECT
129       ELSE
130          PROG_STAT = 1
131       END IF
132
133       IF (PROG_STAT = 1) THEN
134               PRINT INREC::SEQ_NO;TAB(9);INREC::SOC_SEC;TAB(20);&
135                     INREC::LAST_NAME
136
137               INPUT 'Do you wish to delete this record? (Y/<CR>)';&
138                                                    DELETE_FLAG
139
140               IF (DELETE_FLAG = 'Y' OR DELETE_FLAG = 'y') THEN
141                     RET_STATUS = SYS$DELETE(RAB_PTR BY VALUE,,)
142                     CALL LIB$STOP(RET_STATUS BY VALUE) &
143                             IF (RET_STATUS AND 1%)=0%
144               END IF
145       END IF
146
147         INPUT 'Enter SEQ_NO';KEY_IN  IF PROG_STAT <> 3
148
149     NEXT
150   cleanup:
151     CALL SYS$CLOSE(FAB_PTR BY VALUE,,)
152     CALL LIB$SHOW_TIMER()
153     GOTO done
154   err_check:
155
156     ON ERROR GO TO 0
157
158   done:     END
```

```
200         SUB SETFAB (FAB$TYPE FAB)
    !                                                       BAS_SETRMS.BAS
    !
    !   EXTERNAL BASIC routines called by COBPARSE_INDX.COB or
    !                                     by BASPARSE_INDX.BAS
    !
    OPTION  TYPE =  EXPLICIT

    %INCLUDE        'FABRABDEF.BAS'

  !  Extension # blocks if file is extended
    FAB::FAB$W_DEQ = 10

  !  File access desired for USER
    FAB::FAB$B_FAC = (FAB::FAB$B_FAC OR FAB$M_DEL OR FAB$M_GET &
            OR FAB$M_PUT OR FAB$M_UPD)

  !  File options desired
    FAB::FAB$L_FOP = (FAB::FAB$L_FOP OR FAB$M_DFW)  !  deferred write

  !  # global buffers if wish to use them or set to zero if wish to
  !       override global and use local buffers if someone already
  !       has file opened with global buffers enabled
  !  The following statement must be moved to SETRAB routine
  !  prior to SYS$CONNECT call.
  !
  !       FAB::FAB$W_GBC = ?

  !  Sharing attributes - what others can do or set TO FAB$M_SHRNIL
    FAB::FAB$B_SHR = (FAB::FAB$B_SHR OR FAB$M_SHRPUT &
            OR FAB$M_SHRGET OR FAB$M_SHRDEL         &
            OR FAB$M_SHRUPD)

    END SUB


400         SUB SETRAB      (RAB$TYPE RAB,&
                    LONG REC_PTR, &
                    WORD REC_LENGTH,&
                    LONG KEY_PTR,&
                    WORD KEY_LENGTH)

    OPTION  TYPE =  EXPLICIT

    %INCLUDE        'FABRABDEF.BAS'

  !
  !  Provide address of key_value if keyed retrieval to be used
  ! and size of key
    RAB::RAB$L_KBF  =  KEY_PTR
    RAB::RAB$B_KSZ  =  KEY_LENGTH
```

```
52
53      !  Specify key of reference
54        RAB::RAB$B_KRF  =  0              ! primary key
55
56      !  Specify number of local buffers you want RMS to allocate on CONNECT
57        RAB::RAB$B_MBF  =  3
58
59      !  Specify type of mode to use for record access
60        RAB::RAB$B_RAC  =  RAB$C_KEY     ! RAB$C_SEQ or RAB$C_RFA
61
62      !
63      !  Provide address and length of output record buffer
64        RAB::RAB$L_RBF  =  REC_PTR
65        RAB::RAB$W_RSZ  =  REC_LENGTH
66      !
67      !  Provide address and length of input record buffer
68        RAB::RAB$L_UBF  =  REC_PTR
69        RAB::RAB$W_USZ  =  REC_LENGTH
70
71      !  Enable any record processing options to be used for entire run
72        RAB::RAB$L_ROP  =  (RAB::RAB$L_ROP OR RAB$M_FDL)  !  fast delete
73
74        END SUB
```

```
1       !+
2       !       FABRABDEF.BAS
3       !
4       !       RMS Data Structures Definitions
5       !+
6
7        RECORD fab$TYPE
8
9       BYTE fab$b_bid
10      BYTE fab$b_bln
11      WORD fab$w_ifi
12      LONG fab$l_fop
13      LONG fab$l_sts
14      LONG fab$l_stv
15      LONG fab$l_alq
16      WORD fab$w_deq
17      BYTE fab$b_fac
18      BYTE fab$b_shr
19      LONG fab$l_ctx
20      BYTE fab$b_rtv
21      BYTE fab$b_org
22      BYTE fab$b_rat
23      BYTE fab$b_rfm
24      LONG fab$l_jnl
25      LONG fab$l_xab
26      LONG fab$l_nam
27      LONG fab$l_fna
28      LONG fab$l_dna
29      BYTE fab$b_fns
30      BYTE fab$b_dns
31      WORD fab$w_mrs
32      LONG fab$l_mrn
33      WORD fab$w_bls
34      BYTE fab$b_bks
35      BYTE fab$b_fsz
36      LONG fab$l_dev
37      LONG fab$l_sdc
38      WORD fab$w_gbc
39      BYTE fab$b_acm
40      BYTE fab$b_rcf
41      BYTE fill(4)
42
43       END RECORD
44
45
46       RECORD rab$TYPE
47
48      BYTE rab$b_bid
49      BYTE rab$b_bln
50      WORD rab$b_isi
51      LONG rab$l_rop
```

```
52      LONG rab$l_sts
53      LONG rab$l_stv
54      RFA  rab$w_rfa              ! 6 bytes
55      WORD fill
56      LONG rab$l_ctx
57      WORD fill
58      BYTE rab$b_rac
59      BYTE rab$b_tmo
60      WORD rab$w_usz
61      WORD rab$w_rsz
62      LONG rab$l_ubf
63      LONG rab$l_rbf
64      LONG rab$l_rhb
65      VARIANT
66      CASE
67          LONG rab$l_kbf
68          BYTE rab$b_ksz
69      CASE
70          LONG rab$l_pbf
71          BYTE rab$b_psz
72      END VARIANT
73      BYTE rab$b_krf
74      BYTE rab$b_mbf
75      BYTE rab$b_mbc
76      VARIANT
77      CASE
78          LONG rab$l_bkt
79      CASE
80          LONG rab$l_dct
81      END VARIANT
82      LONG rab$l_fab
83      LONG rab$l_xab
84      END RECORD
85
86    ! +
87    ! declarations of FAB and RAB CONSTANTS
88    ! +
89      EXTERNAL BYTE CONSTANT  FAB$M_DEL,&
90                              FAB$M_GET,&
91                              FAB$M_PUT,&
92                              FAB$M_UPD,&
93                              FAB$M_DFW,&
94                              FAB$M_SHRPUT,&
95                              FAB$M_SHRGET,&
96                              FAB$M_SHRDEL,&
97                              FAB$M_SHRUPD,&
98                              FAB$M_MSE
99
100     EXTERNAL BYTE CONSTANT  RAB$C_KEY,&
101                             RAB$C_SEQ,&
102                             RAB$C_RFA,&
103                             RAB$M_FDL
```

```
1  *                                                COBPARSE_INDX.COB
2  *
3  *  COBOL PROGRAM which sets up RMS structures using FDL$PARSE
4  *     and calls TWO external non-COBOL routines to SET values in
5  *     FAB & RAB. The program retrieves records randomly by key value
6  *     entered by USER and gives user option to DELETE record.
7  *
8  *  This version is appropriate for linking in the following SETRMS
9  *     external routines:
10 *
11 *        $ LINK COBPARSE_INDX,BAS_SETRMS
12 *        $ LINK COBPARSE_INDX,FOR_SETRMS
13 *        $ LINK COBPARSE_INDX,PAS_SETRMS
14 *
15 *    Program assumes FILE SHARING and RECORD LOCKING
16 *
17 IDENTIFICATION DIVISION.
18 *
19 PROGRAM-ID. COBPARSE_INDX.
20 *
21 *
22 ENVIRONMENT DIVISION.
23
24 DATA DIVISION.
25
26 WORKING-STORAGE SECTION.
27 01   IN_REC.
28      02  SEQ_NO              PIC X(7).
29      02  LAST_NAME  PIC X(15).
30      02  FIRST_NAME PIC X(10).
31      02  SOC_SEC             PIC X(9).
32      02  STREET              PIC X(18).
33      02  CITY                PIC X(14).
34      02  STATE               PIC XX.
35      02  ZIP_CODE    PIC X(5).
36 * CONSTANTS
37 01   INREC_LENGTH    PIC 9(5) COMP   VALUE IS 80.
38 01   KEY_VALUE_LENGTH        PIC 9(5) COMP   VALUE IS 7.
39 01   TIME_WAIT               USAGE COMP-1    VALUE IS 10.0.
40 * VARIABLES
41 01   FAB_PTR                 USAGE IS POINTER.
42 01   RAB_PTR                 USAGE IS POINTER.
43 01   INREC_PTR               POINTER         VALUE REFERENCE IN_REC.
44 01   KEY_VALUE               PIC X(7)        JUSTIFIED RIGHT.
45 01   KEY_VALUE_PTR  POINTER          VALUE REFERENCE KEY_VALUE.
46 01   FDL_FILE                PIC  X(80).
47 01   RET_STATUS              PIC S9(9) COMP.
48 01   DELETE_FLAG             PIC X           VALUE IS "N".
49
50 01   PROG_STAT               PIC 9.
51      88  NO_ERROR                    VALUE 1.
52      88  SOME_ERROR                  VALUE 2.
```

```
53          88  WAIT_READ_AGAIN                         VALUE 3.
54          88  END_OF_INPUT                            VALUE 4.
55
56      01  RMS$_RNF                  PIC 9(9) COMP  VALUE EXTERNAL RMS$_RNF.
57      01  RMS$_RLK                  PIC 9(9) COMP  VALUE EXTERNAL RMS$_RLK.
58
59      PROCEDURE DIVISION.
60      000-BEGIN.
61
62          CALL 'LIB$INIT_TIMER'.
63
64          DISPLAY 'Enter FDL file name:  ' WITH NO ADVANCING.
65          ACCEPT FDL_FILE.
66      *
67      *    Call FDL$PARSE to set up RMS structures (FAB, RAB)
68          CALL 'FDL$PARSE' USING BY DESCRIPTOR FDL_FILE
69                                 BY REFERENCE FAB_PTR RAB_PTR
70                                 BY VALUE 0 0 0
71                           GIVING RET_STATUS.
72          IF RET_STATUS IS FAILURE CALL 'LIB$STOP'.
73      *
74      *   call external routine SETFAB in a non_COBOL language
75      *   that supports FAB structure to set any fields
76      *           needed which were not set in FDL file
77          CALL 'SETFAB' USING BY VALUE FAB_PTR.
78      *
79      *   Open input data file
80          CALL 'SYS$OPEN' USING BY VALUE FAB_PTR 0 0
81                          GIVING RET_STATUS.
82          IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
83                          USING BY VALUE RET_STATUS.
84      *
85      *   call external routine SETRAB WRITTEN IN BASIC,
86      *   FORTRAN or PASCAL or some higher-level language
87      *   that supports RAB structure and at a minimum
88      *           initialize addresses of record and keyvalue buffers
89      *   and their sizes
90      *
91      *   NOTE:  If the external routine were written in MACRO
92      *   all the PTR arguments HAVE TO BE PASSED BY VALUE
93      *   and only the lengths BY REFERENCE.
94      *
95      *
96          CALL 'SETRAB' USING  BY VALUE RAB_PTR
97                               BY REFERENCE  INREC_PTR
98                               BY REFERENCE  INREC_LENGTH
99                               BY REFERENCE  KEY_VALUE_PTR
100                              BY REFERENCE  KEY_VALUE_LENGTH.
101     *
```

```
102  *  Connect record
103     CALL 'SYS$CONNECT' USING BY VALUE RAB_PTR 0 0
104                        GIVING RET_STATUS.
105     IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
106                      USING BY VALUE RET_STATUS.
107
108     DISPLAY "************************************".
109     DISPLAY "     HIT CNTL+Z TO STOP RUN"
110     DISPLAY "************************************".
111
112     MOVE 0 TO PROG_STAT.
113     PERFORM 100-CHOOSE-RECORD UNTIL END_OF_INPUT.
114     PERFORM 250-CLEANUP.
115
116  100-CHOOSE-RECORD.
117  *
118  *  Prompt user for key value
119     IF PROG_STAT NOT EQUAL TO 3 THEN
120             DISPLAY "Enter SEQ_NO:  " WITH NO ADVANCING
121             ACCEPT KEY_VALUE WITH CONVERSION AT END
122                     MOVE 4 TO PROG_STAT
123     END-IF.
124
125     IF PROG_STAT NOT EQUAL TO 4 THEN
126             PERFORM 150-READ-BY-PRIMARY-KEY.
127
128
129  150-READ-BY-PRIMARY-KEY.
130
131     CALL 'SYS$GET' USING BY VALUE RAB_PTR 0 0
132                    GIVING RET_STATUS.
133
134     IF RET_STATUS IS FAILURE THEN
135        MOVE 2 TO PROG_STAT
136        EVALUATE RET_STATUS
137
138          WHEN RMS$_RNF
139               DISPLAY "NO RECORD WITH SEQ_NO = ",KEY_VALUE
140
141          WHEN RMS$_RLK
142               DISPLAY "RECORD CURRENTLY LOCKED - ",
143                     "WILL TRY AGAIN SHORTLY"
144               CALL 'LIB$WAIT' USING
145                                BY REFERENCE TIME_WAIT
146               MOVE 3 TO PROG_STAT
147
148          WHEN OTHER
149               CALL 'LIB$STOP' USING BY VALUE RET_STATUS
150        END-EVALUATE
151     ELSE
152        MOVE 1 TO PROG_STAT
153     END-IF.
```

```
154
155     IF PROG_STAT IS EQUAL TO 1 THEN
156             DISPLAY SEQ_NO,"  ",SOC_SEC,"  ",LAST_NAME
157             PERFORM 200-DELETE-RECORD
158     END-IF.
159
160   200-DELETE-RECORD.
161     DISPLAY "Do you wish to delete this record? (Y/<CR>):  "
162             WITH NO ADVANCING.
163     ACCEPT DELETE_FLAG.
164     IF DELETE_FLAG = "Y" OR DELETE_FLAG = "y"
165             CALL 'SYS$DELETE' USING BY VALUE RAB_PTR 0 0
166                     GIVING RET_STATUS
167             IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
168                     USING BY VALUE RET_STATUS
169     END-IF.
170
171   250-CLEANUP.
172     CALL 'SYS$CLOSE' USING BY VALUE FAB_PTR 0 0
173                      GIVING RET_STATUS.
174     CALL 'LIB$SHOW_TIMER'.
175          STOP RUN.
```

COBOL 1 (Sheet 5 of 10)

BASIC External Routines (1 of 2)

```
 1  200         SUB SETFAB (FAB$TYPE FAB)
 2     !                                                  BAS_SETRMS.BAS
 3     !
 4     !   EXTERNAL BASIC routines called by COBPARSE_INDX.COB or
 5     !                                    by BASPARSE_INDX.BAS
 6     !
 7     OPTION  TYPE =  EXPLICIT
 8
 9     %INCLUDE        'FABRABDEF.BAS'
10
11    !  Extension # blocks if file is extended
12     FAB::FAB$W_DEQ = 10
13
14    !  File access desired for USER
15     FAB::FAB$B_FAC = (FAB::FAB$B_FAC OR FAB$M_DEL OR FAB$M_GET &
16             OR FAB$M_PUT OR FAB$M_UPD)
17
18    !  File options desired
19     FAB::FAB$L_FOP = (FAB::FAB$L_FOP OR FAB$M_DFW)  !  deferred write
20
21    !  # global buffers if wish to use them or set to zero if wish to
22    !       override global and use local buffers if someone already
23    !       has file opened with global buffers enabled
24    !  The following statement must be moved to SETRAB routine
25    !  prior to SYS$CONNECT call.
26    !
27    !       FAB::FAB$W_GBC = ?
28
29    !  Sharing attributes - what others can do or set TO FAB$M_SHRNIL
30     FAB::FAB$B_SHR = (FAB::FAB$B_SHR OR FAB$M_SHRPUT &
31             OR FAB$M_SHRGET OR FAB$M_SHRDEL         &
32             OR FAB$M_SHRUPD)
33
34     END SUB
35
36
37  400        SUB SETRAB      (RAB$TYPE RAB,&
38                     LONG REC_PTR, &
39                     WORD REC_LENGTH,&
40                     LONG KEY_PTR,&
41                     WORD KEY_LENGTH)
42
43     OPTION  TYPE =  EXPLICIT
44
45     %INCLUDE        'FABRABDEF.BAS'
46
47    !
48    !  Provide address of key_value if keyed retrieval to be used
49    ! and size of key
50     RAB::RAB$L_KBF  =  KEY_PTR
51     RAB::RAB$B_KSZ  =  KEY_LENGTH
```

COBOL 1 (Sheet 6 of 10)

BASIC External Routines (2 of 2)

```
52
53    !  Specify key of reference
54      RAB::RAB$B_KRF  =  0              ! primary key
55
56    !  Specify number of local buffers you want RMS to allocate on CONNECT
57      RAB::RAB$B_MBF  =  3
58
59    !  Specify type of mode to use for record access
60      RAB::RAB$B_RAC  =  RAB$C_KEY     ! RAB$C_SEQ or RAB$C_RFA
61
62    !
63    !  Provide address and length of output record buffer
64      RAB::RAB$L_RBF  =  REC_PTR
65      RAB::RAB$W_RSZ  =  REC_LENGTH
66    !
67    !  Provide address and length of input record buffer
68      RAB::RAB$L_UBF  =  REC_PTR
69      RAB::RAB$W_USZ  =  REC_LENGTH
70
71    !  Enable any record processing options to be used for entire run
72      RAB::RAB$L_ROP  =  (RAB::RAB$L_ROP OR RAB$M_FDL)  !  fast delete
73
74      END SUB
```

FORTRAN External Routines (1 of 2)

```
1  !                                                    FOR_SETRMS.FOR
2  !
3  !   EXTERNAL FORTRAN routines called by COBPARSE_INDX.COB or
4  !                              by FORPARSE_INDX.FOR
5  !
6      SUBROUTINE SETFAB (FAB)
7      IMPLICIT        NONE
8
9      INCLUDE         '($FABDEF)'
10     RECORD  /FABDEF/  FAB
11
12 !   Extension # blocks if file is extended
13     FAB.FAB$W_DEQ = 10
14
15 !   File access desired for USER
16     FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_DEL .OR. FAB$M_GET
17     1        .OR. FAB$M_PUT .OR. FAB$M_UPD
18
19 !   File options desired
20     FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_DFW    !  deferred write
21
22 !   # global buffers if wish to use them or set to zero if wish to
23 !   override global and use local buffers if someone already
24 !   has file opened with global buffers enabled
25 !   The following statement must be moved to SETRAB routine
26 !   prior to SYS$CONNECT call.
27 !   FAB.FAB$W_GBC = ?
28
29 !   Sharing attributes - what others can do or set TO FAB$M_SHRNIL
30     FAB.FAB$B_SHR = FAB.FAB$B_SHR .OR. FAB$M_SHRPUT
31     1        .OR. FAB$M_SHRGET .OR. FAB$M_SHRDEL
32     2        .OR. FAB$M_SHRUPD
33
34     RETURN
35     END
36
37
38     SUBROUTINE SETRAB (RAB,REC_PTR,REC_LENGTH,KEY_PTR,KEY_LENGTH)
39     IMPLICIT        NONE
40
41     INTEGER*4       REC_PTR,KEY_PTR
42     INTEGER*2       REC_LENGTH,KEY_LENGTH
43     INCLUDE         '($RABDEF)'
44     RECORD  /RABDEF/  RAB
45 !
46 !   Provide address of key_value if keyed retrieval to be used
47 !   and size of key
48     RAB.RAB$L_KBF   =  KEY_PTR
49     RAB.RAB$B_KSZ   =  KEY_LENGTH
50
51 !   Specify key of reference
52     RAB.RAB$B_KRF   =  0            ! primary key
53
```

COBOL 1 (Sheet 8 of 10)

FORTRAN External Routines (2 of 2)

```
54  !  Specify number of local buffers you want RMS to allocate on CONNECT
55     RAB.RAB$B_MBF  =  3
56
57  !  Specify type of mode to use for record access
58     RAB.RAB$B_RAC  =  RAB$C_KEY     ! RAB$C_SEQ or RAB$C_RFA
59
60  !
61  !  Provide address and length of output record buffer
62     RAB.RAB$L_RBF  =  REC_PTR
63     RAB.RAB$W_RSZ  =  REC_LENGTH
64  !
65  !  Provide address and length of input record buffer
66     RAB.RAB$L_UBF  =  REC_PTR
67     RAB.RAB$W_USZ  =  REC_LENGTH
68
69  !  Enable any record processing options to be used for entire run
70     RAB.RAB$L_ROP  =  RAB.RAB$L_ROP .OR. RAB$M_FDL  !  fast delete
71
72     RETURN
73     END
```

COBOL 1 (Sheet 9 of 10)

PASCAL External Routines (1 of 2)

```
1  {                                                          PAS_SETRMS.PAS
2
3      EXTERNAL PASCAL routines called by COBPARSE_INDX.COB
4                      and         by PASPARSE_INDX.PAS
5  }
6
7      [INHERIT ('SYS$LIBRARY: STARLET')] MODULE RMS_SETTINGS;
8
9      TYPE
10             WORD_INTEGER = [WORD] 0..64534;
11
12     [GLOBAL] PROCEDURE SETFAB (VAR FAB : FAB$TYPE);
13
14     BEGIN
15
16 {   Extension # blocks if file is extended }
17     FAB.FAB$W_DEQ := 10;
18
19 {   File access desired for USER }
20     FAB.FAB$B_FAC := FAB$M_DEL + FAB$M_GET + FAB$M_PUT + FAB$M_UPD;
21
22 {   File options desired  }
23     FAB.FAB$L_FOP := FAB$M_DFW;      {  deferred write }
24
25 {   # global buffers if wish to use them or set to zero if wish
26     to override global and use local buffers if someone already
27     has file opened  with global buffers enabled
28     The following statement must be moved to SETRAB routine
29 {   prior to SYS$CONNECT call.                          }
30 {   FAB.FAB$W_GBC := #;  }
31
32
33 {   Sharing attributes - what others can do or set TO FAB$M_SHRNIL }
34     FAB.FAB$B_SHR := FAB$M_SHRPUT + FAB$M_SHRGET + FAB$M_SHRDEL
35             + FAB$M_SHRUPD;
36
37     END;  {  SETFAB }
38
39
40
41     [GLOBAL] PROCEDURE SETRAB (VAR RAB : RAB$TYPE;
42             REC_PTR          : INTEGER;
43             REC_LENGTH       : WORD_INTEGER;
44             KEY_PTR          : INTEGER;
45             KEY_LENGTH       : WORD_INTEGER);
46     BEGIN
47
48 {   Provide address of key_value if keyed retrieval to be used
49     and size of key }
50     RAB.RAB$L_KBF   :=  KEY_PTR;
51     RAB.RAB$B_KSZ   :=  KEY_LENGTH;
52
```

COBOL 1 (Sheet 10 of 10)

PASCAL External Routines (2 of 2)

```
53  {  Specify key of reference }
54     RAB.RAB$B_KRF  :=  0;              { primary key }
55
56  {  Specify number of local buffers you want RMS to allocate on CONNECT }
57     RAB.RAB$B_MBF  :=  3;
58
59  {  Specify type of mode to use for record access }
60     RAB.RAB$B_RAC  :=  RAB$C_KEY;    { RAB$C_SEQ or RAB$C_RFA }
61
62  {  Provide address and length of output record buffer }
63     RAB.RAB$L_RBF  :=  REC_PTR;
64     RAB.RAB$W_RSZ  :=  REC_LENGTH;
65
66  {  Provide address and length of input record buffer }
67     RAB.RAB$L_UBF  :=  REC_PTR;
68     RAB.RAB$W_USZ  :=  REC_LENGTH;
69
70  {  Enable any record processing options to be used for entire run }
71     RAB.RAB$L_ROP  :=  RAB$M_FDL;             {  fast delete}
72
73     END;  { SETRAB }
74      END.  { MODULE RMS_SETTINGS }
```

COBOL 2 (Sheet 1 of 5)

```
1  *                                          COBMAR_PARSE_INDX.COB
2  *
3  *  COBOL PROGRAM which sets up RMS structures using FDL$PARSE
4  *     and calls TWO external non-COBOL routines to SET values in
5  *     FAB & RAB. The program retrieves records randomly by key value
6  *     entered by USER and gives user option to DELETE record.
7  *
8  *  This version is appropriate for linking in the external routines
9  *     written in MACRO:
10 *
11 *  $ LINK COBPARSE_INDX,COBMAR_SETRMS
12 *
13 *
14 *  See COB_PARSE_INDX.COB for BASIC, FORTRAN and PASCAL versions.
15 *
16 IDENTIFICATION DIVISION.
17 *
18 PROGRAM-ID. COBMAR_PARSE_INDX.
19 *
20 *
21 ENVIRONMENT DIVISION.
22
23 DATA DIVISION.
24
25 WORKING-STORAGE SECTION.
26 01  IN_REC.
27     02  SEQ_NO              PIC X(7).
28     02  LAST_NAME  PIC X(15).
29     02  FIRST_NAME PIC X(10).
30     02  SOC_SEC             PIC X(9).
31     02  STREET              PIC X(18).
32     02  CITY                PIC X(14).
33     02  STATE               PIC XX.
34     02  ZIP_CODE   PIC X(5).
35 * CONSTANTS
36 01  INREC_LENGTH   PIC 9(5) COMP   VALUE IS 80.
37 01  KEY_VALUE_LENGTH       PIC 9(5) COMP   VALUE IS 7.
38 01  TIME_WAIT              USAGE COMP-1    VALUE IS 10.0.
39 * VARIABLES
40 01  FAB_PTR                USAGE IS POINTER.
41 01  RAB_PTR                USAGE IS POINTER.
42 01  INREC_PTR              POINTER         VALUE REFERENCE IN_REC.
43 01  KEY_VALUE              PIC X(7)        JUSTIFIED RIGHT.
44 01  KEY_VALUE_PTR  POINTER         VALUE REFERENCE KEY_VALUE.
45 01  FDL_FILE               PIC  X(80).
46 01  RET_STATUS             PIC S9(9) COMP.
47 01  DELETE_FLAG            PIC X           VALUE IS "N".
48
49 01  PROG_STAT              PIC 9.
50     88  NO_ERROR                   VALUE 1.
51     88  SOME_ERROR                 VALUE 2.
```

```
52        88  WAIT_READ_AGAIN                          VALUE 3.
53        88  END_OF_INPUT                             VALUE 4.
54
55    01  RMS$_RNF                 PIC 9(9) COMP  VALUE EXTERNAL RMS$_RNF.
56    01  RMS$_RLK                 PIC 9(9) COMP  VALUE EXTERNAL RMS$_RLK.
57
58    PROCEDURE DIVISION.
59    000-BEGIN.
60
61        CALL 'LIB$INIT_TIMER'.
62
63        DISPLAY 'Enter FDL file name:  ' WITH NO ADVANCING.
64        ACCEPT FDL_FILE.
65    *
66    *    Call FDL$PARSE to set up RMS structures (FAB, RAB)
67        CALL 'FDL$PARSE' USING BY DESCRIPTOR FDL_FILE
68                               BY REFERENCE FAB_PTR RAB_PTR
69                               BY VALUE 0 0 0
70                         GIVING RET_STATUS.
71        IF RET_STATUS IS FAILURE CALL 'LIB$STOP'.
72    *
73    *   call external routine SETFAB in a non_COBOL language
74    *   that supports FAB structure to set any fields
75    *           needed which were not set in FDL file
76        CALL 'SETFAB' USING BY VALUE FAB_PTR.
77    *
78    *   Open input data file
79        CALL 'SYS$OPEN' USING BY VALUE FAB_PTR 0 0
80                        GIVING RET_STATUS.
81        IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
82                          USING BY VALUE RET_STATUS.
83    *
84    *   call external routine SETRAB WRITTEN IN MACRO and at
85    *           a minimum initialize addresses of record and keyvalue
86    *   buffers and their sizes
87    *
88    *   NOTE:  If the external routine were written in BASIC,
89    *   FORTRAN or PASCAL, ONLY the RAB_PTR would be passed
90    *   BY VALUE and all other arguments BY REFERENCE.
91    *
92    *
93        CALL 'SETRAB' USING  BY VALUE RAB_PTR
94                             BY VALUE       INREC_PTR
95                             BY REFERENCE   INREC_LENGTH
96                             BY VALUE       KEY_VALUE_PTR
97                             BY REFERENCE   KEY_VALUE_LENGTH.
98    *
99    *   Connect record
100       CALL 'SYS$CONNECT' USING BY VALUE RAB_PTR 0 0
101                          GIVING RET_STATUS.
102       IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
103                         USING BY VALUE RET_STATUS.
104
```

```
105     DISPLAY "************************************".
106     DISPLAY "     HIT CNTL+Z TO STOP RUN"
107     DISPLAY "************************************".
108
109     MOVE 0 TO PROG_STAT.
110     PERFORM 100-CHOOSE-RECORD UNTIL END_OF_INPUT.
111     PERFORM 250-CLEANUP.
112
113   100-CHOOSE-RECORD.
114   *
115   *   Prompt user for key value
116     IF PROG_STAT NOT EQUAL TO 3 THEN
117             DISPLAY "Enter SEQ_NO:  " WITH NO ADVANCING
118             ACCEPT KEY_VALUE WITH CONVERSION AT END
119                MOVE 4 TO PROG_STAT
120     END-IF.
121
122     IF PROG_STAT NOT EQUAL TO 4 THEN
123             PERFORM 150-READ-BY-PRIMARY-KEY.
124
125
126   150-READ-BY-PRIMARY-KEY.
127
128     CALL 'SYS$GET' USING BY VALUE RAB_PTR 0 0
129                    GIVING RET_STATUS.
130
131     IF RET_STATUS IS FAILURE THEN
132        MOVE 2 TO PROG_STAT
133        EVALUATE RET_STATUS
134
135          WHEN RMS$_RNF
136               DISPLAY "NO RECORD WITH SEQ_NO = ",KEY_VALUE
137
138          WHEN RMS$_RLK
139               DISPLAY "RECORD CURRENTLY LOCKED - ",
140                     "WILL TRY AGAIN SHORTLY"
141               CALL 'LIB$WAIT' USING
142                                BY REFERENCE TIME_WAIT
143               MOVE 3 TO PROG_STAT
144
145          WHEN OTHER
146               CALL 'LIB$STOP' USING BY VALUE RET_STATUS
147        END-EVALUATE
148     ELSE
149        MOVE 1 TO PROG_STAT
150     END-IF.
151
152     IF PROG_STAT IS EQUAL TO 1 THEN
153             DISPLAY SEQ_NO,"  ",SOC_SEC,"  ",LAST_NAME
154             PERFORM 200-DELETE-RECORD
155     END-IF.
```

COBOL 2 (Sheet 4 of 5)

```
156
157  200-DELETE-RECORD.
158     DISPLAY "Do you wish to delete this record? (Y/<CR>):  "
159          .   WITH NO ADVANCING.
160     ACCEPT DELETE_FLAG.
161     IF DELETE_FLAG = "Y" OR DELETE_FLAG = "y"
162             CALL 'SYS$DELETE' USING BY VALUE RAB_PTR 0 0
163                     GIVING RET_STATUS
164             IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
165                     USING BY VALUE RET_STATUS
166     END-IF.
167
168  250-CLEANUP.
169     CALL 'SYS$CLOSE' USING BY VALUE FAB_PTR 0 0
170                     GIVING RET_STATUS.
171     CALL 'LIB$SHOW_TIMER'.
172          STOP RUN.
```

COBOL 2 (Sheet 5 of 5)

MACRO External Routines (1 of 1)

```
1   ;                                                       COBMAR_SETRMS.MAR
2   ;
3   ;  EXTERNAL MACRO ROUTINES CALLED BY COBPARSE_INDX.COB
4   ;
5
6
7       .ENTRY  SETFAB,^M<>
8
9       $FAB_STORE          FAB=@L^4(AP),-
10                          DEQ=#10,-           ; extension # blocks if file extended
11                          FAC=<DEL,GET,PUT,UPD>,-   ; what you can do
12                          FOP=DFW,-           ; file options - deferred write
13                          SHR=<PUT,GET,DEL,UPD>     ; what others can do
14      RET
15
16
17      .ENTRY  SETRAB,^M<>
18
19      $RAB_STORE          RAB=@L^4(AP),-   ; RAB address
20                          RBF=@L^8(AP),-   ; output record buffer
21                          RSZ=@W^12(AP),-
22                          UBF=@L^8(AP),-   ; input record buffer
23                          USZ=@W^12(AP),-
24                          KBF=@L^16(AP),- ; input KEY buffer
25                          KSZ=@W^20(AP),-
26                          KRF=#0,-         ; primary key
37                          MBF=#3,-         ; number of local buffers
28                          RAC=KEY,-        ; type of record access SEQ or RFA
29                          ROP=FDL          ; record options for whole run -
30  ;                                                   fast delete
31
32      RET
33
34      .END
```

```
1   !                                                       FORPARSE_INDX.FOR
2   !  NOTE:  $LINK FORPARSE_INDX,FOR_SETRMS
3   !
4   !         FOR_SETRMS.FOR contains two external subroutines:
5   !
6   !                     SETFAB
7   !                     SETRAB
8   !
9   !  FORTRAN program using FDL$PARSE to set up RMS structures
10  !    for accessing an INDEXED file rather than using
11  !    USEROPEN function.  In addition, all access to the
12  !    file is done using RMS services rather than regular
13  !    FORTRAN I/O.  User is prompted for key value for
14  !    random retrieval of records and given option to
15  !    delete any record retrieved.
16  !
17  !  Program assumes FILE SHARING and RECORD LOCKING
18  !
19      PROGRAM FORPARSE_INDX
20      IMPLICIT                    NONE
21
22      REAL                        TIME_WAIT
23      PARAMETER                   (TIME_WAIT=10.0)  ! No. seconds wait
24                                                    !  if record locked
25
26      INTEGER*2                   INREC_LENGTH/80/
27      INTEGER*2                   KEY_VALUE_LENGTH/7/
28
29      STRUCTURE /ACCOUNT_STRUC/
30              CHARACTER*7         SEQ_NO            ! KEY 0
31              CHARACTER*15        LAST_NAME         ! KEY 1
32              CHARACTER*10        FIRST_NAME
33              CHARACTER*9         SOC_SEC
34              CHARACTER*18        STREET
35              CHARACTER*14        CITY
36              CHARACTER*2         STATE
37              CHARACTER*5         ZIP_CODE
38      END STRUCTURE
39
40      RECORD /ACCOUNT_STRUC/      IN_REC
41
42      INTEGER                     FAB_PTR,
43      1                           RAB_PTR,
44      2                           RET_STATUS,
45      3                           LENGTH_FDLFILE,
46      4                           KEY_IN,
47      5                           INREC_PTR,
48      6                           KEY_VALUE_PTR
49      CHARACTER*80                FDL_FILE
50      CHARACTER*7                 KEY_VALUE
51      CHARACTER*1                 DELETE_FLAG
52
```

```
53      INTEGER                 PROG_STAT/0/    !  VALUE  1 = no error
54                                              !  VALUE  2 = some error
55                                              !  VALUE  3 = wait-read-again
56
57      INTEGER                 FDL$PARSE,
58      1                       SYS$OPEN,
59      2                       SYS$CONNECT,
60      3                       SYS$GET,
61      4                       SYS$DELETE,
62      5                       SYS$CLOSE
63
64      INCLUDE '($RMSDEF)'
65
66      CALL LIB$INIT_TIMER()
67
68      WRITE (6,1)
69      READ (5,2) length_FDLFILE,FDL_FILE
70
71      RET_STATUS = FDL$PARSE(
72      1       FDL_FILE(1:length_FDLFILE),
73      2       FAB_PTR,                        ! Ptr to FAB returned byFDL
74      3       RAB_PTR,,,)                     ! Ptr to RAB returned by FDL
75      IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))
76
77      CALL SETFAB(%VAL(FAB_PTR))
78
79      RET_STATUS = SYS$OPEN(%VAL(FAB_PTR),,)
80      IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))
81
82      INREC_PTR       = %LOC(IN_REC)      ! internal addr input rec buff
83      KEY_VALUE_PTR   = %LOC(KEY_VALUE)   ! internal addr key value
84
85      CALL SETRAB(
86      1       %VAL(RAB_PTR),              !  Ptr to RAB
87      2       INREC_PTR,                  !  Internal addr record buffer
88      3       INREC_LENGTH,               !  Length record buffer
89      4       KEY_VALUE_PTR,              !  Internal addr key value
90      5       KEY_VALUE_LENGTH)           !  Length key value
91
92      RET_STATUS = SYS$CONNECT(%VAL(RAB_PTR),,)
93      IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))
94
95      WRITE (6,6)
96      WRITE (6,5)             !  directions to stop run
97      WRITE (6,6)
98
99      WRITE (6,3)
100     READ (5,4) KEY_IN
101
```

```
102     DO WHILE (KEY_IN .GT. 0)
103   !
104   !   Convert integer KEY_IN to right justified character string KEY_VALUE
105
106       IF (PROG_STAT .NE. 3)
107     1         WRITE (UNIT=KEY_VALUE,FMT='(I7)') KEY_IN
108
109       RET_STATUS = SYS$GET (%VAL(RAB_PTR),,)
110
111        IF (.NOT.RET_STATUS) THEN
112             PROG_STAT = 2
113
114             IF (RET_STATUS .EQ. RMS$_RNF) THEN
115                WRITE (6,11) KEY_VALUE
116             ELSE
117
118               IF (RET_STATUS .EQ. RMS$_RLK) THEN
119                   WRITE (6,12)
120                   CALL LIB$WAIT (TIME_WAIT)
121                   PROG_STAT = 3
122               ELSE
123                CALL LIB$STOP(%VAL(RET_STATUS))
124               END IF
125             END IF
126          ELSE
127             PROG_STAT = 1
128          END IF
129
130          IF (PROG_STAT .EQ. 1) THEN
131
132                WRITE (6,*) IN_REC.SEQ_NO,'  ',IN_REC.SOC_SEC,'  ',
133     1                      IN_REC.LAST_NAME
134
135                WRITE (6,14)
136                READ  (5,15) DELETE_FLAG
137
138                IF (DELETE_FLAG .EQ. 'Y'
139     1                     .OR. DELETE_FLAG .EQ. 'y') THEN
140                     RET_STATUS = SYS$DELETE(%VAL(RAB_PTR),,)
141                     IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))
142                END IF
143           END IF  !  prog_stat = 1
144
145           IF (PROG_STAT .NE. 3) THEN
146
147               WRITE (6,3)
148               READ (5,4) KEY_IN
149           END IF  !  prog_stat .ne. 3
150
151     END DO
```

```
152
153      CALL SYS$CLOSE(%VAL(FAB_PTR),,)
154
155      CALL LIB$SHOW_TIMER()
156
157      CALL EXIT
158
159    1 FORMAT ('$Enter name of FDL file:  ')
160    2 FORMAT (Q,A)
161    3 FORMAT ('$Enter SEQ_NO:  ')
162    4 FORMAT (I)
163    6 FORMAT ('**********************************************')
164    5 FORMAT ('     Hit <CR> or enter zero to stop run')
165   11 FORMAT (' Record NOT FOUND with SEQ_NO = ',A)
166   12 FORMAT (' Record currently LOCKED ',
167      1                ' - will try again shortly')
168   14 FORMAT ('$Do you wish to delete this record? (Y/<CR>):  ')
169   15 FORMAT (A)
170      END
```

```
 1  !                                                    FOR_SETRMS.FOR
 2  !
 3  !   EXTERNAL FORTRAN routines called by COBPARSE_INDX.COB or
 4  !                                  by FORPARSE_INDX.FOR
 5  !
 6      SUBROUTINE SETFAB (FAB)
 7      IMPLICIT        NONE
 8
 9      INCLUDE         '($FABDEF)'
10      RECORD  /FABDEF/  FAB
11
12  !   Extension # blocks if file is extended
13      FAB.FAB$W_DEQ = 10
14
15  !   File access desired for USER
16      FAB.FAB$B_FAC = FAB.FAB$B_FAC .OR. FAB$M_DEL .OR. FAB$M_GET
17     1        .OR. FAB$M_PUT .OR. FAB$M_UPD
18
19  !   File options desired
20      FAB.FAB$L_FOP = FAB.FAB$L_FOP .OR. FAB$M_DFW    !  deferred write
21
22  !   # global buffers if wish to use them or set to zero if wish to
23  !   override global and use local buffers if someone already
24  !   has file opened with global buffers enabled
25  !   The following statement must be moved to SETRAB routine
26  !   prior to SYS$CONNECT call.
27  !   FAB.FAB$W_GBC = ?
28
29  !   Sharing attributes - what others can do or set TO FAB$M_SHRNIL
30      FAB.FAB$B_SHR = FAB.FAB$B_SHR .OR. FAB$M_SHRPUT
31     1        .OR. FAB$M_SHRGET .OR. FAB$M_SHRDEL
32     2        .OR. FAB$M_SHRUPD
33
34      RETURN
35      END
36
37
38      SUBROUTINE SETRAB (RAB,REC_PTR,REC_LENGTH,KEY_PTR,KEY_LENGTH)
39      IMPLICIT        NONE
40
41      INTEGER*4       REC_PTR,KEY_PTR
42      INTEGER*2       REC_LENGTH,KEY_LENGTH
43      INCLUDE         '($RABDEF)'
44      RECORD  /RABDEF/  RAB
45  !
46  !   Provide address of key_value if keyed retrieval to be used
47  !   and size of key
48      RAB.RAB$L_KBF   =  KEY_PTR
49      RAB.RAB$B_KSZ   =  KEY_LENGTH
50
```

FORTRAN (Sheet 6 of 6)

```
51  !  Specify key of reference
52     RAB.RAB$B_KRF  =  0              ! primary key
53
54  !  Specify number of local buffers you want RMS to allocate on CONNECT
55     RAB.RAB$B_MBF  =  3
56
57  !  Specify type of mode to use for record access
58     RAB.RAB$B_RAC  =  RAB$C_KEY      ! RAB$C_SEQ or RAB$C_RFA
59
60  !
61  !  Provide address and length of output record buffer
62     RAB.RAB$L_RBF  =  REC_PTR
63     RAB.RAB$W_RSZ  =  REC_LENGTH
64  !
65  !  Provide address and length of input record buffer
66     RAB.RAB$L_UBF  =  REC_PTR
67     RAB.RAB$W_USZ  =  REC_LENGTH
68
69  !  Enable any record processing options to be used for entire run
70     RAB.RAB$L_ROP  =  RAB.RAB$L_ROP .OR. RAB$M_FDL  !  fast delete
71
72     RETURN
73     END
```

```
1  {                                                    PASPARSE_INDX.PAS
2
3           $LINK PASPARSE_INDX,PAS_SETRMS
4
5     PAS_SETRMS is EXTERNAL module containing two routines:
6
7                    SETFAB
8                    SETRAB
9
10    PASCAL program using FDL$PARSE to set RMS control blocks for
11    an indexed file.  Thereafter, all access is done using
12    calls to RMS directly.  User is prompted for SEQ_NO
13    of record to be retrieved randomly and given option
14    of deleting any records retrieved.
15
16    Program assumes FILE SHARING and RECORD LOCKING
17 }
18    [INHERIT ('SYS$LIBRARY: STARLET')]
19    PROGRAM PASPARSE_INDX(INPUT,OUTPUT);
20
21    CONST
22          TIME_WAIT = 10.0;                    { Number of seconds wait
23                                                   if record locked }
24    TYPE
25
26      ACCOUNT_STRUC = RECORD
27            SEQ_NO     : [KEY(0)] PACKED ARRAY [1..7] OF CHAR;
28            LAST_NAME  : [KEY(1)] PACKED ARRAY [1..15] OF CHAR;
29            FIRST_NAME : PACKED ARRAY [1..10] OF CHAR;
30            SOC_SEC    : PACKED ARRAY [1..9] OF CHAR;
31            STREET     : PACKED ARRAY [1..18] OF CHAR;
32            CITY       : PACKED ARRAY [1..14] OF CHAR;
33            STATE      : PACKED ARRAY [1..2] OF CHAR;
34            ZIP_CODE   : PACKED ARRAY [1..5] OF CHAR;
35      END;
36
37      ACCOUNT_REC   =  FILE OF ACCOUNT_STRUC;
38
39      WORD_INTEGER  =  [WORD] 0..64534;
40
41      PTR_TO_FAB    =  ^FAB$TYPE;
42      PTR_TO_RAB    =  ^RAB$TYPE;
43
44    VAR
45
46      FAB_PTR       :  PTR_TO_FAB;
47      RAB_PTR       :  PTR_TO_RAB;
48      IN_REC        :  ACCOUNT_STRUC;
49      INREC_LENGTH  :  WORD_INTEGER := 80;
50      RET_STATUS    :  INTEGER;
51      FDL_FILENAME  :  [VOLATILE] PACKED ARRAY [1..80] OF CHAR;
52      KEY_IN        :  INTEGER;
```

```
53        KEY_VALUE      :  VARYING [7] OF CHAR;
54        KEY_LENGTH     :  WORD_INTEGER := 7;
55        INREC_PTR,
56        KEY_VALUE_PTR  :  INTEGER;
57        DELETE_FLAG    :  CHAR;
58
59        PROG_STAT      :  INTEGER:=0;      { VALUE  1 = no error
60                                             VALUE  2 = some error
61                                             VALUE  3 = wait-read-again  }
62
63      PROCEDURE LIB$WAIT (
64              num_secs     : REAL); EXTERN;
65
66      PROCEDURE LIB$STOP (
67              %IMMED cond_value : INTEGER); EXTERN;
68
69      FUNCTION FDL$PARSE (FDLFILE : [CLASS_S] PACKED ARRAY
70                          [L..U:INTEGER] OF CHAR;
71                          VAR FAB_PTR : PTR_TO_FAB;
72                          VAR RAB_PTR : PTR_TO_RAB): INTEGER; EXTERN;
73
74      PROCEDURE SETFAB (VAR FAB : FAB$TYPE); EXTERN;
75
76      PROCEDURE SETRAB (VAR RAB : RAB$TYPE;
77                         INREC_PTR : INTEGER;
78                         INREC_LENGTH : WORD_INTEGER;
79                         KEY_VALUE_PTR : INTEGER;
80                         KEY_LENGTH : WORD_INTEGER); EXTERN;
81
82      PROCEDURE LIB$INIT_TIMER (
83              VAR HANDLER_ADR : INTEGER:= %IMMED 0); EXTERN;
84
85      PROCEDURE LIB$SHOW_TIMER (
86              HANDLER_ADR : INTEGER := %IMMED 0;
87              CODE : INTEGER := %IMMED 0;
88              [IMMEDIATE,UNBOUND] PROCEDURE ACTION_RTN (OUT_STR :
89                [CLASS_S] PACKED ARRAY [L..U:INTEGER]
90                OF CHAR):= %IMMED 0;
91              %IMMED USER_ARG : INTEGER := %IMMED 0); EXTERN;
92
93      BEGIN  { MAIN }
94
95        LIB$INIT_TIMER;
96
97        WRITE ('Enter FDL filename:  ');
98        READLN  (FDL_FILENAME);
99
100       RET_STATUS := FDL$PARSE (FDL_FILENAME,FAB_PTR,RAB_PTR);
101
102       IF NOT ODD(RET_STATUS) THEN LIB$STOP (RET_STATUS);
103
```

```
104       SETFAB (%IMMED FAB_PTR);
105
106       RET_STATUS := $OPEN(%IMMED FAB_PTR,,);
107       IF NOT ODD(RET_STATUS) THEN LIB$STOP (RET_STATUS);
108
109       INREC_PTR     := IADDRESS(IN_REC);
110       KEY_VALUE_PTR := IADDRESS(KEY_VALUE.BODY);
111
112       SETRAB (%IMMED RAB_PTR,
113               INREC_PTR,
114               INREC_LENGTH,
115               KEY_VALUE_PTR,
116               KEY_LENGTH);
117
118       RET_STATUS := $CONNECT (%IMMED RAB_PTR,,);
119
120       WRITELN ('************************************');
121       WRITELN ('     Enter ZERO to stop run');
122       WRITELN ('************************************');
123
124       WRITE ('Enter SEQ_NO:  ');
125       READLN (KEY_IN);
126
127       WHILE (KEY_IN > 0) DO
128       BEGIN  { while loop }
129
130  { Convert integer KEY_IN to right justified character string KEY_VALUE }
131       IF (PROG_STAT <> 3) THEN
132             WRITEV (KEY_VALUE,KEY_IN:7);
133
134           RET_STATUS := $GET(%IMMED RAB_PTR,,);
135
136         IF NOT ODD (RET_STATUS) THEN
137         BEGIN
138
139           PROG_STAT := 2;
140
141           CASE RET_STATUS OF
142
143               RMS$_RNF :  BEGIN
144                            WRITELN (' Record NOT FOUND with',
145                                ' SEQ_NO = ',KEY_VALUE);
146                           END;
147
148               RMS$_RLK :  BEGIN
149                            WRITELN (' Record currently LOCKED ',
150                                     ' - will try again shortly');
151                            LIB$WAIT (TIME_WAIT);
152                            PROG_STAT := 3;
153                           END
154
155               OTHERWISE
```

```
156                     LIB$STOP (%IMMED RET_STATUS);
157               END  { CASE }
158
159            END  { IF NOT ODD }
160            ELSE  PROG_STAT := 1;
161
162         IF (PROG_STAT = 1) THEN
163         BEGIN
164
165              WRITELN (IN_REC.SEQ_NO,'  ',IN_REC.SOC_SEC,'  ',
166                       IN_REC.LAST_NAME);
167
168              WRITE ('Do you wish to delete this record?',
169                              ' (Y/any char):  ');
170              READLN (DELETE_FLAG);
171
172              IF ((DELETE_FLAG = 'Y') OR (DELETE_FLAG = 'y')) THEN
173              BEGIN
174                       RET_STATUS := $DELETE(%IMMED RAB_PTR,,);
175                       IF NOT ODD(RET_STATUS) THEN
176                                LIB$STOP(RET_STATUS);
177              END;
178         END;  { prog_stat = 1}
179
180         IF (PROG_STAT <> 3) THEN
181         BEGIN
182
183              WRITE ('Enter SEQ_NO:  ');
184              READLN(KEY_IN);
185
186         END;  { prog_stat <> 3 }
187
188      END;  { while loop }
189
190      $CLOSE(%IMMED FAB_PTR,,);
191
192      LIB$SHOW_TIMER;
193
194      END.  { Main program }
```

```
 1  {                                                    PAS_SETRMS.PAS
 2
 3     EXTERNAL PASCAL routines called by COBPARSE_INDX.COB
 4                     and        by PASPARSE_INDX.PAS
 5  }
 6
 7     [INHERIT ('SYS$LIBRARY: STARLET')] MODULE RMS_SETTINGS;
 8
 9     TYPE
10             WORD_INTEGER = [WORD] 0..64534;
11
12     [GLOBAL] PROCEDURE SETFAB (VAR FAB : FAB$TYPE);
13
14     BEGIN
15
16  {  Extension # blocks if file is extended }
17     FAB.FAB$W_DEQ := 10;
18
19  {  File access desired for USER }
20     FAB.FAB$B_FAC := FAB$M_DEL + FAB$M_GET + FAB$M_PUT + FAB$M_UPD;
21
22  {  File options desired  }
23     FAB.FAB$L_FOP := FAB$M_DFW;      {  deferred write }
24
25  {  # global buffers if wish to use them or set to zero if wish
26     to override global and use local buffers if someone already
27     has file opened  with global buffers enabled
28     The following statement must be moved to SETRAB routine
29  {  prior to SYS$CONNECT call.                    }
30  {  FAB.FAB$W_GBC := #;  }
31
32
33  {  Sharing attributes - what others can do or set TO FAB$M_SHRNIL }
34     FAB.FAB$B_SHR := FAB$M_SHRPUT + FAB$M_SHRGET + FAB$M_SHRDEL
35             + FAB$M_SHRUPD;
36
37     END;  {  SETFAB }
38
39
40
41     [GLOBAL] PROCEDURE SETRAB (VAR RAB : RAB$TYPE;
42             REC_PTR          : INTEGER;
43             REC_LENGTH       : WORD_INTEGER;
44             KEY_PTR          : INTEGER;
45             KEY_LENGTH       : WORD_INTEGER);
46     BEGIN
47
48  {  Provide address of key_value if keyed retrieval to be used
49     and size of key }
50     RAB.RAB$L_KBF   :=  KEY_PTR;
51     RAB.RAB$B_KSZ   :=  KEY_LENGTH;
52
```

PASCAL (Sheet 6 of 6)

```
53  {  Specify key of reference }
54     RAB.RAB$B_KRF  :=  0;              { primary key }
55
56  {  Specify number of local buffers you want RMS to allocate on CONNECT }
57     RAB.RAB$B_MBF  :=  3;
58
59  {  Specify type of mode to use for record access }
60     RAB.RAB$B_RAC  :=  RAB$C_KEY;    { RAB$C_SEQ or RAB$C_RFA }
61
62  {  Provide address and length of output record buffer }
63     RAB.RAB$L_RBF  :=  REC_PTR;
64     RAB.RAB$W_RSZ  :=  REC_LENGTH;
65
66  {  Provide address and length of input record buffer }
67     RAB.RAB$L_UBF  :=  REC_PTR;
68     RAB.RAB$W_USZ  :=  REC_LENGTH;
69
70  {  Enable any record processing options to be used for entire run }
71     RAB.RAB$L_ROP  :=  RAB$M_FDL;      {  fast delete}
72
73     END;  { SETRAB }
74      END.  { MODULE RMS_SETTINGS }
```

MODULE 14
ADVANCED USE OF FILE SPECIFICATIONS

Major Topics

- Search lists and wildcards
- RMS procedures — $PARSE and $SEARCH

Source

Guide to VAX/VMS File Applications — Chapter 5

MODULE 14
ADVANCED USE OF FILE SPECIFICATIONS

Major Topics

- Search lists and wildcards
- RMS procedures — $PARSE and $SEARCH

Source

Guide to VAX/VMS File Applications — Chapter 5

SEARCH LISTS AND WILDCARDS

Full file specification:

NODE::DEVICE:[root.][DIRECTORY]filename.type; version-no

NOTE

Node and root are optional in the full file specification.

Defaults or Logical Names

Only node, device, or, if only a file name is specified, filename, may be a logical name. The translation may contain any filename element.

In addition to applying the process-default device and directory, RMS allows an application program to specify defaults for the device and directory components, as well as other components, of a file specification. The method that RMS uses to apply defaults and translate any logical names present is called file parsing. In effect, RMS merges the various default strings (after translating any logical names) to generate the file specification used to locate the file.

Search List

A search list is a logical name that contains more than one file specification.

Example

```
$ASSIGN   [SMITH]Test1.DAT,
            [SMITH]DATA2.DAT     SEARCH
```

A search list should be used when a predefined group of files is processed by a program that is not intended to be interactive. Using a search list is particularly desirable if the files have unrelated file names or if they are located on different directories or devices. A search list also minimizes processing time by searching for a definite group of files. If the search line does not contain any wildcards and the user wishes to process only the first match found, no special processing is required.

Wildcards

The following wildcards can be used with search lists or file specifications:

*, %, or ellipsis (...)

Example

```
$Assign [WOODS]TEST*.DAT    WILD
```

RMS DEFAULT FILE-PARSING ACTIVITIES

An RMS file service that operates on an unopened file (such as the Create and Open services) will perform the following file-parsing activity by default:

- Examine a file specification for validity
- Translate any logical names present
- Apply defaults
- Attempt to locate the file

If a name block is present, additional file-parsing activities can occur by default.

- Return the actual complete file specification used to access the file and its associated file identifier.
- Return the length of each component of a file specification, as well as other information about the file specification.

RMS FILE-PARSING ACTIVITIES NOT DONE BY DEFAULT

Certain RMS file services, including the Open and Create services, cannot process a file specification that contains a wildcard character. Therefore, these RMS file services must be preceded by another RMS file service called the Parse service. (If a search list with no wildcards is present, the Parse service is usually not needed.) The Parse service can be used to determine whether wildcards or search lists are present. It also initializes control block fields that are necessary to search for multiple files using the RMS Search service. To use the Search service, a name block must be present when the Parse service is invoked.

If a file specification contains one or more wildcards, it must be preprocessed using the Parse and Search services before the file can be located. The Parse service sets certain bit values in a name block field called the file name status bits field (NAM$L_FNB) that can be tested to determine whether a wildcard or a search list logical name is present. The Search service locates a file and specifies its name (without wildcards). If wildcards are present, you must first invoke the Search service before processing (opening or creating) the file. If wildcards are not present, the file can be processed without invoking the Search service, with one exception. If the user wishes more than one file in a search list to be processed, the Search service must be invoked as many times as needed to return the next file specification to be processed.

The sequence of special file processing steps required involves one call to the Parse service followed by one or more calls to the Search service prior to each file open. To process a single file, invoke the Search service only once; to process many files, invoke the Search service as many times as needed to return the next qualified file specification to be processed. When no more files match the file specification, the Search service returns a "no-more-files-found" message (RMS$_NMF). Two Run-Time Library routines, LIB$FIND_FILE and LIB$FILE_SCAN, perform functions that are similar to the Parse and Search services.

For general-purpose applications, the programmer may test for wildcards and/or search lists by invoking the Parse service and testing the appropriate bits in the NAM$L_FNB field. In cases where the program assumes only a single file, the results of this test may be used to explicitly disallow wildcards or search lists.

NAM$L_FNB Status Bits

Field Offset	Description
NAM$V_CNCL_DEV	Device name is a concealed device.
NAM$V_DIR_LVLS	Number of subdirectory levels (value is 0 if there is a user file directory only), a 3-bit field.
NAM$V_EXP_DEV	Device name is explicit.
NAM$V_EXP_DIR	Directory specification is explicit.
NAM$V_EXP_NAME	File name is explicit.
NAM$V_EXP_TYPE	File type is explicit.
NAM$V_EXP_VER	Version number is explicit.
NAM$V_GRP_MBR	Directory specification is in the group/member number format.
NAM$V_HIGHVER	A higher-numbered version(s) of the file exists (output from Create and Enter services).
NAM$V_LOWVER	A lower-numbered version(s) of the file exists (output from Create and Enter services).
NAM$V_NODE	File specification includes a node name.
NAM$V_PPF	File is indirectly accessed process permanent file.
NAM$V_QUOTED	File specification includes a quoted string; indicates that the file name length and address field contains a quoted string file specification. Applies to network operations or magnetic tape devices only.*
NAM$V_ROOT_DIR	Device name incorporates a root directory.

* To distinguish network quoted string file specifications from quoted strings containing ASCII "a" file names (supported for ANSI-labeled magnetic tapes), both the NAM$V_QUOTED and NAM$V_NODE bits will be set.

NAM$L_FNB Status Bits (Cont.)

Field Offset	Description
NAM$V_SEARCH_LIST	A search list logical name is present in the file specification.
NAM$V_WILDCARD	File specification string includes a wildcard; returned whenever any of the other wildcard bits are set.
NAM$V_WILD_DIR	Directory specification includes a wildcard character(s).
NAM$V_WILD_GRP	Group number contains a wildcard character(s).
NAM$V_WILD_MBR	Member number contains a wildcard character(s).
NAM$V_WILD_NAME	File name contains a wildcard character(s).
NAM$V_WILD_SFD1 to NAM$V_WILD_SFD7	Subdirectory 1 through 7 specification includes a wildcard character(s).
NAM$V_WILD_TYPE	File type contains a wildcard character(s).
NAM$V_WILD_UFD	User file directory specification includes a wildcard character(s).
NAM$V_WILD_VER	Version number contains a wildcard character(s).

Example 1. Processing Filename(s) With Wildcards Using LIB$FIND_FILE

(Sheet 1 of 2)

```
                                                        FORFIND1.FOR
!
!
!  NOTE:  TEST data for this program ---->    INTER*.DAT
!
!  FORTRAN program which opens a sequential
!  file.  It calls LIB$FIND_FILE
!  and recalls OPEN if the file spec. parses
!  to more than one file spec through use of
!  WILDCARDS
!
!


      PROGRAM FORSEARCH
      IMPLICIT          NONE

      INTEGER            IUNIT/1/
      INTEGER            RET_STATUS,
     1                   OPENFILE,
     2                   RMS_STS,
     3                   RMS_STV

      CHARACTER*255      FILENAME,
     1                   RESULT_SPEC,
     1                   RELATED_SPEC

      INTEGER            LEN_FILENAME,
     1                   USER_FLAGS/0/,
     2                   CONTEXT/0/

      STRUCTURE /EMPLOYEE_STRUC/
              CHARACTER*10     last_name
              CHARACTER*100    fill
              CHARACTER*2      seq_no
      END STRUCTURE

      RECORD /EMPLOYEE_STRUC/  IN_REC

      INTEGER            LIB$STOP,
     1                   LIB$FIND_FILE

      EXTERNAL            OPENFILE
      INCLUDE            '($RMSDEF)'

      WRITE (6,1)
    1 FORMAT ('$','Enter filename:  ')
      READ (5,2)  len_filename,FILENAME
    2 FORMAT (Q,A)


  100         RET_STATUS = LIB$FIND_FILE (FILENAME,
     1                                    RESULT_SPEC,
     2                                    CONTEXT,
     3                                    ,                  !  default file spec.
     4                                    RELATED_SPEC,
     5                                    ,                  !  stv_addr
     6                                    USER_FLAGS)

      IF (RET_STATUS .EQ. RMS$_NMF) GO TO 300

```

Example 1 (Sheet 2 of 2)

```
61      OPEN (UNIT=IUNIT,FILE=RESULT_SPEC,STATUS='OLD',
62     1      FORM='UNFORMATTED',RECORDTYPE='FIXED',
63     2      ORGANIZATION='INDEXED',ACCESS='SEQUENTIAL')
64
65
66      READ (IUNIT,END=200,IOSTAT=RET_STATUS) IN_REC
67
68      DO WHILE (RET_STATUS .EQ. 0)
69
70              WRITE (6,*) IN_REC.LAST_NAME,'  ',IN_REC.SEQ_NO
71
72              READ (IUNIT,IOSTAT=RET_STATUS) IN_REC
73              IF (RET_STATUS .EQ. -1) GOTO 200
74
75      END DO
76
77              CALL ERRSNS(,RMS_STS,RMS_STV,IUNIT,)
78              CALL LIB$STOP(%VAL(RMS_STS),%VAL(RMS_STV))
79
80  200         CLOSE (IUNIT)
81
82      GO TO 100
83
84  300         CALL EXIT
85      END
```

LIB$FIND_FILE -- FIND FILE

LIB$FIND_FILE is called with a wildcard file specification for which it searches. LIB$FIND_FILE returns all file specifications that satisfy that wildcard file specification.

Format

```
LIB$FIND_FILE  file-spec,result-spec,context
               [,default-spec][,related-spec]
               [,stv-addr][,user-flags]
```

Arguments

File-spec

```
type:         character string
access:       read only
mechanism:    by descriptor
```

The file specification may contain wildcards that LIB$FIND_FILE uses to search for the desired file. The file-spec argument is the address of a descriptor pointing to the file specification. The maximum length of a file specification is 255 bytes.

The file specification used may also contain a search list logical name. If present, the search list logical name elements can be used as accumulative to related file specifications, so that unspecified portions of file specifications will be inherited from previous file specifications.

result-spec

```
type:         character string
access:       modify
mechanism:    by descriptor, dynamic string
```

The result-spec argument is the address of the resultant file specification that LIB$FIND_FILE returns when it find a file that matches the specification.

context

```
type:         longword integer (signed)
access:       modify
mechanism:    by reference
```

The context argument is a zero or an address of an internal FAB/NAM buffer from a previous call to LIB$FIND_FILE. It is a signed longword integer containing the address of the context. LIB$FIND_FILE uses this argument to retain the context when processing multiple input files. Unspecified portions of file specifications are inherited from the last files processed because the file contexts are retained in this argument.

default-spec

type: character string
access: read only
mechanism: by descriptor

The default-spec argument is the default file specification. It is the address of a descriptor pointing to the default file specification.

related-spec

type: character string
access: read only
mechanism: by descriptor

The related-spec argument is the related file specification containing the context of the last file processed. It is the address of a descriptor pointing to the related file specification.

stv-addr

type: longword integer (signed)
access: write only
mechanism: by reference

The stv-addr argument is the RMS secondary status value from a failing RMS operation. It is a signed longword integer containing the address of a longword-length buffer to receive the RMS secondary status value (usually returned in the file access block field, FAB$L_STV).

user-flags

type: longword (unsigned)
access: read only
mechanism: by reference

The user-flags argument is the address of an unsigned longword containing the user flags.

The flag bits, their corresponding symbols, and descriptions are described below.

Bit	Symbol	Description
0	NOWILD	If set, LIB$FIND_FILE returns an error if a wildcard is input.
1	MULTIPLE	If set, this performs temporary defaulting for multiple input files and the related-spec argument is ignored. See the description of context in LIB$FILE_SCAN. Each time LIB$FIND_FILE is called with a different file specification, the specification from the previous call is automatically used as a related file specification. This allows parsing of the elements of a search list logical name such as DISK2:[SMITH] FIL1.TYP,FIL*2.TYP, and so on. Use of this feature is required to get the desired defaulting with search list logical name. LIB$FIND_FILE_END must be called between each command line in interactive use or the defaults from the previous command line will affect the current file specification.

Condition Values Returned

RMS$_NORMAL	Routine completed successfully.
SHR$_NOWILD	LIB facility code. A wildcard was present in the file specification parsed and the wildcard flag bit was set to "no wildcard."

Any condition value returned by RMS Parse and Search services, LIBGET_VM, LIBFREE_VM, or LIB$SCOPY_R_DX.

Example 2. Processing Filename(s) With Wildcards or Search Lists Using RMS $PARSE and $SEARCH

(Sheet 1 of 4)

```
 1  {                                                        PASSEARCH.PAS  }
 2
 3  {   NOTE:  Test data for this program  ------>   INTER*.DAT              }
 4
 5  {   In PASCAL you do not have to call FDL$PARSE to set up
 6      RMS structures.  You are able to set them up and
 7      initialize them directly in PASCAL.
 8
 9      This PASCAL program sets up the RMS structures and does
10      all access to a sequential file using RMS services
11      rather than using any regular PASCAL I/O.
12
13      It also checks whether a wild card is used in the file
14      spec. or a logical name search list by calling $PARSE.
15      If either is used then $SEARCH is called repeatedly until
16      there are no further file name translations.             }
17
18    [INHERIT('SYS$LIBRARY:STARLET')]
19    PROGRAM PASSEARCH(INPUT,OUTPUT);
20
21    CONST
22        STR_LEN = 255;
23        NAM$V_WILD_VER = 3;
24        NAM$V_WILD_TYPE = 4;
25        NAM$V_WILD_NAME = 5;
26        NAM$V_WILDCARD = 8;
27        NAM$V_SEARCH_LIST = 11;
28        NAM$V_WILD_DIR = 20;
29
30    TYPE
31        EMPLOYEE_STRUC  =  RECORD
32           LASTNAME :  PACKED ARRAY [1..10] OF CHAR;
33           FILLER   :  PACKED ARRAY [1..100] OF CHAR;
34           SEQNO    :  PACKED ARRAY [1..2] OF CHAR;
35        END;
36
37       BIT_VALUE    =  0..1;
38       BIT_TYPE     =  PACKED RECORD
39         BIT_ARRAY  :  [BIT(32),POS(0)] PACKED ARRAY
40                           [0..31] OF BIT_VALUE;
41       END;
42
43    VAR
44        IN_REC       :  EMPLOYEE_STRUC;
45        FAB          :  FAB$TYPE;
46        RAB          :  RAB$TYPE;
47        NAM          :  NAM$TYPE;
48        RET_STATUS   :  INTEGER;
49        MOREFILES    :  INTEGER := 1;
50        FNM_STR      :  PACKED ARRAY [1..255] OF CHAR;
51        RES_STR      :  PACKED ARRAY [1..255] OF CHAR;
52        EXP_STR      :  PACKED ARRAY [1..255] OF CHAR;
53        BIT_SET      :  UNSIGNED := 0;
54        INDEX        :  0..255;
55        PARSE_CHK    :  ARRAY [1..6] OF INTEGER :=
56                              (NAM$V_WILD_VER,NAM$V_WILD_TYPE,
57                               NAM$V_WILD_NAME,NAM$V_WILDCARD,
58                               NAM$V_SEARCH_LIST,NAM$V_WILD_DIR);
59        BIT_SUM      :  UNSIGNED := 0;
60
```

```
61      PROCEDURE LIB$STOP (
62              %IMMED cond_value : INTEGER); EXTERN;
63
64      PROCEDURE SEARCHFILE;
65      BEGIN
66
67         RET_STATUS := $SEARCH(FAB);
68         IF (RET_STATUS = RMS$_NMF) THEN MOREFILES := 0;
69
70         FOR INDEX := 1 TO NAM.NAM$B_RSL DO
71              WRITE (RES_STR[INDEX]);
72         WRITELN;
73      END;  { searchfile procedure }
74
75      PROCEDURE PARSEFILE;
76
77      BEGIN
78
79              FAB                 := ZERO;
80
81              FAB.FAB$B_BID   := FAB$C_BID;
82              FAB.FAB$B_BLN   := FAB$C_BLN;
83              FAB.FAB$L_FNA   := IADDRESS(FNM_STR);
84              FAB.FAB$B_FNS   := STR_LEN;
85              FAB.FAB$B_ORG   := FAB$C_SEQ;
86              FAB.FAB$B_FAC   := FAB$M_GET;
87              FAB.FAB$L_NAM   := IADDRESS(NAM);
88              FAB.FAB$L_FOP   := FAB$M_NAM;
89
90              NAM                 := ZERO;
91
92              NAM.NAM$B_BID    := NAM$C_BID;
93              NAM.NAM$B_BLN    := NAM$C_BLN;
94              NAM.NAM$L_RSA    := IADDRESS(RES_STR);
95              NAM.NAM$B_RSS    := STR_LEN;
96              NAM.NAM$L_ESA    := IADDRESS(EXP_STR);
97              NAM.NAM$B_ESS    := STR_LEN;
98
99              RAB             := ZERO;
100
101             RAB.RAB$B_BID := RAB$C_BID;
102             RAB.RAB$B_BLN := RAB$C_BLN;
103             RAB.RAB$L_FAB := IADDRESS(FAB);
104             RAB.RAB$B_RAC := RAB$C_SEQ;
105
106        RET_STATUS    := $PARSE(FAB);
107
108     FOR INDEX := 1 TO 6 DO
109     BEGIN
110        BIT_SET       := NAM.NAM$L_FNB::BIT_TYPE.BIT_ARRAY
111                                        [PARSE_CHK[INDEX]];
112        BIT_SUM       := BIT_SUM + BIT_SET;
113     END;
114
115     IF (BIT_SUM > 0) THEN  SEARCHFILE;
116
117     END;  { parsefile procedure }
118
119     PROCEDURE OPENFILE;
120     BEGIN
```

Example 2 (Sheet 3 of 4)

```
121
122         RET_STATUS := $OPEN(FAB);
123
124         IF ODD(RET_STATUS) THEN
125              RET_STATUS := $CONNECT(RAB)
126              ELSE
127              LIB$STOP(RET_STATUS);
128
129     END;  { useropen procedure }
130
131     PROCEDURE GETFILE;
132
133     BEGIN
134
135             RAB.RAB$L_UBF := IADDRESS(IN_REC);
136             RAB.RAB$W_USZ := SIZE(IN_REC);
137
138             RET_STATUS    := $GET(RAB);
139
140             WHILE ODD(RET_STATUS) DO
141             BEGIN
142
143               WRITELN (IN_REC.LASTNAME,'   ', IN_REC.SEQNO);
144
145               RET_STATUS := $GET(RAB);
146             END;
147
148             IF (RET_STATUS <> RMS$_EOF) THEN
149                   LIB$STOP (RET_STATUS);
150
151     END;  { getfile }
152
153     BEGIN  { MAIN }
154
155             WRITE ('Enter filename: ');
156             READLN  (FNM_STR);
157
158             PARSEFILE;
159             OPENFILE;
160             GETFILE;
161             $CLOSE(FAB);
162
163     WHILE (MOREFILES > 0) DO
164     BEGIN
165
166             SEARCHFILE;
167
168             IF (MOREFILES > 0) THEN
169             BEGIN
170
171                     OPENFILE;
172                     GETFILE;
173                     $CLOSE(FAB);
174             END;
175
176     END;
177
178     END.  { MAIN }
```

Example 2 (Sheet 4 of 4)

```
$ RUN PASSEARCH
Enter filename: INTER%.DAT
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER1.DAT;1
RAKOS
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER2.DAT;1
ASHE
RAKOS
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER3.DAT;1
ASHE
RAKOS
TODD
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER4.DAT;1
ASHE
JONES
RAKOS
TODD
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER5.DAT;1
ASHE
JONES
RAKOS
TODD
VAIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER6.DAT;1
ASHE
BUSH
JONES
RAKOS
TODD
VAIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER7.DAT;1
ASHE
BUSH
EVANS
JONES
RAK.S
TO.D
VAIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER8.DAT;1
ASHE
BUSH
EVANS
JONES
RAKOS
SACK
TODD
VAIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER9.DAT;1
ASHE
BUSH
EVANS
JONES
MAYO
RAKOS
SACK
TODD
VAIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER%.DAT;
```

MODULE 15
PROCESS QUOTAS AND LIMITS

Source

Guide to VAX/VMS File Applications — Chapter 1 (Sections 1.6-1.7)

PROCESS AND SYSTEM RESOURCES FOR FILE APPLICATIONS

To use RMS files efficiently, your application requires certain resources that are defined for the system or each process. Specific resources and quotas may need to be adjusted for the process running a file application. Coordinate process and system requirements with your system manager during the application design (or redesign) procedure before implementing the RMS options that require the additional resources for the application. In some cases, the system manager may want to order additional memory or disk drives to ensure that sufficient system resources are available.

Memory Requirements

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

RMS maintains not only the specified buffers when a file is opened (or created), but also control structures that are charged against process memory use. Memory use generally increases with the number of files to be processed at the same time. The amount of memory needed for I/O buffers can vary greatly for each file; the amount of memory needed for control structures is fairly constant for each file.

The memory use (working set) of a process is governed by three SYSGEN parameters. (Process quotas)

1. Working set default (WSDEFAULT) specifies the initial size of the working set, in pages (512 bytes).

2. Working set quota (WSQUOTA) specifies the maximum size, in pages, that the working set can grow to (unless physical memory pages are available and a larger working set extent value is specified).

3. Working set extent (WSEXTENT) specifies the maximum size, in pages, that the working set can grow to, including the use of free pages of physical memory.

These values can ensure that the process will have sufficient memory to perform the application with a minimum of paging.

Process Record-Locking Quota

When an application will access a shared file for which record modifications or additions are allowed, the process enqueue quota should be examined. The need to increase the process enqueue

quota (ENQLM) varies with the number of records that may be simultaneously locked, multiplied by the number of open files.

The enqueue quota (ENQLM) limits the number of locks a process (and its subprocesses) can own. VAX RMS uses the Lock Management Facility to synchronize shared file access, global buffers, and record locks. Because VAX RMS takes out one lock for every shared file, local buffer, global buffer section, and outstanding record lock, users who expect to perform large amounts of VAX RMS file sharing should have ENQLM set to a large value.

If your process performs extensive VAX RMS file sharing without a sufficient enqueue quota, you could receive the SS$_EXENQLM error message.

If your system performs extensive VAX RMS file sharing and the value of the LOCKIDTBL system parameter is too low, you could receive the SS$_NOLOCKID error message. Your system manager would need to increase both the value of LOCKIDTBL and the value of RESHASHTBL.

Estimate the number of locks per process per file as one per file, plus the multibuffer count for that file, plus the number of records locked (which is usually one unless manual locking is enabled). Use the DCL command SHOW RMS_DEFAULT to display the default multibuffer counts.

Other Limits

Other limits that should be examined are:

1. Process Open File Limit

   The number of files that a process will have open simultaneously is governed by the open file limit (FILLM).

2. Process Asynchronous I/O Limit

   If asynchronous record I/O will occur, the following limits should be examined.

   - Asynchronous system trap limit (ASTLM)
   - Buffered I/O limit (BIOLM)
   - Direct I/O limit (DIOLM)

The values suggested to the system manager for these and other limits are provided in the table below. For a complete description of these limits, see the Guide to VAX/VMS System Management and Daily Operations. Managers reference manual

Process Resource Limits, Suggested Values, Types, and Descriptions

Limit	Value	Type*	Description
ASTLM	24	N	AST queue limit
BIOLM	18	N	Buffered I/O count limit
BYTLM	8192	P	I/O byte count limit
CPU	0	D	CPU time limit (0 = no limit)
DIOLM	18	N	Direct I/O count limit
ENQLM	30	P	Enqueue quota
FILLM	20	P	Open file limit
JTQUOTA	1024	P	Initial byte quota for job-wide logical name table
MAXACCTJOBS	0	S	Maximum active processes for a single ~~username~~ account (0 = no limit)
MAXDETACH	0	S	Maximum detached processes for a single username (0 = no limit)
MAXJOBS	0	S	Maximum active processes for a single username (0 = no limit)
PGFLQUO	12800	P	Paging file limit
PRCLM	2	P	Subprocess creation limit
SHRFILM	0	P	Maximum number open shared files (0 = no limit)
TQELM	10	P	Timer queue entry limit

Process Resource Limits, Suggested Values, Types, and Descriptions (Cont.)

Limit	Value	Type*	Description
WSDEF	300	N	Default working set size
WSEXTENT	700	N	Working set extent
WSQUO	350	N	Working set quota

* D = Deductible
N = Nondeductible
P = Pooled
S = System-wide

Source: VMS V4.0 System Management and Daily Operations (September 1984)

In addition to process requirements, a shared file may want to use the capabilities of global buffers to avoid needless I/O when the desired block is already in memory. The memory use of global buffers is governed by the following SYSGEN parameters.

1. The number of RMS global buffers (RMS_GBLBUFQUO) specifies the maximum number of RMS global buffers in use on a system simultaneously, regardless of the number of users or files.

2. The number of global sections (GBLSECTIONS) specifies the maximum number of global sections in use simultaneously on the system.

3. The number of global page table entries (GBLPAGES) specifies the number of global page table entries in use simultaneously on the system.

4. The number of system-wide pages allowed for global page-file sections or scratch global sections (GBLPAGFIL) specifies the number of system-wide pages allowed for global page-file sections, or scratch global sections, in use simultaneously on the system.

When DCL opens a file (a process-permanent file), RMS places internal structures for this file in a special portion of P1 space called the Process I/O Segment. The size of this segment is determined by the SYSGEN parameter PIOPAGES and cannot be expanded dynamically. If DCL tries to open a file and there is not enough space in the Process I/O Segment for the internal structures, you will receive an error message and the file will not be opened.

For a complete description of these parameters, see the description of the System Generation Utility (SYSGEN) in the VAX/VMS Utilities Reference Volume.

Also see all RMS_~ & ACP_~ parameters

MODULE 16
RMS Utilities

Major Topics

PART 6. ANALYZE/RMS/INTERACTIVE

- INTERACTIVE commands
- Sample interactive sessions
  - Exploring indexed file structures
  - Tracking a record from key value to data record

Source

Guide to VAX/VMS File Applications — Chapter # 1ø

PART 6. ANALYZE/RMS/INTERACTIVE

ANALYZE/RMS_FILE Interactive Commands

Command	Function
AGAIN	Displays the current structure again.
DOWN [branch] DOWN AREA DOWN KEY DOWN DATA DOWN BYTES DOWN RRV DOWN SIDRS	Move the structure pointer down to the next level. If the current node has more than one branch, the branch keyword must be specified. If a branch keyword is required but not specified, the utility will display a list of possibilities to prompt you. You can also display the list by specifying "DOWN ?."
DUMP n	Displays a hexadecimal dump of the specified block.
EXIT	End the interactive session.
FIRST	Moves the structure pointer to the first structure on the current level. The structure is displayed. For example, if you are examining data buckets and want to examine the first bucket, this command will put you there and display the first bucket's header.
HELP [keyword...]	Displays help messages about the interactive commands.
NEXT	Moves the structure pointer to the next structure on the current level. The structure is displayed. Pressing the RETURN key is equivalent to a NEXT command.
REST	Moves the structure pointer along the rest of the structures on the current level, and each is displayed in turn.
TOP	Moves the structure pointer up to the file header. The file header is displayed.
UP	Moves the structure pointer up to the next level. The structure at that level is displayed.

New ANALYZE/RMS_FILE Interactive Commands as of VAX/VMS Version 4.4

Command	Function
BACK	Moves the structure pointer to the previous node if one exists within the current level, and displays that node. The number of structures that the pointer is to be moved can also be specified by using the optional parameter BACK n, where n is an integer.
NEXT	As of 4.4, will accept the optional parameter n (NEXT n) to specify the number of structures that the pointer is to be moved forward.
POSITION/BUCKET	Positions the pointer to a specific bucket of the file. This command can be used to bypass step-by-step positioning, and also to position the pointer at a bucket that would otherwise be inaccessible due to structural errors in the file.
POSITION/RECORD	Positions the pointer at a specific record in the current bucket, allowing subsequent structures to be accessed easier.

```
                                                ANALYZE> DOWN BYTES
ANALYZE> DOWN KEY                                  7  6  5  4  3  2  1  0          01234567
BUCKET HEADER (VBN 4)                            ------------------------          --------
       Check Character: %X'00'                    31 30 30 30 30 30 00 49|  0000  |I.000001|
       Area Number: 0                             20 4C 41 54 49 47 49 44|  0008  |DIGITAL |
       VBN Sample: 4                              4E 45 4D 50 49 55 51 45|  0010  |EQUIPMEN|
       Free Space Offset: %X'0104'                52 4F 50 52 4F 43 20 54|  0018  |T CORPOR|
       Free Record ID Range: 4- 255               31 31 20 4E 4F 49 54 41|  0020  |ATION 11|
       Next Bucket VBN: 4                         42 20 54 49 50 53 20 30:  0028  |0 SPIT B|
       Level: 0                                   41 4F 52 20 4B 4F 4F 52|  0030  |ROOK ROA|
       Bucket Header Flags:                       41 55 48 53 41 4E 20 44|  0038  |D NASHUA|
              (0) BKT$V_LASTBKT     1             33 30 48 4E 20 20 20 20|  0040  |    NH03|
              (1) BKT$V_ROOTBKT     0                          31 36 30|  0048  |061     |
```

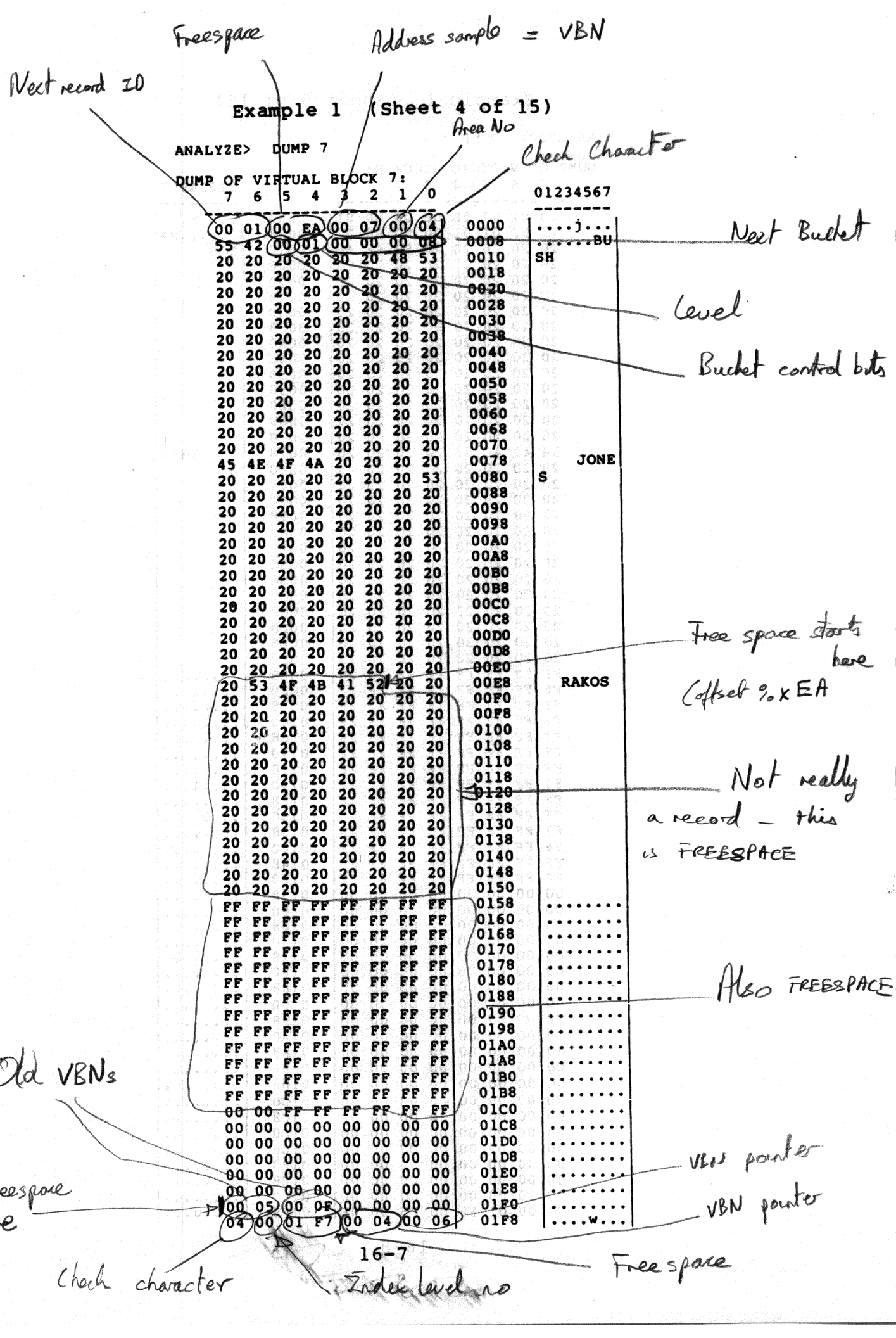
Example 1. Exploring Indexed File Structure (Sheet 1 of 15)

```
$ ANALYZE/RMS/INTERACTIVE  INTER11.DAT

FILE HEADER

   File Spec: DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INTER11.DAT;1
   File ID: (2068,41,0)
   Owner UIC: [010,007]
   Protection:  System: RWED, Owner: RWED, Group: RE, World:
   Creation Date:     1-JAN-1986 20:55:59.74
   Revision Date:     3-FEB-1986 20:22:59.60, Number: 5
   Expiration Date: none specified
   Backup Date:      none posted
   Contiguity Options:  contiguous-best-try
   Performance Options: none
   Reliability Options: none
   Journaling Enabled:  none

ANALYZE>  DOWN

RMS FILE ATTRIBUTES

   File Organization: indexed
   Record Format: fixed
   Record Attributes:    carriage-return
   Maximum Record Size: 112
   Longest Record: 112
   Blocks Allocated: 16, Default Extend Size: 1
   Bucket Size: 1
   Global Buffer Count: 0

ANALYZE>  DOWN

FIXED PROLOG

   Number of Areas: 3, VBN of First Descriptor: 3
   Prolog Version: 3

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, AREAS          Area descriptors
%ANLRMS-I-DOWNPATH, KEYS           Key descriptors

ANALYZE>  DOWN KEYS
```

```
KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

   Next Key Descriptor VBN: 2, Offset: %X'0000'
   Index Area: 1, Level 1 Index Area: 1, Data Area: 0
   Root Level: 2
   Index Bucket Size: 1, Data Bucket Size: 1
   Root VBN: 9
   Key Flags:
        (0)  KEY$V_DUPKEYS    0
        (3)  KEY$V_IDX_COMPR  0
        (4)  KEY$V_INITIDX    0
        (6)  KEY$V_KEY_COMPR  0
        (7)  KEY$V_REC_COMPR  0
   Key Segments: 1
   Key Size: 110
   Minimum Record Size: 110
   Index Fill Quantity: 512, Data Fill Quantity: 512
   Segment Positions:       0
   Segment Sizes:         110
   Data Type: string
   Name: "LAST_NAME"
   First Data Bucket VBN: 4

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, INDEX          Root index bucket
%ANLRMS-I-DOWNPATH, DATA           Data buckets

ANALYZE>  DOWN INDEX

BUCKET HEADER (VBN 9)

   Check Character: %X'01'
   Key of Reference: 0
   VBN Sample: 9
   Free Space Offset: %X'00EA'
   Free Record ID: 1
   Next Bucket VBN: 9
   Level: 2
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT    1
          (1)  BKT$V_ROOTBKT    1
   Bucket Pointer Size: 2

   VBN Free Space Offset: %X'01F7'

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, RECORDS        Index records
```

Example 1 (Sheet 3 of 15)

```
ANALYZE>  DOWN RECORDS

   INDEX RECORD (VBN 9, offset %X'000E')

          2-Byte Bucket Pointer: 7
          Key:
                 7  6  5  4  3  2  1  0          01234567
               ------------------------          --------
                20 20 20 53 45 4E 4F 4A|  0000  |JONES   |
                20 20 20 20 20 20 20 20|  0008  |        |
                20 20 20 20 20 20 20 20|  0010  |        |
                20 20 20 20 20 20 20 20|  0018  |        |
                20 20 20 20 20 20 20 20|  0020  |        |
                20 20 20 20 20 20 20 20|  0028  |        |
                20 20 20 20 20 20 20 20|  0030  |        |
                20 20 20 20 20 20 20 20|  0038  |        |
                20 20 20 20 20 20 20 20|  0040  |        |
                20 20 20 20 20 20 20 20|  0048  |        |
                20 20 20 20 20 20 20 20|  0050  |        |
                20 20 20 20 20 20 20 20|  0058  |        |
                20 20 20 20 20 20 20 20|  0060  |        |
                20 20 20 20 20 20|  0068  |        |

ANALYZE>  NEXT

   INDEX RECORD (VBN 9, offset %X'007C')

          2-Byte Bucket Pointer: 8
          Key:
                 7  6  5  4  3  2  1  0          01234567
               ------------------------          --------
                FF FF FF FF FF FF FF FF|  0000  |........|
                FF FF FF FF FF FF FF FF|  0008  |........|
                FF FF FF FF FF FF FF FF|  0010  |........|
                FF FF FF FF FF FF FF FF|  0018  |........|
                FF FF FF FF FF FF FF FF|  0020  |........|
                FF FF FF FF FF FF FF FF|  0028  |........|
                FF FF FF FF FF FF FF FF|  0030  |........|
                FF FF FF FF FF FF FF FF|  0038  |........|
                FF FF FF FF FF FF FF FF|  0040  |........|
                FF FF FF FF FF FF FF FF|  0048  |........|
                FF FF FF FF FF FF FF FF|  0050  |........|
                FF FF FF FF FF FF FF FF|  0058  |........|
                FF FF FF FF FF FF FF FF|  0060  |........|
                FF FF FF FF FF FF|  0068  |......  |

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, DEEPER          Index or data buckets

ANALYZE>  DOWN DEEPER

BUCKET HEADER (VBN 8)
```


Freespace
Address sample = VBN
Next record ID
Example 1 (Sheet 4 of 15)
Area No
Check Character
ANALYZE> DUMP 7
DUMP OF VIRTUAL BLOCK 7:
7 6 5 4 3 2 1 0
01234567
00 01 00 EA 00 07 00 04 0000 ....j...
55 42 00 01 00 00 00 08 0008 .....BU
20 20 20 20 20 20 48 53 0010 SH
0018
0020
0028
0030
0038
0040
0048
0050
0058
0060
0068
0070
45 4E 4F 4A 20 20 20 20 0078 JONE
20 20 20 20 20 20 20 53 0080 S
0088
0090
0098
00A0
00A8
00B0
00B8
00C0
00C8
00D0
00D8
00E0
20 53 4F 4B 41 52 20 20 00E8 RAKOS
00F0
00F8
0100
0108
0110
0118
0120
0128
0130
0138
0140
0148
0150
FF FF FF FF FF FF FF FF 0158 ....
FF FF FF FF FF FF FF FF 0160 ....
FF FF FF FF FF FF FF FF 0168 ....
FF FF FF FF FF FF FF FF 0170 ....
FF FF FF FF FF FF FF FF 0178 ....
FF FF FF FF FF FF FF FF 0180 ....
FF FF FF FF FF FF FF FF 0188 ....
FF FF FF FF FF FF FF FF 0190 ....
FF FF FF FF FF FF FF FF 0198 ....
FF FF FF FF FF FF FF FF 01A0 ....
FF FF FF FF FF FF FF FF 01A8 ....
FF FF FF FF FF FF FF FF 01B0 ....
FF FF FF FF FF FF FF FF 01B8 ....
00 00 FF FF FF FF FF FF 01C0 ....
00 00 00 00 00 00 00 00 01C8 ....
00 00 00 00 00 00 00 00 01D0 ....
00 00 00 00 00 00 00 00 01D8 ....
00 00 00 00 00 00 00 00 01E0 ....
00 00 00 00 00 00 00 00 01E8 ....
00 05 00 0E 00 00 00 00 01F0 ....
04 00 01 F7 00 04 00 06 01F8 ....w...
Next Bucket
Level
Bucket control bits
Free space starts here
Offset % x EA
Not really a record — this is FREESPACE
Also FREESPACE
Old VBNs
VBN pointer
Reverse freespace starts here
VBN pointer
Check character
Index level no
Freespace
16-7

Example 1 (Sheet 5 of 15)

```
ANALYZE>  DUMP 8

DUMP OF VIRTUAL BLOCK 8:
     7  6  5  4  3  2  1  0                 01234567
    ------------------------                --------
    00 01 01 58 00 08 00 01|   0000   |....X...|
    41 52 01 01 00 00 00 07|   0008   |......RA|
    20 20 20 20 20 53 4F 4B|   0010   |KOS     |
    20 20 20 20 20 20 20 20|   0018   |        |
    20 20 20 20 20 20 20 20|   0020   |        |
    20 20 20 20 20 20 20 20|   0028   |        |
    20 20 20 20 20 20 20 20|   0030   |        |
    20 20 20 20 20 20 20 20|   0038   |        |
    20 20 20 20 20 20 20 20|   0040   |        |
    20 20 20 20 20 20 20 20|   0048   |        |
    20 20 20 20 20 20 20 20|   0050   |        |
    20 20 20 20 20 20 20 20|   0058   |        |
    20 20 20 20 20 20 20 20|   0060   |        |
    20 20 20 20 20 20 20 20|   0068   |        |
    20 20 20 20 20 20 20 20|   0070   |        |
    54 49 4D 53 20 20 20 20|   0078   |    SMIT|
    20 20 20 20 20 20 20 48|   0080   |H       |
    20 20 20 20 20 20 20 20|   0088   |        |
    20 20 20 20 20 20 20 20|   0090   |        |
    20 20 20 20 20 20 20 20|   0098   |        |
    20 20 20 20 20 20 20 20|   00A0   |        |
    20 20 20 20 20 20 20 20|   00A8   |        |
    20 20 20 20 20 20 20 20|   00B0   |        |
    20 20 20 20 20 20 20 20|   00B8   |        |
    20 20 20 20 20 20 20 20|   00C0   |        |
    20 20 20 20 20 20 20 20|   00C8   |        |
    20 20 20 20 20 20 20 20|   00D0   |        |
    20 20 20 20 20 20 20 20|   00D8   |        |
    20 20 20 20 20 20 20 20|   00E0   |        |
    FF FF FF FF FF FF 20 20|   00E8   |  ......|
    FF FF FF FF FF FF FF FF|   00F0   |........|
    FF FF FF FF FF FF FF FF|   00F8   |........|
    FF FF FF FF FF FF FF FF|   0100   |........|
    FF FF FF FF FF FF FF FF|   0108   |........|
    FF FF FF FF FF FF FF FF|   0110   |........|
    FF FF FF FF FF FF FF FF|   0118   |........|
    FF FF FF FF FF FF FF FF|   0120   |........|
    FF FF FF FF FF FF FF FF|   0128   |........|
    FF FF FF FF FF FF FF FF|   0130   |........|
    FF FF FF FF FF FF FF FF|   0138   |........|
    FF FF FF FF FF FF FF FF|   0140   |........|
    FF FF FF FF FF FF FF FF|   0148   |........|
    FF FF FF FF FF FF FF FF|   0150   |........|
    00 00 00 00 00 00 00 00|   0158   |........|
    00 00 00 00 00 00 00 00|   0160   |........|
    00 00 00 00 00 00 00 00|   0168   |........|
    00 00 00 00 00 00 00 00|   0170   |........|
    00 00 00 00 00 00 00 00|   0178   |........|
    00 00 00 00 00 00 00 00|   0180   |........|
    00 00 00 00 00 00 00 00|   0188   |........|
    00 00 00 00 00 00 00 00|   0190   |........|
    00 00 00 00 00 00 00 00|   0198   |........|
    00 00 00 00 00 00 00 00|   01A0   |........|
    00 00 00 00 00 00 00 00|   01A8   |........|
    00 00 00 00 00 00 00 00|   01B0   |........|
    00 00 00 00 00 00 00 00|   01B8   |........|
    00 00 00 00 00 00 00 00|   01C0   |........|
    00 00 00 00 00 00 00 00|   01C8   |........|
    00 00 00 00 00 00 00 00|   01D0   |........|
    00 00 00 00 00 00 00 00|   01D8   |........|
    00 00 00 00 00 00 00 00|   01E0   |........|
    00 00 00 00 00 00 00 00|   01E8   |........|
    00 10 00 00 00 00 00 00|   01F0   |........|
    01 00 01 F5 00 05 00 0F|   01F8   |....u...|
```

```
ANALYZE>  UP

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')


ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, INDEX          Root index bucket
%ANLRMS-I-DOWNPATH, DATA           Data buckets

ANALYZE>  DOWN DATA

BUCKET HEADER (VBN 4)

   Check Character: %X'09'
   Key of Reference: 0
   VBN Sample: 4
   Free Space Offset: %X'011B'
   Free Record ID: 6
   Next Bucket VBN: 6
   Level: 0
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT    0

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, RECORDS        Primary data records

ANALYZE>  DOWN RECORDS

   PRIMARY DATA RECORD (VBN 4, offset %X'000E')

          Record Control Flags:
                 (2)  IRC$V_DELETED     0
                 (3)  IRC$V_RRV         0
                 (4)  IRC$V_NOPTRSZ     0
                 (5)  IRC$V_RU_DELETE   0
                 (6)  IRC$V_RU_UPDATE   0
          Record ID: 2
          RRV ID: 2, 4-Byte Bucket Pointer: 4
          Key:
                    7  6  5  4  3  2  1  0          01234567
                   -----------------------         --------
                   20 20 20 20 45 48 53 41 | 0000  |ASHE    |
                   20 20 20 20 20 20 20 20 | 0008  |        |
                   20 20 20 20 20 20 20 20 | 0010  |        |
                   20 20 20 20 20 20 20 20 | 0018  |        |
                   20 20 20 20 20 20 20 20 | 0020  |        |
                   20 20 20 20 20 20 20 20 | 0028  |        |
                   20 20 20 20 20 20 20 20 | 0030  |        |
                   20 20 20 20 20 20 20 20 | 0038  |        |
                   20 20 20 20 20 20 20 20 | 0040  |        |
                   20 20 20 20 20 20 20 20 | 0048  |        |
                   20 20 20 20 20 20 20 20 | 0050  |        |
                   20 20 20 20 20 20 20 20 | 0058  |        |
                   20 20 20 20 20 20 20 20 | 0060  |        |
                      20 20 20 20 20 20 20 | 0068  |        |
```

Example 1 (Sheet 7 of 15)

```
ANALYZE>  NEXT

   PRIMARY DATA RECORD (VBN 4, offset %X'0087')

          Record Control Flags:
                  (2)  IRC$V_DELETED     0
                  (3)  IRC$V_RRV         0
                  (4)  IRC$V_NOPTRSZ     0
                  (5)  IRC$V_RU_DELETE   0
                  (6)  IRC$V_RU_UPDATE   0
          Record ID: 5
          RRV ID: 5, 4-Byte Bucket Pointer: 4
          Key:
                   7  6  5  4  3  2  1  0          01234567
                 ------------------------          --------
                  20 20 20 20 48 53 55 42|  0000  |BUSH    |
                  20 20 20 20 20 20 20 20|  0008  |        |
                  20 20 20 20 20 20 20 20|  0010  |        |
                  20 20 20 20 20 20 20 20|  0018  |        |
                  20 20 20 20 20 20 20 20|  0020  |        |
                  20 20 20 20 20 20 20 20|  0028  |        |
                  20 20 20 20 20 20 20 20|  0030  |        |
                  20 20 20 20 20 20 20 20|  0038  |        |
                  20 20 20 20 20 20 20 20|  0040  |        |
                  20 20 20 20 20 20 20 20|  0048  |        |
                  20 20 20 20 20 20 20 20|  0050  |        |
                  20 20 20 20 20 20 20 20|  0058  |        |
                  20 20 20 20 20 20 20 20|  0060  |        |
                        20 20 20 20 20 20|  0068  |        |

ANALYZE>  NEXT

***  VBN 4:  Key and/or data bytes do not fit in primary data record.
   PRIMARY DATA RECORD (VBN 4, offset %X'0100')

          Record Control Flags:
                  (2)  IRC$V_DELETED     0
                  (3)  IRC$V_RRV         1
                  (4)  IRC$V_NOPTRSZ     0
                  (5)  IRC$V_RU_DELETE   0
                  (6)  IRC$V_RU_UPDATE   0
          Record ID: 1
          RRV ID: 1, 4-Byte Bucket Pointer: 5
```

```
ANALYZE>  NEXT

***  VBN 4:  Key and/or data bytes do not fit in primary data record.
   PRIMARY DATA RECORD (VBN 4, offset %X'0109')

          Record Control Flags:
                  (2)  IRC$V_DELETED     0
                  (3)  IRC$V_RRV         1
                  (4)  IRC$V_NOPTRSZ     0
                  (5)  IRC$V_RU_DELETE   0
                  (6)  IRC$V_RU_UPDATE   0
          Record ID: 3
          RRV ID: 1, 4-Byte Bucket Pointer: 16


ANALYZE>  DOWN NEXT or <RET>

***  VBN 4:  Key and/or data bytes do not fit in primary data record.
   PRIMARY DATA RECORD (VBN 4, offset %X'0112')

          Record Control Flags:
                  (2)  IRC$V_DELETED     0
                  (3)  IRC$V_RRV         1
                  (4)  IRC$V_NOPTRSZ     0
                  (5)  IRC$V_RU_DELETE   0
                  (6)  IRC$V_RU_UPDATE   0
          Record ID: 4
          RRV ID: 2, 4-Byte Bucket Pointer: 6

ANALYZE>  NEXT

%ANLRMS-I-NONEXT, There is no structure following the current one.

ANALYZE>  UP

BUCKET HEADER (VBN 4)

   Check Character: %X'09'
   Key of Reference: 0
   VBN Sample: 4
   Free Space Offset: %X'011B'
   Free Record ID: 6
   Next Bucket VBN: 6
   Level: 0
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT     0
```

pointer size – not significant – always = 0 for Prolog 3

(RMS journalling – ignore)

```
ANALYZE>  REST

BUCKET HEADER (VBN 6)

   Check Character: %X'01'
   Key of Reference: 0
   VBN Sample: 6
   Free Space Offset: %X'0100'
   Free Record ID: 3
   Next Bucket VBN: 5
   Level: 0
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT    0


BUCKET HEADER (VBN 5)

   Check Character: %X'04'
   Key of Reference: 0
   VBN Sample: 5
   Free Space Offset: %X'0112'
   Free Record ID: 6
   Next Bucket VBN: 15
   Level: 0
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT    0


BUCKET HEADER (VBN 15)

   Check Character: %X'03'
   Key of Reference: 0
   VBN Sample: 15
   Free Space Offset: %X'0109'
   Free Record ID: 6
   Next Bucket VBN: 16
   Level: 0
   Bucket Header Flags:
   (0)  BKT$V_LASTBKT    0


BUCKET HEADER (VBN 16)

   Check Character: %X'01'
   Key of Reference: 0
   VBN Sample: 16
   Free Space Offset: %X'0179'
   Free Record ID: 4
   Next Bucket VBN: 4
   Level: 0
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT    1

%ANLRMS-I-RESTDONE, All structures at this level have been displayed.
```

Free space pointer

Area no.

Next record no

Block no

Check char

Example 1 (Sheet 10 of 15)

```
ANALYZE>  DUMP 4

DUMP OF VIRTUAL BLOCK 4:
   7  6  5  4  3  2  1  0          01234567
  ------------------------          --------
  00 06 01 1B 00 04 00 09 |  0000 |........|
  02 02 00 00 00 00 00 06 |  0008 |........|
  41 00 00 00 04 00 02 00 |  0010 |.......A|
  20 20 20 20 20 45 48 53 |  0018 |SHE     |
  20 20 20 20 20 20 20 20 |  0020 |        |
  20 20 20 20 20 20 20 20 |  0028 |        |
  20 20 20 20 20 20 20 20 |  0030 |        |
  20 20 20 20 20 20 20 20 |  0038 |        |
  20 20 20 20 20 20 20 20 |  0040 |        |
  20 20 20 20 20 20 20 20 |  0048 |        |
  20 20 20 20 20 20 20 20 |  0050 |        |
  20 20 20 20 20 20 20 20 |  0058 |        |
  20 20 20 20 20 20 20 20 |  0060 |        |
  20 20 20 20 20 20 20 20 |  0068 |        |
  20 20 20 20 20 20 20 20 |  0070 |        |
  20 20 20 20 20 20 20 20 |  0078 |        |
  02 00 02 20 20 20 20 20 |  0080 |     ...|
  00 00 00 04 00 05 00 05 |  0088 |........|
  20 20 20 20 48 53 55 42 |  0090 |BUSH    |
  20 20 20 20 20 20 20 20 |  0098 |        |
  20 20 20 20 20 20 20 20 |  00A0 |        |
  20 20 20 20 20 20 20 20 |  00A8 |        |
  20 20 20 20 20 20 20 20 |  00B0 |        |
  20 20 20 20 20 20 20 20 |  00B8 |        |
  20 20 20 20 20 20 20 20 |  00C0 |        |
  20 20 20 20 20 20 20 20 |  00C8 |        |
  20 20 20 20 20 20 20 20 |  00D0 |        |
  20 20 20 20 20 20 20 20 |  00D8 |        |
  20 20 20 20 20 20 20 20 |  00E0 |        |
  20 20 20 20 20 20 20 20 |  00E8 |      .|
  20 20 20 20 20 20 20 20 |  00F0 |        |
  00 06 20 20 20 20 20 20 |  00F8 |      ..|
  00 00 05 00 01 00 01 0A |  0100 |........|
  00 10 00 01 00 03 0A 00 |  0108 |........|
  06 00 02 00 04 0A 00 00 |  0110 |........|
  20 20 20 20 20 00 00 00 |  0118 |...     |
  20 20 20 20 20 20 20 20 |  0120 |        |
  20 20 20 20 20 20 20 20 |  0128 |        |
  20 20 20 20 20 20 20 20 |  0130 |        |
  20 20 20 20 20 20 20 20 |  0138 |        |
  20 20 20 20 20 20 20 20 |  0140 |        |
  20 20 20 20 20 20 20 20 |  0148 |        |
  20 20 20 20 20 20 20 20 |  0150 |        |
  20 20 20 20 20 20 20 20 |  0158 |        |
  20 20 20 20 20 20 20 20 |  0160 |        |
  20 20 20 20 20 20 20 20 |  0168 |        |
  04 20 20 20 20 20 20 20 |  0170 |       .|
  00 05 00 01 00 01 0A 00 |  0178 |........|
  05 00 02 00 03 0A 00 00 |  0180 |........|
  00 02 00 04 0A 00 00 00 |  0188 |........|
  20 20 20 20 00 00 00 06 |  0190 |....    |
  20 20 20 20 20 20 20 20 |  0198 |        |
  20 20 20 20 20 20 20 20 |  01A0 |        |
  20 20 20 20 20 20 20 20 |  01A8 |        |
  20 20 20 20 20 20 20 20 |  01B0 |        |
  20 20 20 20 20 20 20 20 |  01B8 |        |
  20 20 20 20 20 20 20 20 |  01C0 |        |
  20 20 20 20 20 20 20 20 |  01C8 |        |
  20 20 20 20 20 20 20 20 |  01D0 |        |
  20 20 20 20 20 20 20 20 |  01D8 |        |
  20 20 20 20 20 20 20 20 |  01E0 |        |
  20 20 20 20 20 20 20 20 |  01E8 |        |
  05 00 01 00 01 0A 00 03 |  01F0 |........|
  09 00 00 00 00 00 00 00 |  01F8 |........|
```

Next Bucket

level

Bucket control bits

Start of RRV — always %x0A

Start of free space

Check char

Example 1 (Sheet 11 of 15)

```
ANALYZE>  UP [UPs...to Key Descriptor 0 or TOP and DOWNs...]

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')


ANALYZE>  NEXT

KEY DESCRIPTOR #1 (VBN 2, offset %X'0000')

   Index Area: 2, Level 1 Index Area: 2, Data Area: 2
   Root Level: 1
   Index Bucket Size: 1, Data Bucket Size: 1
   Root VBN: 14
   Key Flags:
           (0)  KEY$V_DUPKEYS     0
           (1)  KEY$V_CHGKEYS     0
           (2)  KEY$V_NULKEYS     0
           (3)  KEY$V_IDX_COMPR   0
           (4)  KEY$V_INITIDX     0
           (6)  KEY$V_KEY_COMPR   0
   Key Segments: 1
   Key Size: 2
   Minimum Record Size: 112
   Index Fill Quantity: 512, Data Fill Quantity: 512
   Segment Positions:     110
   Segment Sizes:           2
   Data Type: unsigned word
   Name: "SEQ_NO"
   First Data Bucket VBN: 13

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, INDEX          Root index bucket
%ANLRMS-I-DOWNPATH, DATA           Data buckets
```

Example 1 (Sheet 12 of 15)

```
ANALYZE>  DOWN INDEX

BUCKET HEADER (VBN 14)

   Check Character: %X'00'
   Key of Reference: 1
   VBN Sample: 14
   Free Space Offset: %X'0010'
   Free Record ID: 1
   Next Bucket VBN: 14
   Level: 1
   Bucket Header Flags:
           (0)  BKT$V_LASTBKT    1
           (1)  BKT$V_ROOTBKT    1
   Bucket Pointer Size: 2

   VBN Free Space Offset: %X'01F9'


ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, RECORDS        Index records


ANALYZE>  DOWN RECORDS

   INDEX RECORD (VBN 14, offset %X'000E')

          2-Byte Bucket Pointer: 13
          Key:
                  7  6  5  4  3  2  1  0          01234567
                 -----------------------          --------
                                  FF FF|  0000  |..      |
```

```
ANALYZE> UP   [  ...  UP ]

KEY DESCRIPTOR #1 (VBN 2, offset %X'0000')


ANALYZE>  DOWN DATA

BUCKET HEADER (VBN 13)

   Check Character: %X'0B'
   Key of Reference: 1
   VBN Sample: 13
   Free Space Offset: %X'0071'
   Free Record ID: 1
   Next Bucket VBN: 13
   Level: 0
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT    1

ANALYZE>  DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNPATH, SIDRS        SIDR record

ANALYZE>  DOWN SIDRS

   SIDR RECORD (VBN 13, offset %X'000E')

          Key:
                    7  6  5  4  3  2  1  0          01234567
                   ------------------------        --------
                                  00 01|  0000  |..      |
ANALYZE>  REST

   SIDR RECORD (VBN 13, offset %X'0017')

          Key:
                    7  6  5  4  3  2  1  0          01234567
                   ------------------------        --------
                                  00 02|  0000  |..      |

   SIDR RECORD (VBN 13, offset %X'0020')

          Key:
                    7  6  5  4  3  2  1  0          01234567
                   ------------------------        --------
                                  00 03|  0000  |..      |

   SIDR RECORD (VBN 13, offset %X'0029')

          Key:
                    7  6  5  4  3  2  1  0          01234567
                   ------------------------        --------
                                  00 04|  0000  |..      |
```

Example 1 (Sheet 14 of 15)

```
SIDR RECORD (VBN 13, offset %X'0032')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 05|  0000  |..      |

SIDR RECORD (VBN 13, offset %X'003B')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 06|  0000  |..      |

SIDR RECORD (VBN 13, offset %X'0044')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 07|  0000  |..      |

SIDR RECORD (VBN 13, offset %X'004D')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 08|  0000  |..      |

SIDR RECORD (VBN 13, offset %X'0056')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 09|  0000  |..      |

SIDR RECORD (VBN 13, offset %X'005F')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 0A|  0000  |..      |

SIDR RECORD (VBN 13, offset %X'0068')

        Key:
                7  6  5  4  3  2  1  0          01234567
               ------------------------         --------
                                  00 0B|  0000  |..      |

%ANLRMS-I-RESTDONE, All structures at this level have been displayed.
```

Example 1 (Sheet 15 of 15)

```
ANALYZE>  DUMP 13

DUMP OF VIRTUAL BLOCK 13:
     7  6  5  4  3  2  1  0          01234567
    -----------------------         --------
    00 01 00 71 00 0D 01 0B|  0000  |....q...|
    00 07 01 00 00 00 00 0D|  0008  |........|
    07 00 04 00 01 80 00 01|  0010  |........|
    00 04 00 02 80 00 02 00|  0018  |........|
    04 00 03 80 00 03 00 07|  0020  |........|
    00 04 80 00 04 00 07 00|  0028  |........|
    03 80 00 05 00 07 00 04|  0030  |........|
    80 00 06 00 07 00 05 00|  0038  |........|
    00 07 00 07 00 04 00 05|  0040  |........|
    08 00 07 00 06 00 01 80|  0048  |........|
    00 07 00 05 00 04 80 00|  0050  |........|
    07 00 05 00 05 80 00 09|  0058  |........|
    00 0F 00 04 80 00 0A 00|  0060  |........|
    0F 00 05 80 00 0B 00 07|  0068  |........|
    00 00 00 00 00 00 00 00|  0070  |........|
    00 00 00 00 00 00 00 00|  0078  |........|
    00 00 00 00 00 00 00 00|  0080  |........|
    00 00 00 00 00 00 00 00|  0088  |........|
    00 00 00 00 00 00 00 00|  0090  |........|
    00 00 00 00 00 00 00 00|  0098  |........|
    00 00 00 00 00 00 00 00|  00A0  |........|
    00 00 00 00 00 00 00 00|  00A8  |........|
    00 00 00 00 00 00 00 00|  00B0  |........|
    00 00 00 00 00 00 00 00|  00B8  |........|
    00 00 00 00 00 00 00 00|  00C0  |........|
    00 00 00 00 00 00 00 00|  00C8  |........|
    00 00 00 00 00 00 00 00|  00D0  |........|
    00 00 00 00 00 00 00 00|  00D8  |........|
    00 00 00 00 00 00 00 00|  00E0  |........|
    00 00 00 00 00 00 00 00|  00E8  |........|
    00 00 00 00 00 00 00 00|  00F0  |........|
    00 00 00 00 00 00 00 00|  00F8  |........|
    00 00 00 00 00 00 00 00|  0100  |........|
    00 00 00 00 00 00 00 00|  0108  |........|
    00 00 00 00 00 00 00 00|  0110  |........|
    00 00 00 00 00 00 00 00|  0118  |........|
    00 00 00 00 00 00 00 00|  0120  |........|
    00 00 00 00 00 00 00 00|  0128  |........|
    00 00 00 00 00 00 00 00|  0130  |........|
    00 00 00 00 00 00 00 00|  0138  |........|
    00 00 00 00 00 00 00 00|  0140  |........|
    00 00 00 00 00 00 00 00|  0148  |........|
    00 00 00 00 00 00 00 00|  0150  |........|
    00 00 00 00 00 00 00 00|  0158  |........|
    00 00 00 00 00 00 00 00|  0160  |........|
    00 00 00 00 00 00 00 00|  0168  |........|
    00 00 00 00 00 00 00 00|  0170  |........|
    00 00 00 00 00 00 00 00|  0178  |........|
    00 00 00 00 00 00 00 00|  0180  |........|
    00 00 00 00 00 00 00 00|  0188  |........|
    00 00 00 00 00 00 00 00|  0190  |........|
    00 00 00 00 00 00 00 00|  0198  |........|
    00 00 00 00 00 00 00 00|  01A0  |........|
    00 00 00 00 00 00 00 00|  01A8  |........|
    00 00 00 00 00 00 00 00|  01B0  |........|
    00 00 00 00 00 00 00 00|  01B8  |........|
    00 00 00 00 00 00 00 00|  01C0  |........|
    00 00 00 00 00 00 00 00|  01C8  |........|
    00 00 00 00 00 00 00 00|  01D0  |........|
    00 00 00 00 00 00 00 00|  01D8  |........|
    00 00 00 00 00 00 00 00|  01E0  |........|
    00 00 00 00 00 00 00 00|  01E8  |........|
    00 00 00 00 00 00 00 00|  01F0  |........|
    0B 00 00 00 00 00 00 00|  01F8  |........|

ANALYZE>  EXIT
```

Example 2. Tracking a Record from Key Value to Data Record

(Sheet 1 of 4)

```
$ ANALYZE/RMS/INTER  INTER11.DAT

FILE HEADER
.
.
.


ANALYZE>  DOWN

RMS FILE ATTRIBUTES
.
.
.

ANALYZE>  DOWN

FIXED PROLOG

   Number of Areas: 3, VBN of First Descriptor: 3
   Prolog Version: 3


ANALYZE> DOWN KEYS

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

   Next Key Descriptor VBN: 2, Offset: %X'0000'
   Index Area: 1, Level 1 Index Area: 1, Data Area: 0
   Root Level: 2
   Index Bucket Size: 1, Data Bucket Size: 1
   Root VBN: 9
   Key Flags:
        (0)  KEY$V_DUPKEYS    0
        (3)  KEY$V_IDX_COMPR  0
        (4)  KEY$V_INITIDX    0
        (6)  KEY$V_KEY_COMPR  0
        (7)  KEY$V_REC_COMPR  0
   Key Segments: 1
   Key Size: 110
   Minimum Record Size: 110
   Index Fill Quantity: 512, Data Fill Quantity: 512
   Segment Positions:       0
   Segment Sizes:         110
   Data Type: string
   Name: "LAST_NAME"
   First Data Bucket VBN: 4
```

Example 2 (Sheet 2 of 4)

```
 ANALYZE>  DOWN INDEX

BUCKET HEADER (VBN 9)

   Check Character: %X'01'
   Key of Reference: 0
   VBN Sample: 9
   Free Space Offset: %X'00EA'
   Free Record ID: 1
   Next Bucket VBN: 9
   Level: 2
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT     1
          (1)  BKT$V_ROOTBKT     1
   Bucket Pointer Size: 2

   VBN Free Space Offset: %X'01F7'


ANALYZE>  DOWN RECORDS

   INDEX RECORD (VBN 9, offset %X'000E')
          2-Byte Bucket Pointer: 7
          Key:
```

7	6	5	4	3	2	1	0		01234567
20	20	20	53	45	4E	4F	4A	0000	JONES
20	20	20	20	20	20	20	20	0008	
20	20	20	20	20	20	20	20	0010	
20	20	20	20	20	20	20	20	0018	
20	20	20	20	20	20	20	20	0020	
20	20	20	20	20	20	20	20	0028	
20	20	20	20	20	20	20	20	0030	
20	20	20	20	20	20	20	20	0038	
20	20	20	20	20	20	20	20	0040	
20	20	20	20	20	20	20	20	0048	
20	20	20	20	20	20	20	20	0050	
20	20	20	20	20	20	20	20	0058	
20	20	20	20	20	20	20	20	0060	
		20	20	20	20	20	20	0068	

```
ANALYZE>  DOWN DEEPER

BUCKET HEADER (VBN 7)

   Check Character: %X'04'
   Key of Reference: 0
   VBN Sample: 7
   Free Space Offset: %X'00EA'
   Free Record ID: 1
   Next Bucket VBN: 8
   Level: 1
   Bucket Header Flags:
          (0)  BKT$V_LASTBKT     0
          (1)  BKT$V_ROOTBKT     0
   Bucket Pointer Size: 2
   VBN Free Space Offset: %X'01F7'
```

Example 2 (Sheet 3 of 4)

```
ANALYZE>  DOWN RECORDS

   INDEX RECORD (VBN 7, offset %X'000E')

           2-Byte Bucket Pointer: 4
           Key:
                    7  6  5  4  3  2  1  0          01234567
                   -----------------------         --------
                   20 20 20 20 48 53 55 42 | 0000  |BUSH    |
                   20 20 20 20 20 20 20 20 | 0008  |        |
                   20 20 20 20 20 20 20 20 | 0010  |        |
                   20 20 20 20 20 20 20 20 | 0018  |        |
                   20 20 20 20 20 20 20 20 | 0020  |        |
                   20 20 20 20 20 20 20 20 | 0028  |        |
                   20 20 20 20 20 20 20 20 | 0030  |        |
                   20 20 20 20 20 20 20 20 | 0038  |        |
                   20 20 20 20 20 20 20 20 | 0040  |        |
                   20 20 20 20 20 20 20 20 | 0048  |        |
                   20 20 20 20 20 20 20 20 | 0050  |        |
                   20 20 20 20 20 20 20 20 | 0058  |        |
                   20 20 20 20 20 20 20 20 | 0060  |        |
                      20 20 20 20 20 20 20 | 0068  |        |

ANALYZE>  DOWN DEEPER

BUCKET HEADER (VBN 4)

   Check Character: %X'09'
   Key of Reference: 0
   VBN Sample: 4
   Free Space Offset: %X'011B'
   Free Record ID: 6
   Next Bucket VBN: 6
   Level: 0
   Bucket Header Flags:
           (0)  BKT$V_LASTBKT    0
```

Example 2 (Sheet 4 of 4)

```
ANALYZE>  DOWN RECORDS

   PRIMARY DATA RECORD (VBN 4, offset %X'000E')

           Record Control Flags:
                  (2)  IRC$V_DELETED    0
                  (3)  IRC$V_RRV        0
                  (4)  IRC$V_NOPTRSZ    0
                  (5)  IRC$V_RU_DELETE  0
                  (6)  IRC$V_RU_UPDATE  0
           Record ID: 2
           RRV ID: 2, 4-Byte Bucket Pointer: 4
           Key:
                 7  6  5  4  3  2  1  0          01234567
                ------------------------          --------
                20 20 20 20 45 48 53 41|  0000  |ASHE    |
                20 20 20 20 20 20 20 20|  0008  |        |
                20 20 20 20 20 20 20 20|  0010  |        |
                20 20 20 20 20 20 20 20|  0018  |        |
                20 20 20 20 20 20 20 20|  0020  |        |
                20 20 20 20 20 20 20 20|  0028  |        |
                20 20 20 20 20 20 20 20|  0030  |        |
                20 20 20 20 20 20 20 20|  0038  |        |
                20 20 20 20 20 20 20 20|  0040  |        |
                20 20 20 20 20 20 20 20|  0048  |        |
                20 20 20 20 20 20 20 20|  0050  |        |
                20 20 20 20 20 20 20 20|  0058  |        |
                20 20 20 20 20 20 20 20|  0060  |        |
                      20 20 20 20 20 20|  0068  |        |

ANALYZE>  EXIT
```

MODULE 17
DATA RECOVERY FOR CORRUPTED INDEXED FILES

Major Topics

- Detecting problems
  - ANALYZE/RMS_FILE/CHECK
  - DUMP utility
- Guidelines for recovering data from corrupted indexed files
- Introduction to PATCH utility
- Data recovery examples

Source

Guide to VAX/VMS File Applications — Chapter 10 (Section 10.1)

MODULE 17
DATA RECOVERY FOR
CORRUPTED INDEXED FILES

Major Topics:

- Detecting problems
  - ANALYZE/RMS_FILE/CHECK
  - DUMP utility
- Guidelines for recovering data from corrupted indexed files
- Introduction to PATCH utility
- Data recovery examples

Source:

Guide to VAX/VMS File Applications — Chapter 10 (Section 10.1)

DETECTING PROBLEMS

Corrupted files are seldom encountered in a VAX environment. Programmers who work with indexed files on a daily basis month after month may eventually encounter a corrupted file due to a hardware problem, such as a power failure or disk head failure. In the event that a recent backup copy of the file does not provide a satisfactory solution, this module provides some tools for trying to salvage the data yourself. In some cases, particularly with serious hardware problems, the file may not be able to be recovered.

Some indicators of a possible data corruption problem are:

- Error message -- Bucket format check failed for VBN = #
- The file doesn't have as many records as it should

If the error was immediately preceded by a series of hardware errors, it may be best to restore a backup copy of the file, since there may be other problems that have not been encountered as yet.

The following utilities are tools for detecting problems and recovering data.

- COPY or BACKUP utility
- ANALYZE/RMS/CHECK utility
- DUMP utility
- CONVERT utility
- PATCH utility (in extreme cases)

Example 1. ANALYZE/RMS/CHECK Output For Corrupted File

(Sheet 1 of 2)

```
$ANALYZE/RMS/CHECK/OUT=CORRUPT1.CHECK  CORRUPT1.DAT
-------------------------------------------------------

Check RMS File Integrity                    6-DEC-1985 21:47:04.38
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT1.DAT;1          Page 1

FILE HEADER

    File Spec: DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT1.DAT;1
    File ID: (15352,46,0)
    Owner UIC: [010,007]
    Protection:  System: RWED, Owner: RWED, Group: RWED, World: RWED
    Creation Date:   15-NOV-1985 14:46:57.24
    Revision Date:   18-NOV-1985 12:31:35.53, Number: 3
    Expiration Date: none specified
    Backup Date:     none posted
    Contiguity Options:  contiguous-best-try
    Performance Options: none
    Reliability Options: none
    Journaling Enabled:  none

RMS FILE ATTRIBUTES

    File Organization: indexed
    Record Format: fixed
    Record Attributes:   carriage-return
    Maximum Record Size: 112
    Longest Record: 112
    Blocks Allocated: 16, Default Extend Size: 1
    Bucket Size: 1
    Global Buffer Count: 0


FIXED PROLOG

    Number of Areas: 3, VBN of First Descriptor: 3
    Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')

    Bucket Size: 1
    Reclaimed Bucket VBN: 0
    Current Extent Start: 15, Blocks: 2, Used: 2, Next: 17
    Default Extend Quantity: 1
    Total Allocation: 8

AREA DESCRIPTOR #1 (VBN 3, offset %X'0040')

    Bucket Size: 1
    Reclaimed Bucket VBN: 0
    Current Extent Start: 7, Blocks: 6, Used: 3, Next: 10
    Default Extend Quantity: 1
    Total Allocation: 6
```

Example 1 (Sheet 2 of 2)

```
Check RMS File Integrity                    6-DEC-1985 21:47:04.54
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT1.DAT;1              Page 2

AREA DESCRIPTOR #2 (VBN 3, offset %X'0080')

   Bucket Size: 1
   Reclaimed Bucket VBN: 0
   Current Extent Start: 13, Blocks: 2, Used: 2, Next: 15
   Default Extend Quantity: 2
   Total Allocation: 2

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

   Next Key Descriptor VBN: 2, Offset: %X'0000'
   Index Area: 1, Level 1 Index Area: 1, Data Area: 0
   Root Level: 2
   Index Bucket Size: 1, Data Bucket Size: 1
   Root VBN: 9
   Key Flags:
           (0)  KEY$V_DUPKEYS    0
           (3)  KEY$V_IDX_COMPR  0
           (4)  KEY$V_INITIDX    0
           (6)  KEY$V_KEY_COMPR  0
           (7)  KEY$V_REC_COMPR  0
   Key Segments: 1
   Key Size: 110
   Minimum Record Size: 110
   Index Fill Quantity: 512, Data Fill Quantity: 512
   Segment Positions:       0
   Segment Sizes:         110
   Data Type: string
   Name: "LAST_NAME"
   First Data Bucket VBN: 4
***  VBN 8:  Index bucket references missing data bucket with VBN 200.
***  Drastic structure error precludes further analysis.


The analysis uncovered 2 errors.

ANALYZE/RMS/CHECK/OUT=CORRUPT1.CHECK CORRUPT1.DAT
```

DUMP Utility

The DUMP command offers you various capabilities.

- DUMP/HEADER gives you a formatted printout of all the fields in the file header.

- DUMP/RECORDS dumps just the data records. This is very useful with files that contain a lot of overhead that does not interest you (such as indexes).

- DUMP can print your file (or its records) in a wide variety of formats: per byte, per word, or per longword; in octal, decimal, or hexadecimal.

- /BLOCKS[=(option[,...])]

  Specifies that the input medium be dumped one block at a time. This is the default for all devices except network devices. You cannot specify /BLOCKS for network devices.

  You can use one or more of the following options to select a range of blocks to be dumped:

  START:n — Specifies the number of the first block to be dumped. By default, the dump begins with the first block of the file or device.

  END:n — Specifies the number of the last block to be dumped. By default, the dump ends with the last block of the file or device. If the input is a disk file, the /ALLOCATED qualifier determines whether the last block is the end of file block or the last allocated block.

  COUNT:n — Specifies the number of blocks to be dumped. This option provides an alternate way to specify the last block to be dumped.

  If you specify only one option, you can omit the parentheses. You cannot specify both END and COUNT.

  Blocks are usually numbered beginning with 1. However, for a disk device that is mounted with the /FOREIGN qualifier, blocks are numbered beginning with 0.

  If you specify /BLOCKS, you cannot specify /RECORDS.

- /OUTPUT[=file-spec]

  Specifies that the DUMP output be written to the specified file. By default, the DUMP command writes output to SYS$OUTPUT. If you specify /OUTPUT without a file specification, the DUMP command writes output to a file with the same file name as the input file and the file type DMP.

Example 2. DUMP/BLOCKS Output

(Sheet 1 of 2)

```
$DUMP/BLOCKS=(START:1,END:4)/OUTPUT=DUMP_INTER11.LIS INTER11.DAT

--------------------------------------------------------------------------------

Dump of file DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INTER11.DAT;1 on 15-JAN-1986
                                                              14:52:40.35
File ID (33247,11,0)    End of file block 16 / Allocated 16

Virtual block number 1 (00000001), 512 (0200) bytes

 00000000 02000200 006E006E 00010000 00000009 01010200 01010000 00000002 .....................n.n..... 000000
 20202045 4D414E5F 5453414C 00000000 0000006E 00000000 00000000 00000000 ...........n.......LAST_NAME    000020
 00000000 00000000 00000004 20202020 20202020 20202020 20202020 20202020                          ..... 000040
 00000000 00000000 00000003 00000000 00000000 00000000 03030000 00000000 ................................ 000060
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000080
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000A0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000C0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000E0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000100
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000120
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000140
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000160
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000180
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0001A0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0001C0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ...............................F? 0001E0
 BF460000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Dump of file DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INTER11.DAT;1 on 15-JAN-1986
                                                              14:52:40.35
File ID (33247,11,0)    End of file block 16 / Allocated 16

Virtual block number 2 (00000002), 512 (0200) bytes

 0000006E 02000200 00700102 00010200 0000000E 01010102 02020000 00000000 .....................p.....n... 000000
 20202020 20204F4E 5F514553 00000000 00000002 00000000 00000000 00000000 ...................SEQ_NO         000020
 00000000 00000002 0000000D 20202020 20202020 20202020 20202020 20202020                          ..... 000040
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000060
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000080
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000A0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000C0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000E0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000100
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000120
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000140
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000160
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000180
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0001A0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0001C0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................! 0001E0
 A1970000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Example 2 (Sheet 2 of 2)

```
Dump of file DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INTER11.DAT;1 on 15-JAN-1986
                                                              14:52:40.35

File ID (33247,11,0)   End of file block 16 / Allocated 16

Virtual block number 3 (00000003), 512 (0200) bytes

 00000000 00000011 00000002 00000002 0000000F 00000000 20000000 01000000 ....... ........................ 000000
 00000000 00000000 00000000 00080000 00000000 00000000 00000001 00000000 ................................ 000020
 00000000 0000000A 00000003 00000006 00000007 00000000 20000000 01010000 ....... ........................ 000040
 00000000 00000000 00000000 00060000 00000000 00000000 00000001 00000000 ................................ 000060
 00000000 0000000F 00000002 00000002 0000000D 00000000 20000000 01020000 ....... ........................ 000080
 00000000 00000000 00000000 00020000 00000000 00000000 00000002 00000000 ................................ 0000A0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000C0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0000E0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000100
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000120
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000140
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000160
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 000180
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0001A0
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ................................ 0001C0
 63750000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ............................uc 0001E0

Dump of file DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INTER11.DAT;1 on 15-JAN-1986
                                                              14:52:40.35

File ID (33247,11,0)   End of file block 16 / Allocated 16

Virtual block number 4 (00000004), 512 (0200) bytes

 20202020 20454853 41000000 04000200 02020000 00000006 0006011B 00040009 ....................ASHE     000000
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          000020
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          000040
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          000060
 20202020 20202020 20202020 48535542 00000004 00050005 02000220 20202020     ..........BUSH       000080
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          0000A0
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          0000C0
 00062020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                        .. 0000E0
 20202020 20000000 06000200 040A0000 00100001 00030A00 00000500 0100010A ........................     000100
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          000120
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          000140
 00050001 00010A00 04202020 20202020 20202020 20202020 20202020 20202020                 ......... 000160
 20202020 20202020 20202020 00000006 00020004 0A000000 05000200 030A0000 ..................       000180
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          0001A0
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020                          0001C0
 09000000 00000000 05000100 010A0003 20202020 20202020 20202020 20202020             ............. 0001E0
```

Example 3. DUMP/RECORDS Output

```
$DUMP/RECORDS=(START:1,END:8)/OUTPUT=DUMP_INDEX2.LIS INDEX2.DAT

------------------------------------------------------------------------------

Dump of file DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDEX2.DAT;1 on 15-JAN-1986
                                                              15:03:11.48
File ID (31656,25,0)   End of file block 166 / Allocated 166

Record number 1 (00000001), 80 (0050) bytes

 20454E49 444C4152 45472020 20202020 20202020 20204B41 5A312020 20202020        1ZAK            GERALDINE  000000
 59474749 48202020 44564C42 20454C50 414D2032 39313533 39393130 30313832 2810019935192 MAPLE BLVD   HIGGY 000020
                                     32303534 32495620 2020524F 4E414D20  MANOR   VI24502..............  000040

Record number 2 (00000002), 80 (0050) bytes

 2020202E 43204252 41422020 20202020 2020204E 4F544C55 46322020 20202020        2FULTON         BARB C.    000000
 4E574F52 42202020 20594157 2052454B 41422031 38303336 34383637 36383439 9486768463081 BAKER WAY    BROWN 000020
                                     30393731 39414320 20202045 4C4C4956 VILLE    CA91790.............  000040

Record number 3 (00000003), 80 (0050) bytes

 2020202E 4C205952 41472020 20202020 20524153 45424549 54332020 20202020        3TIEBESAR       GARY L.    000000
 52454B41 42435249 4320454E 41524843 4F432039 33303631 38303932 34313436 6414290816039 COCHRANE CIRCBAKER 000020
                                     36303030 32494420 20202045 4C4C4956 VILLE    DI20006.............  000040

Record number 4 (00000004), 80 (0050) bytes

 2020202E 55204C4C 49422020 20202020 204E414D 52454B43 41342020 20202020        4ACKERMAN       BILL U.    000000
 4C414843 4D202044 414F5220 454C4148 434D2034 35383137 35383336 35353437 7455638571854 MCHALE ROAD  MCHAL 000020
                                     36303030 32494420 20202020 4E4F5445 ETON     DI20006.............  000040

Record number 5 (00000005), 80 (0050) bytes

 202E4A20 41485452 414D2020 20202020 2020204E 4F544C55 46352020 20202020        5FULTON         MARTHA J.  000000
 45574843 5320454C 43524943 20594747 49482030 35353533 33313034 30383330 0380401335550 HIGGY CIRCLE SCHWE 000020
                                     30393731 3941434C 4C495654 52454B49 IKERTVILLCA91790.............  000040

Record number 6 (00000006), 80 (0050) bytes

 4D205245 4D495452 4F4D2020 20202020 2020204E 4F544C55 46362020 20202020        6FULTON         MORTIMER M 000000
 45574843 53204441 4F522054 4E414C4C 41472031 33353834 30343033 37393735 5797304048531 GALLANT ROAD SCHWE 000020
                                     31303139 3745544C 4C495654 52454B49 IKERTVILLTE79101.............  000040

Record number 7 (00000007), 80 (0050) bytes

 2020202E 5320414E 4E412020 20202020 20202020 20454853 41372020 20202020        7ASHE           ANNA S.    000000
 414C4C41 47202020 20202059 4157204B 415A2030 30353635 33323431 37323733 3727142356500 ZAK WAY      GALLA 000020
                                     32303534 32495648 47554F52 4F42544E NTBOROUGHVI24502.............  000040

Record number 8 (00000008), 80 (0050) bytes

 202E4E20 444C414E 4F442020 20202020 20202020 20205541 4E382020 20202020        8NAU            DONALD N.  000000
 4C414843 4D435249 4320454E 41524843 4F432036 30323432 32333237 31303633 3601723224206 COCHRANE CIRCMCHAL 000020
                                     36313232 334C4620 2020454C 4C495645 EVILLE   FL32216.............  000040
```

GUIDELINES FOR RECOVERING DATA FROM CORRUPTED INDEXED FILES

1. Make a backup copy of the file immediately. In attempting to recover the data, use the backup copy.

2. Verify that the data file that may be corrupted is the correct data file. Use ANALYZE/RMS/INTERACTIVE or DUMP to check the file header, prologue, and key descriptors to make sure that there has not been a mixup in files. Possible areas to check out:

   - A logical name may be causing the program to use the wrong file.
   - If users are sharing an account, they sometimes decide to use the same file name for different data files.
   - Incorrect usage of DCL commands can make a copy of a file with different attributes from the original file. (For example, an indexed file input to EDIT/EDT is output as a sequential file.)

3. Determine whether the file is really corrupt using ANALYZE/RMS/CHECK. The /CHECK qualifier does not find all types of corruption.

   If the /CHECK qualifier detects any errors, the file has been corrupted. If you have had a hardware problem, such as a power failure or a disk head failure, then the hardware probably caused the corruption. If you have not had any hardware problems, then a software error may have caused the corruption.

   If the /CHECK qualifier indicates a number of severe errors, you should probably stop at this step and go back to the most recent backup copy that is uncorrupted.

4. If some particular virtual block is identified as having a problem by the /CHECK qualifier, get a dump of all the blocks in the index bucket or data bucket that begins with that virtual block.

   Before checking it against the internal layouts provided in Module 5, do the next step.



5. Use the CONVERT utility to try to restore the data.

   a. If none of the primary level 0 data buckets are corrupted, you will generally be able to recover the data as follows:

      $CONVERT/STAT corrupted_index_file new_index_file

   b. In general, if the corruption is associated with a primary index bucket or with any of the secondary index buckets then the convert in step (a.) should be successful. There is one exception to this: when the corruption involves one of the primary index buckets in the initial pathway down to the first level 0 data bucket, the CONVERT utility is not able to recover the data from the primary key. If you have at least one alternate key, attempt to convert the data by outputting a sequential version based on the alternate key sorted order as a first step and then convert the sequential file to an indexed one as a second step.

      Step 1.

      If you do not have an FDL file for the indexed file, you can obtain one from the corrupted file.

      $ANAL/RMS/FDL corrupted_index_file

      The FDL file produced by ANALYZE will have to be edited.

      $EDIT new_index.FDL

      - Delete the version number in the filename.

      Step 2.

      Produce an FDL file for the sequential file.

      $EDIT/FDL seq.FDL

Step 3.

```
$CONVERT/KEY=1/FDL=seq.FDL/STAT ___________  ___________
                                   ^            ^
                                   |            |
                          corrupted_index_file  |
                                                |
                              good_sequential_file_output
```

Step 4.

```
$CONVERT/FDL=new_index.FDL/STAT ______________  ___________
                                good_sequential  new_index
```

If you do not have an alternate key or the CONVERT utility using an alternate key is not successful, see Step 7.

6. If the CONVERT utility is not able to restore the data file, or is not able to restore all the data, examine the dump you obtained in Step 4.

   Are you able to identify what the problem is? For example, is the check byte at the beginning of the bucket equal to the one at the end of the bucket?

7. If you were able to identify what the problem is in Step 6, or if the corruption involves the initial pathway down through the primary index bucket and you have no alternate key, you may wish to try to patch the file.

   CAUTION

   Patching a corrupted data file is not encouraged. The recommended procedure is to use the CONVERT utility, or go to a backup copy. Patching is undertaken by users at their own risk. The ISAM structures are extremely fragile. If in patching a file, a byte or even a bit is misplaced, subsequent processing of the file may crash the system.

   You can attempt to patch the file if you are able to detect:

   - what the source of the corruption is
   - what value in a particular byte location is not what ISAM expects
   - what the value should be

Before undertaking this step, be sure you have a backup copy of the corrupted file as directed in Step 1. Also be sure you have a file copy in your directory of the DUMP (see Step 4) of the bucket in question.

You should not think of using the PATCH utility if there are a number of errors. It should only be used selectively.

The PATCH utility is primarily for patching "IMAGE" files, but it may be used to patch any kind of file by using the qualifier /ABSOLUTE. In addition, to do the patch in place use the qualifier /NONEW_VERSION; otherwise, in the case of an indexed file, it outputs a new SEQUENTIAL version.

The steps for using the PATCH utility are described at the end of this section.

IMPORTANT

The patched file definitely must be converted to produce a valid index structure (see Step 9).

8. If you used the PATCH utility in Step 7, rerun ANALYZE/RMS/CHECK on the patched data file.

9. If the data was recovered by the PATCH utility (even if no errors are now identified by /CHECK), make a new copy of the file using the CONVERT utility so that the index structures are rebuilt.

10. If the problem was with a data bucket (primary level 0 data buckets), use the dump obtained of the respective bucket (Step 4) to identify whether any data records in the bucket need to be updated or deleted.

    If the problem was with a primary index bucket or with a bucket in a secondary index tree, then the convert in Step 5 or in Step 9 reconstructed these buckets and no further action is required.

11. If the corruption appears to be due to something other than a minor hardware failure (such as a power failure in the middle of writing an updated bucket back to disk), try to find out what caused the corruption.

    You may be able to save yourself a lot of work in the future by trying to find out what happened to the file. If, for example, the file is always corrupted on Tuesday, it is worth investigating what is different on Tuesday.

12. In all cases, even if your data recovery was successful, if you believe that some DIGITAL software caused the error, submit a Software Performance Report (SPR). Always include the ANALYZE/RMS/CHECK report, a copy of the data file, and a description of what was done with the data file. If possible, also supply a version of the file prior to the corruption and the program or procedure which led to the corruption. Being able to reproduce the problem is of tremendous value.

Qualifiers

/ABSOLUTE
The /ABSOLUTE function allows a user to patch any file (not just image files) at absolute virtual addresses relative to the beginning of the file. This feature allows replacement of existing data with new data of the same length. If data is smaller than that of the original data, the PATCH utility uses the appropriate fill character for the mode in use. For example, if the current mode is data (numeric or ASCII) mode, a null is used for fill. Any patch operation that results in a data replacement longer than the length of the original data generates an error message and terminates the command in progress; either the PATCH utility prompt or DCL prompt is then displayed, whichever is appropriate.

/NEW_VERSION
The /NEW_VERSION qualifier is used in conjunction with the /ABSOLUTE qualifier to control whether a new version of the patched file is created or the contents of the existing file are modified in place. /NEW_VERSION is the default. If /NONEW_VERSION is selected, the PATCH command UPDATE will act as a checkpoint operation; all modifications made to the file are written back to the file instead of waiting until image exit. If /ABSOLUTE is not specified with /NONEW_VERSION, /NONEW_VERSION is ignored; a new version of the file will be created.

NOTE

If /NONEW_VERSION is specified, the file will be overwritten. No attempt on the part of the user, including pressing CTRL/Y, will prevent this result. Therefore, you should have a backup copy of the file before making any attempt to patch it.

There are only two PATCH commands appropriate for replacing data when the /ABSOLUTE qualifier is used.

1. The EXAMINE command for read operations.

2. The REPLACE command for write operations.

Commands that attempt to expand the file, such as ALIGN and INSERT, should be avoided because they will probably corrupt the file. (These commands will be trapped by the PATCH utility and an error message will be issued indicating that the replacement data must not exceed the length of the original data.)

EXAMINE
Displays the contents of the specified locations in terms of the current mode settings.

```
PATCH> EXAMINE location [:location][,...]
```

location
Specifies one or more locations whose contents are to be displayed. Several locations can be specified in a comma-separated list or colon-separated range. Both lists and ranges can be specified in a single command.

The location parameter can also be represented by a backslash operator (\).

REPLACE
Replaces the contents of one or more locations with new instructions or data in terms of the current mode settings.

```
PATCH> REPLACE location =current-contents[,...] new-content...
```

location
Specifies either a single location whose contents are to be replaced or the starting address of a sequence of locations whose contents are to be replaced. The length of the sequence depends on the current mode settings (/BYTE, /WORD, or /LONG). The default length is a longword (4 bytes).

current-contents
Specifies one or more data entries to be replaced. The data specified must be the actual contents.

new-contents
Specifies one or more data entries that are to replace the current contents.

Do not specify conflicting data types within a single REPLACE command.

Use the REPLACE command to replace the contents of one or more locations with new data in terms of the current mode settings. Before performing the replacement, the REPLACE command confirms the contents of the specified locations.

When you replace ASCII or numeric data, the number of replacement entries cannot exceed the number of existing entries. For example, this means that if you confirm the contents of six consecutive locations, you can replace the contents of only those six locations. If the number of replacement entries is less than the number of existing entries, the remaining locations are filled with zeros.

In addition, the PATCH utility truncates replacement entries if they exceed the limit imposed on them by the current length mode. For ASCII characters, the right-most characters are discarded. For numeric data, the left-most digits are discarded.

Example

```
$ PATCH/ABSOLUTE/NONEW_VERSION LIN.COM
PATCH>EX/ASCII 57
00000057:  'MANA'
PATCH>REPLACE/ASCII 57='MANA'
NEW>  'mana'
NEW>  'test'
NEW>  exit
old:          00000057: 'MANA'
%PATCH-E-REPLACEERR, replacement value too large for location
PATCH>replace/ascii 57='MANA'
NEW>  'mana'
NEW>  exit
old:          00000057: 'MANA'
new:          00000057: 'mana'
```

DATA RECOVERY EXAMPLES

Example 4. Data Recovery Lab for CORRUPT3.DAT

Problem observed with INTER11.DAT:

Data file had eleven records when last accessed. Today the file appears to have a total of two records.

Steps

1. $COPY INTER11.DAT CORRUPT3.DAT

2. $ANALYZE/RMS/INTER CORRUPT3.DAT

   Verified was INTER11.DAT format

3. $ANALYZE/RMS/CHECK/OUT=CORRUPT3.CHECK CORRUPT3.DAT

   $ANLRMS-I-ERRORS, DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT3.DAT;1
   1 error

   CHECK /OUTPUT identifies bucket format check
   failed for VBN = 6

   NOTE

   VBN 6 is a level 0 data bucket.

4. $DUMP/BLOCKS=(START:6,END:6)/OUTPUT=CORRUPT3.DUMP CORRUPT3.DAT

5. Bucket VBN 6 is a data bucket.

   $CONVERT/STAT CORRUPT3.DAT GOOD3.DAT

   %CONV-F-READERR, error reading
   DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT3.DAT;1
   -RMS-F-CHK, bucket format check failed for VBN = 6

   $DELETE GOOD3.DAT;1 < NULL FILE >

6. Examine output from CORRUPT3.DUMP

```
                          CHECK bytes

VBN 6 - BYTE 0            02
        BYTE 511          01
```

NOTE

In this atypical case, bucket size happens to be one block.

7. Use the PATCH utility to change VBN 6 -- byte 511 to 02.

a. Location

```
512 x 6 = 3072
           - 1 (backup one byte to byte 511 in VBN 6)
          ----
          3071
```

b. Calculate location in HEX

```
$ X == 3071
$ SH SYM X
  X = 3071   Hex = 00000BFF   Octal = 00000005777
```

c. Run PATCH

```
$ PATCH/ABSOLUTE/NONEW CORRUPT3.DAT
%PATCH-I-NOGBL, some or all global symbols not accessible
%PATCH-I-NOLCL, image does not contain local symbols

PATCH>EXAMINE/BYTE 00000BFF
00000BFF:  01

PATCH>REPLACE/BYTE 00000BFF = 01
NEW>  02
NEW>  EXIT
old:    00000BFF:  01
new:    00000BFF:  02

PATCH>EXIT

%PATCH-I-OVERLAY,DISK$INSTRUCTR:[WOODS.RMS.COURSE]CORRUPT3.DAT;1
        being overwritten
```

8. $RENAME CORRUPT3.DAT GOOD3.DAT

$ANALY/RMS/CHECK/OUT=GOOD3.CHECK GOOD3.DAT

9. $CONVERT/STAT GOOD3.DAT INTER11_CONV.DAT

10. Check CORRUPT3.DUMP output to see whether any data records in VBN 6 need to be updated or deleted.

Example 5. Data Recovery Lab for CORRUPT4.DAT

Step 6 -- Identify nature of problem

The check bytes do not match in one of the index buckets (VBN 7) which lies on the initial path of the primary index. This index bucket is not the root primary index bucket (level 2), but a level 1 bucket.

Dump of VBN 7

```
Dump of file DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT4.DAT;1
File ID (37426,1,0)   End of file block 16 / Allocated 16

Virtual block number 7 (00000007), 512 (0200) bytes

 20202020 20202020 20202020 20204853 55420001 00000008 000100EA 00070004
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 454E4F4A 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202053
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 20202020 20202020 20202020 20202020 20534F4B 41522020 20202020 20202020
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
 FFFFFFFF FFFFFFFF 20202020 20202020 20202020 20202020 20202020 20202020
 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
 00000000 00000000 00000000 00000000 00000000 00000000 0000FFFF FFFFFFFF
 0300001F7 00040006 0005000F 00000000 00000000 00000000 00000000 00000000
```

Check chars
!
Do not match

There are two alternative ways to recover the data for this example.

1. Use alternate key 1 to recover a sequential version of the data file. See Step 5(b.).

2. Patch the ending check byte to match the beginning check byte in bucket VBN 7. See Step 7.

The first alternative is the preferred one, but for illustration purposes both alternatives will be demonstrated.

Alternative 1 -- CORRUPT4.DAT

Step 5(b.) -- Use alternate key to convert

1. If you don't have an FDL file for the indexed version, obtain one.

   $ANAL/RMS/FDL CORRUPT4.DAT <-- outputs CORRUPT4.FDL

2. $EDIT CORRUPT4.FDL

   - Delete version number in filename.

3. Use EDIT/FDL to produce a sequential FDL specification for output file in Step 4.

```
$EDIT/FDL  seq4.fdl
```

4.
```
$CONVERT/KEY=1/FDL=seq4.fdl/STAT CORRUPT4.DAT seq4.dat
                                              |
                                    sequential output
```

5.
```
$CONVERT/FDL=corrupt4.fdl/STAT seq4.dat GOOD4.DAT
                    |                   |
              FDL indexed          indexed output
```

Alternative 2 -- CORRUPT4.DAT

Step 7 -- Use the PATCH utility

1. Location

```
7 x 512 = 3584
           - 1 (backup one byte to byte 511 in VBN 7)
          ----
          3583
```

2. HEX location

```
$ X == 3583
$SH SYM X
X = 3583    Hex = 00000DFF  Octal = 00000006777
```

3. Patch

```
$PATCH/ABSOLUTE/NONEW  CORRUPT4.DAT
%PATCH-I-NOGBL, some or all global symbols not accessible
%PATCH-I-NOLCL, image does not contain local symbols

PATCH>EXAMINE/BYTE 00000DFF
00000DFF:  03

PATCH>REPLACE/BYTE 00000DFF = 03
NEW>  04
NEW>  EXIT
old:    00000DFF:  03
new:    00000DFF:  04

PATCH>EXIT

%PATCH-I-OVERLAY, DISK$INSTRUCTR:[WOODS.RMS.COURSE]CORRUPT4.DAT;1
      being overwritten
```

Appendix A

Appendix A

APPENDIX A

Indexed files -- specific points in RMS-coded instructions when locking is done at record, bucket, or file level.

1. Record operations (assumes no bucket splits)

   - $PUT
     - Initialization/validation (if sequential access, is key value of new record greater (ascending primary key) or less (descending primary key) than that of last record, etc.)
     - Position to point of insert (involves positioning through the index structure by key, and leaves data bucket locked)
     - Adjust "high set" appropriately
     - Build record overhead fields in bucket; move in record itself
     - Lock new record
     - Update new record (if necessary)
     - Unlock bucket
     - Insert alternate keys (if any) extracted from user buffer
   - $DELETED (assumes previous $GET/$FIND and record locked)
     - Initialization/validation (is there a current record, etc.)
     - If RRV, position by RFA to record (leaves bucket locked)
     - Save record in internal buffer
     - Delete the RRV (if any)
     - Delete the primary record itself
     - Unlock bucket
     - Delete all alternate keys, plucking values from saved record (if FAST_DELETE option not specified)

- $UPDATE (assumes previous $GET/$FIND and record locked)
  - Initialization/validation
  - If RRV, position by RFA to record (leaves bucket locked)
  - If alternate keys will change, then:
    a. Save old record
    b. Unlock data bucket
    c. Insert new SIDR entries
    d. Reposition by RFA to record (leaves bucket locked again)
  - Is new record size less than or equal to old size?
    + YES (smaller or same as old record)
    a. Adjust high set appropriately
    b. Insert record
    + NO (larger than old record)
    a. Save record ID
    b. Perform "pseudo-$DELETE"
    c. Perform "pseudo-$PUT" (stuffing saved record ID)
  - Unlock bucket
  - Delete old SIDR entries (if any) using old record buffer

2. Record operations involving bucket split(s) (assume old bucket is currently locked)
   - Lock area
   - Enough space for a new bucket?
     + YES (a bucket's worth of blocks is available)
     - Allocate new bucket
     - Unlock area

+ NO (must do a $EXTEND)

  - Lock file to prevent file operations
  - Lock prolog
  - Do the $EXTEND
  - Allocate new bucket
  - Unlock area
  - Unlock prolog
  - Unlock file

- Format new bucket
- Set new bucket's next pointer to old bucket's next pointer
- Set old bucket's next pointer to the new bucket
- Move data into new bucket
- Scan old bucket for records past the split point that have RRVs, and keep in a table.
- Update free space in old bucket and unlock it
- Update RRVs in internal table to point to new location of records. This involves multiple positionings by RRV -- one for each RRV to be updated.

  Note that SIDRs are not updated. SIDR entries may point to an RRV, which in turn points to the real record. Because of the RRV updating process, this level of indirection never goes beyond one.

Appendix B

Appendix B

APPENDIX B

Hexadecimal to Decimal Conversion Table

Hexadecimal to Decimal Conversion Table

8		7		6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,929,104	9	150,994,994	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	12,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,685,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

ASCII Table

B7 B6 B5 / BITS B4 B3 B2 B1	ROW	COLUMN 0		COLUMN 1		COLUMN 2		COLUMN 3		COLUMN 4		COLUMN 5		COLUMN 6		COLUMN 7	
0 0 0 0	0	NUL	0 0 0	DLE	20 16 10	SP	40 32 20	0	60 48 30	@	100 64 40	P	120 80 50	`	140 96 60	p	160 112 70
0 0 0 1	1	SOH	1 1 1	DC1 (XON)	21 17 11	!	41 33 21	1	61 49 31	A	101 65 41	Q	121 81 51	a	141 97 61	q	161 113 71
0 0 1 0	2	STX	2 2 2	DC2	22 18 12	"	42 34 22	2	62 50 32	B	102 66 42	R	122 82 52	b	142 98 62	r	162 114 72
0 0 1 1	3	ETX	3 3 3	DC3 (XOFF)	23 19 13	#	43 35 23	3	63 51 33	C	103 67 43	S	123 83 53	c	143 99 63	s	163 115 73
0 1 0 0	4	EOT	4 4 4	DC4	24 20 14	$	44 36 24	4	64 52 34	D	104 68 44	T	124 84 54	d	144 100 64	t	164 116 74
0 1 0 1	5	ENQ	5 5 5	NAK	25 21 15	%	45 37 25	5	65 53 35	E	105 69 45	U	125 85 55	e	145 101 65	u	165 117 75
0 1 1 0	6	ACK	6 6 6	SYN	26 22 16	&	46 38 26	6	66 54 36	F	106 70 46	V	126 86 56	f	146 102 66	v	166 118 76
0 1 1 1	7	BEL	7 7 7	ETB	27 23 17	'	47 39 27	7	67 55 37	G	107 71 47	W	127 87 57	g	147 103 67	w	167 119 77
1 0 0 0	8	BS	10 8 8	CAN	30 24 18	(	50 40 28	8	70 56 38	H	110 72 48	X	130 88 58	h	150 104 68	x	170 120 78
1 0 0 1	9	HT	11 9 9	EM	31 25 19	)	51 41 29	9	71 57 39	I	111 73 49	Y	131 89 59	i	151 105 69	y	171 121 79
1 0 1 0	10	LF	12 10 A	SUB	32 26 1A	*	52 42 2A	:	72 58 3A	J	112 74 4A	Z	132 90 5A	j	152 106 6A	z	172 122 7A
1 0 1 1	11	VT	13 11 B	ESC	33 27 1B	+	53 43 2B	;	73 59 3B	K	113 75 4B	[	133 91 5B	k	153 107 6B	{	173 123 7B
1 1 0 0	12	FF	14 12 C	FS	34 28 1C	,	54 44 2C	<	74 60 3C	L	114 76 4C	\	134 92 5C	l	154 108 6C	|	174 124 7C
1 1 0 1	13	CR	15 13 D	GS	35 29 1D	-	55 45 2D	=	75 61 3D	M	115 77 4D	]	135 93 5D	m	155 109 6D	}	175 125 7D
1 1 1 0	14	SO	16 14 E	RS	36 30 1E	.	56 46 2E	>	76 62 3E	N	116 78 4E	^	136 94 5E	n	156 110 6E	~	176 126 7E
1 1 1 1	15	SI	17 15 F	US	37 31 1F	/	57 47 2F	?	77 63 3F	O	117 79 4F	_	137 95 5F	o	157 111 6F	DEL	177 127 7F

KEY

ASCII CHARACTER

ESC	33	OCTAL
	27	DECIMAL
	1B	HEX

MA-7246