RMS Structures and Utilities on VAX/VMS

Student Guide

Prepared by Educational Services of Digital Equipment Corporation
The following are some of the trademarks of Digital Equipment Corporation.

<table>
<thead>
<tr>
<th>DECtalk</th>
<th>DECUS</th>
<th>DECwriter</th>
<th>Professional</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL-IN-1</td>
<td>DECUS</td>
<td>DECwriter</td>
<td>RAINBOW</td>
</tr>
<tr>
<td>CTS-300</td>
<td>DIBOL</td>
<td>DSM-11</td>
<td>RSTK</td>
</tr>
<tr>
<td>DATATRIEVE</td>
<td>DIRECT</td>
<td>FMS-11</td>
<td>RSX</td>
</tr>
<tr>
<td>DEC</td>
<td>DSM-11</td>
<td>FMS-11</td>
<td>TOPS-10</td>
</tr>
<tr>
<td>DECDirect</td>
<td>KI</td>
<td>KL10</td>
<td>TOPS-20</td>
</tr>
<tr>
<td>DECGmail</td>
<td>MASSBUS</td>
<td></td>
<td>UNIBUS</td>
</tr>
<tr>
<td>DECMate</td>
<td>MicroPower/Pascal</td>
<td>MicroVAX II</td>
<td>VAX</td>
</tr>
<tr>
<td>DECnet</td>
<td>VAXcluster</td>
<td>VAXstation VMS</td>
<td>VT</td>
</tr>
<tr>
<td>DECDservice</td>
<td></td>
<td></td>
<td>Work Processor</td>
</tr>
<tr>
<td>DECsystem-10</td>
<td>PDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECSYSTEM-20</td>
<td>P/OS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CONTENTS

Course Description ............................................x
Course Organization ...........................................x
Prerequisites ...................................................x
Course Goals ..................................................xi
Non-Goals ....................................................xii
Resources .....................................................xii

MODULE 1. OVERVIEW OF RMS DATA
STORES 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

iii
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

<table>
<thead>
<tr>
<th>Part 1. Internals</th>
<th>Overall Tree Structure and Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Indexed Sequential Files</td>
</tr>
<tr>
<td></td>
<td>Prolog Description</td>
</tr>
<tr>
<td>Key Descriptors</td>
<td>Area Descriptors</td>
</tr>
<tr>
<td></td>
<td>Data Bucket Structure</td>
</tr>
<tr>
<td></td>
<td>Data Record Format</td>
</tr>
<tr>
<td></td>
<td>Record Reference Vectors (RRVs) and Bucket Splits</td>
</tr>
<tr>
<td></td>
<td>Simple Bucket Splits</td>
</tr>
<tr>
<td></td>
<td>Multibucket Splits</td>
</tr>
<tr>
<td></td>
<td>Bucket Splits with Duplicate Records</td>
</tr>
<tr>
<td>Key and Data Compression (Prolog 3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Primary Key Compression</td>
</tr>
<tr>
<td></td>
<td>Data Compression</td>
</tr>
<tr>
<td>Index Bucket</td>
<td>Index Record Format</td>
</tr>
<tr>
<td></td>
<td>Index Compression</td>
</tr>
<tr>
<td></td>
<td>Binary Versus Nonbinary Index Search</td>
</tr>
<tr>
<td></td>
<td>Secondary Index Buckets and Data Records (SIDRs)</td>
</tr>
<tr>
<td></td>
<td>Secondary Index Bucket Format</td>
</tr>
<tr>
<td></td>
<td>Secondary Index Data Record Format</td>
</tr>
<tr>
<td></td>
<td>Compression</td>
</tr>
<tr>
<td>Part 2. Simulated Data Example</td>
<td></td>
</tr>
</tbody>
</table>

# MODULE 6. RMS UTILITIES

<table>
<thead>
<tr>
<th>Part 3. Introduction</th>
<th>Analyzing File Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Measuring Run-Time Performance</td>
</tr>
<tr>
<td>Part 4. Evaluating/Utilizing</td>
<td></td>
</tr>
</tbody>
</table>
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 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

Topics
- Connection to VAX/VMS I/O system
- VAX RMS and multitasking primitives (VAX ML and VAX COER)
- VAX's memory management
- VAX's process management

Source:
VAX Reference Manual — Section I and II
- 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

<table>
<thead>
<tr>
<th>Structure</th>
<th>Function</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Access Block</td>
<td>Describes a file and contains file-related information</td>
<td>$FAB</td>
</tr>
<tr>
<td>Record Access Block</td>
<td>Describes a record and contains record-related information</td>
<td>$RAB</td>
</tr>
<tr>
<td>Extended Attribute Blocks</td>
<td>Contains file attribute information beyond that in the File Access Block or record-related information beyond that in the Record Access Block</td>
<td>$XABxxx</td>
</tr>
<tr>
<td>Name Block</td>
<td>Contains file specification information beyond that in the File Access Block</td>
<td>$SNAM</td>
</tr>
</tbody>
</table>
User Program and RMS Data Structures and Buffers

VIRTUAL ADDRESS SPACE

P0 SPACE
- USER DATA STRUCTURES
- USER PROGRAM IMAGE
- USER BUFFERS - RECORDS
- USER RMS CONTROL BLOCKS (FABs, RABs, XABs, NAMs)

P1 SPACE
- RMS INTERNAL DATA STRUCTURES
  INCLUDING: FABs, RABs
- RMS BUFFERS - MULTIBLOCKS OR BUCKETS

S0 SPACE
- RMS.EXE
- RMS SHARED FILE DATA STRUCTURES (GLOBAL 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 ISEGMENT=NOP0OBUNS. 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
BKS  = bucket-size
BLS  = tape block-size
CTX  = user-value
DEQ  = extension-qty
DEV  = device characteristics
DNA  = default filespec-address
DNS  = filespec-size
FAC  = file-access: <value>
FNA  = filespec-address
FNS  = filespec-size
FOP  = file-option: <value>
FSZ  = header-size

GBC  = global buffer count
MRN  = max-rec-number
MRS  = max-rec-size
NAM  = name address block
ORG  = file-organization
RAT  = record-attributes
RFM  = record-format: <value>
RTV  = window-size
SDC  = secondary device characteristics
SHR  = file-sharing: <value>
CTS  = completion status code
SV  = status value
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKT</td>
<td>bucket-code</td>
</tr>
<tr>
<td>CTX</td>
<td>user-value</td>
</tr>
<tr>
<td>FAB</td>
<td>fab-address</td>
</tr>
<tr>
<td>KBF</td>
<td>key-buffer-address</td>
</tr>
<tr>
<td>KRF</td>
<td>key reference number</td>
</tr>
<tr>
<td>KSZ</td>
<td>key-size</td>
</tr>
<tr>
<td>MBC</td>
<td>multiblock count</td>
</tr>
<tr>
<td>MBF</td>
<td>multibuffer count</td>
</tr>
<tr>
<td>PBF</td>
<td>prompt-buffer-address</td>
</tr>
<tr>
<td>PSZ</td>
<td>prompt-buffer-size</td>
</tr>
<tr>
<td>RAC</td>
<td>record-access-mode</td>
</tr>
<tr>
<td>RBT</td>
<td>record-buffer-address</td>
</tr>
<tr>
<td>RPA</td>
<td>record-file-address</td>
</tr>
<tr>
<td>RHB</td>
<td>header-buffer-address</td>
</tr>
<tr>
<td>ROP</td>
<td>record-options: &lt;value&gt;</td>
</tr>
<tr>
<td>RSZ</td>
<td>record-size</td>
</tr>
<tr>
<td>STS</td>
<td>completion status code</td>
</tr>
<tr>
<td>STV</td>
<td>completion value</td>
</tr>
<tr>
<td>TMO</td>
<td>seconds</td>
</tr>
<tr>
<td>UBF</td>
<td>user-buffer-address</td>
</tr>
<tr>
<td>USZ</td>
<td>user-buffer-size</td>
</tr>
<tr>
<td>XAB</td>
<td>next XAB address</td>
</tr>
</tbody>
</table>
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$XAB field in the FAB. This field contains the address of the first XAB to be used. Each successive XAB in the list links to the next using the FAB$XAB 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.
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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccc</td>
<td>3 letters</td>
<td>Identifies the type of control block: FAB, NAM, XAB (for all XABs), and RAB</td>
</tr>
<tr>
<td>$</td>
<td>1 character</td>
<td>Separator character; always a dollar sign ($)</td>
</tr>
<tr>
<td>x</td>
<td>1 letter</td>
<td>Identifies the length of the field:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• B for byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• W for word</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• L for longword</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Q for quadword</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• T for text buffer address</td>
</tr>
<tr>
<td></td>
<td>1 character</td>
<td>Separator character; always an underscore (_)</td>
</tr>
<tr>
<td>fff</td>
<td>3 or more letters</td>
<td>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).</td>
</tr>
</tbody>
</table>

Example

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

Field values involve four different naming conventions.

1. \texttt{xxxSC\_fff}

The first kind of symbolic field values are simple symbolic field values. These are identified by the presence of a \texttt{C} immediately following the block prefix in their name. For example, the \texttt{RABSB\_RAC} field has three symbolic values, one each for sequential, keyed, and RFA access modes. The symbolic names for these values are \texttt{RABSC\_SEQ}, \texttt{RABSC\_KEY}, and \texttt{RABSC\_RFA}. These symbolic field values are used in simple assignment statements.

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

2. \texttt{xxxSM\_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 \texttt{M} immediately following the block prefix in their name. For example, the \texttt{FABSL\_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.

\begin{itemize}
  \item \texttt{FABSM\_CBT} Contiguous 'best try'
  \item \texttt{FABSM\_CTG} Contiguous
  \item \texttt{FABSM\_TEF} Truncate at end of file
\end{itemize}

The \texttt{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. \texttt{xx\_V\_fff} -- bit offset

4. \texttt{xxxSS\_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 \texttt{S} and \texttt{V} values immediately following the block prefix in their names. The \texttt{S} form of the name defines the size of that flag field (usually the value 1 for single bit flag fields), and the \texttt{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 SYSS followed by three or more characters. In the corresponding VAX MACRO macro name, the SYSS prefix is not used. For example, the Create service has an entry point of SYSSCREATE and a VAX MACRO macro name of $CREATE.

#### RMS Services

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSSCLOSE</td>
<td>$CLOSE</td>
<td>Terminates file processing and disconnects all record streams</td>
</tr>
<tr>
<td>SYSSCREATE</td>
<td>$CREATE</td>
<td>Creates and opens a new file of any organization</td>
</tr>
<tr>
<td>SYSSDISPLAY</td>
<td>$DISPLAY</td>
<td>Returns the attributes of an open file to the user program</td>
</tr>
<tr>
<td>SYS$ENTER*</td>
<td>$ENTER</td>
<td>Enters a file name into a directory</td>
</tr>
<tr>
<td>SYSS$ERASE</td>
<td>$ERASE</td>
<td>Deletes a file and removes its directory entry</td>
</tr>
<tr>
<td>SYSS$EXTEND</td>
<td>$EXTEND</td>
<td>Extends the allocated space of a file</td>
</tr>
<tr>
<td>SYSS$OPEN</td>
<td>$OPEN</td>
<td>Opens an existing file and initiates file processing</td>
</tr>
<tr>
<td>SYSS$PARSE</td>
<td>$PARSE</td>
<td>Parses a file specification</td>
</tr>
<tr>
<td>SYSS$RENAME*</td>
<td>$RENAME</td>
<td>Removes a file name from a directory</td>
</tr>
<tr>
<td>SYSS$SEARCH</td>
<td>$SEARCH</td>
<td>Searches a directory, or possibly multiple directories, for a file name</td>
</tr>
</tbody>
</table>

1-12
### RMS Services (Cont.)

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

* This service is not supported for DECnet operations involving remote file access between two VAX/VMS systems.
### RMS Services (Cont.)

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Macro Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block I/O Processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYS$READ</td>
<td>$READ</td>
<td>Retrieves a specified number of bytes from a file, beginning on block boundaries</td>
</tr>
<tr>
<td>SYS$SPACE</td>
<td>$SPACE</td>
<td>Positions forward or backward in a file to a block boundary</td>
</tr>
<tr>
<td>SYS$WRITE</td>
<td>$WRITE</td>
<td>Writes a specified number of bytes to a file, beginning on block boundaries</td>
</tr>
<tr>
<td>Procedure</td>
<td>Operation</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>FDL$CREATE</td>
<td>Creates a file from an FDL specification and then closes the file.</td>
<td></td>
</tr>
<tr>
<td>FDL$GENERATE</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>FDL$PARSE</td>
<td>Parses an FDL specification, allocates RMS control blocks, and then fills in the relevant fields.</td>
<td></td>
</tr>
<tr>
<td>FDL$RELEASE</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
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 3 (Sections 4.1, 4.2, 4.4)
VAX/VMS File Definition Language Facility 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

EDIT/FDL

CONVERT

INDEXED FILE

SEQUNETIAL FILE

ANALYZE/RMS_FILE

CONVERT/RECLAIM
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`
EDIT/FDL Scripts

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 ~2 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] : ?

2-3
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 file 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 indxbckfdl

Parsing Definition File

DISK$INSTRUCTOR: [WOODS.RMS.COURSE] INDBACK.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] :
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Segmentation desired</td>
<td>(Yes/No)[No] : No</td>
</tr>
<tr>
<td>0</td>
<td>Length</td>
<td>(1-50)[-] : 5</td>
</tr>
<tr>
<td>0</td>
<td>Position</td>
<td>(0-45)[0] : 0</td>
</tr>
<tr>
<td>0</td>
<td>Duplicates allowed</td>
<td>(Yes/No)[No] : No</td>
</tr>
<tr>
<td></td>
<td>File Prolog Version</td>
<td>(0-3)[3] : n</td>
</tr>
<tr>
<td></td>
<td>Data Key Compression desired</td>
<td>(Yes/No)[Yes] : n</td>
</tr>
<tr>
<td></td>
<td>Data Record Compression desired</td>
<td>(Yes/No)[Yes] : n</td>
</tr>
</tbody>
</table>

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 PORTTRAN None Print)

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: 3 3 12

Pages Required to Cache Index: 112 112 453

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)

(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  nq
DATA_RECORD_COMPRESSION  no
DUPLICATES  no
INDEX_AREA  1
INDEX_COMPRESSION  no
INDEX_FILL  100
LEVEL 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] IXDBACK.FDL;1 44 lines
TITLE "FDL FOR INDEX BACKWARDS"
IDENT "14-JAN-1986 13:14:58  VAX-11 FDL Editor"
SYSTEM SOURCE VAX/VMS
FILE
NAME "INDEXBACK.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
LEVEL_INDEX_AREA 1
NAME "SEQ_NO"
PROLOG 3
SEQ0_LENGTH 5
SEQ0_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

FDLS$CREATE (fdl-str, [file-spec-str], [default-name],
[result-name], [FID-bid], [flags],
[stmt_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 Ctrl List: None

Total of 1 file. 1.4/184 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$PASSFILES (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 Keys 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

2-12
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)

INDEX

DEPTH

1
2
3

BUCKET SIZE (NUMBER OF BLOCKS)

PV-PROLOG VERSION 3
DK-DUP KEY 0 VALUES NO
RC-DATA RECORD COMP 0%
BF-BUCKET FILL 100%
LM-LOAD METHOD FAST_CONV
WHICH FILE PARAMETER

(1-126 chars)[null]

Text for FDL Title Section : fdl for index1

Data File file-spec : indxl.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] Field Size (1-63) 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 (~2 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)

BUCKET SIZE (NUMBER OF BLOCKS)

INDEX

DEPTH

PV-PROLOG VERSION 3
DK-DUP KEY 0 VALUES YES
RC-DATA RECORD COMP 0%
BF-BUCKET PULL 100%
LM-LOAD METHOD FAST_CONV
(TYPE "FD" TO FINISH DESIGN)
WHICH FILE PARAMETER WHICH FILE PARAMETER

KT-KEY 1 TYPE STRING EM-EMPHASIS FLATTER (3)
KL-KEY 1 LENGTH 15 KP-KEY 1 POSITION 7
KC-DATA KEY COMP 0% IC-INDEX RECORD COMP 0%
RF-RECORD FORMAT FIXED RS-RECORD SIZE 80
IL-INITIAL LOAD 500 AR-ADDED RECORDS 0

(MNEMONIC)[REFRESH] (MNEMONIC)[REFRESH]

FD

BU-2416

2-14
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 index1"
IDENT "14-JAN-1986 13:35:55 VAX-11 FDL Editor"
SYSTEM SOURCE VAX/VMS
FILE NAME "index1.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
LEVELI_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
LEVELI_INDEX_AREA 3
NAME "last_name"
SEG0_LENGTH 15
SEG0_POSITION 7
TYPE string

Press RETURN to continue (~2 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

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

INDEX

DEPTH

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

BUCKET SIZE (NUMBER OF BLOCKS)

Evaluating Line Plot

Breakpoints

3 to 2

2 to 1

# Blocks/Bucket

3

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] :

2-19
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] : Y
Data Record Compression desired (Yes/No)[Yes] : Y
Index Compression desired (Yes/No)[Yes] : Y

```
WORKING...

<table>
<thead>
<tr>
<th></th>
<th>86</th>
<th>80</th>
<th>74</th>
<th>68</th>
<th>62</th>
<th>56</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

BUCKET SIZE (NUMBER OF BLOCKS)

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

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

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

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

BU-2419
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] :

2-21
Key 0 Segmentation desired (Yes/No) [No] :

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

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] :

BUCKET SIZE (NUMBER OF BLOCKS)

<table>
<thead>
<tr>
<th>WORKING...</th>
<th>INITIAL</th>
<th>LOAD</th>
<th>RECORD</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10006</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
<tr>
<td>8440</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
<tr>
<td>7872</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
<tr>
<td>6504</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
<tr>
<td>5336</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
<tr>
<td>4168</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
<tr>
<td>3000</td>
<td>3333333</td>
<td>3/2</td>
<td>22</td>
<td>22222</td>
</tr>
</tbody>
</table>

0001 : 1-(1-1333-1)
0002 : (1-1333-1)
0003 : 

BU-22420
(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 (^2 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] :

<table>
<thead>
<tr>
<th>INDEX</th>
<th>DEPTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BUCKET SIZE (NUMBER OF BLOCKS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

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

Which FILE PARAMETER (MNEMONIC)[REFRESH] : KL KEY 0 LENGTH (1-255)[-] : 75

<table>
<thead>
<tr>
<th>INDEX</th>
<th>DEPTH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BUCKET SIZE (NUMBER OF BLOCKS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

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

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] :

2-25
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
- **SEGO_LENGTH**: 75
- **SEGO_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"
- **SEGO_LENGTH**: 75
- **SEGO_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 1
Overview of MS-DOS
Online Disk File Structure

Table of Contents
- Disk File Structure
- File Allocation
- Disk Protection
- Disk Operations
- DOS Command

Source: MS-DOS, The Programming Interface (Microsoft Press, 1985)
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 (initialize)

Index File (INDEXF.SYS) # set volume # show device / null

Storage Bit Map File (BITMAP.SYS)

Bad Block File (BADBLK.SYS)

Pending Bad Block Log File (BADLOG.SYS)

Master File Directory (000000.DIR)

Core Image File (CORIMG.SYS)

Volume Set List File (VOLSET.SYS)

Continuation File (CONTIN.SYS)

Backup Log File (BACKUP.SYS)

Information

Bootstrap block, home block, back-up homr block, back-up index file header, index file bit map, file headers

Location of available clusters on a volume

Areas of volume not suitable for use

Areas of volume suspected of being unsuitable for use

Pointers to all User File Directories

Not used by VMS. Provided to preserve structure of previous versions

Labels of other volumes in the volume set (if more than one)

The extension file identifier, if a file crosses from one volume to another in a loosely coupled volume set

History of backups on the volume
Comparing the ODS-1 and ODS-2 Structures

ODS-1 and ODS-2 Comparison

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

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 format 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.
- 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

```
<table>
<thead>
<tr>
<th>15</th>
<th>07</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECORD BYTE COUNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERSION LIMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAME BYTE COUNT</td>
<td>FLAGS</td>
<td></td>
</tr>
<tr>
<td>FILE NAME STRING (UPPERCASE ONLY)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERSION NUMBER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILE ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

REPEATED FOR EACH VERSION. STORED IN DESCENDING ORDER.
Using ODS-2 File Headers to Access a File

- INDEX FILE
  - FILE HEADER 1
    - INDEXF.SYS
  - FILE HEADER 2
    - BITMAP.SYS
  - FILE HEADER 3
    - BADBLK.SYS
  - FILE HEADER 4
    - 000000.DIR

- FILE HEADER 1
  - COURSE.DIR

- FILE HEADER J
  - TEST.DAT

- MFD DATA BLOCKS
  - 000000.DIR
  - COURSE.DIR
  - JONES.DIR
  - SYSMGR.DIR

- UFD [COURSE] DATA BLOCKS
  - TEST.DAT

- FILE TEST.DAT DATA BLOCKS

# Each file header has a number used for locating file data.
# &w$_full &w$_id ?

\[(\text{\texttt{m}}, \text{\texttt{n}}, \text{\texttt{p}})\]

- file_header_no
- vol_no. in multi-volume set

no of times this file header not used
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)

- Used to locate file data blocks on the disk.
DISK PHYSICAL CHARACTERISTICS

A track is comprised of the area at a single radius on one recording surface.

A cylinder consists of these tracks in the same radius on all the recording surfaces.

Recording occurs on both surfaces of each platter. The extreme top and bottom surfaces of some disk models are not used for recording.

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.

NOTE

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.
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:

<table>
<thead>
<tr>
<th>Disk</th>
<th>Track-Size</th>
<th>Cluster-Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA80</td>
<td>31</td>
<td>1, 31</td>
</tr>
<tr>
<td>RA81</td>
<td>51</td>
<td>1, 3, 17, 51</td>
</tr>
<tr>
<td>RM05</td>
<td>32</td>
<td>1, 2, 4, 8, 16, 32</td>
</tr>
<tr>
<td>RP07</td>
<td>50</td>
<td>1, 2, 5, 10, 25, 50</td>
</tr>
</tbody>
</table>
Relationship Between Blocks, Clusters, and Extents of a Typical Disk File

Relative to start of file

LOGICAL BLOCK 221 (LB 221)

VIRTUAL BLOCK 1 (VB 1)

LB 222  VB 2

LB 223  VB 3

LB 224  VB 4

LB 225  VB 5

LB 226  VB 6

EXTENT 1

CLUSTERSIZE = 3

EXTENT 2

FILE DATA BLOCKS

LB 405  VB 7

LB 406  VB 8

LB 407  VB 9
$ SHOW DEV/FULL DISK$STUDENT (to determine disk cluster size)

Disk $15DUA2: (BUD), device type RA81, is online, mounted, file-oriented device, shareable, available to cluster, error logging is enabled.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error count</td>
<td>0</td>
</tr>
<tr>
<td>Owner process</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Owner process ID</td>
<td>00000000</td>
</tr>
<tr>
<td>Reference count</td>
<td>2</td>
</tr>
<tr>
<td>Total blocks</td>
<td>891072</td>
</tr>
<tr>
<td>Total cylinders</td>
<td>1248</td>
</tr>
<tr>
<td>Host name</td>
<td>&quot;BUD&quot;</td>
</tr>
<tr>
<td>Alternate host name</td>
<td>&quot;LOU&quot;</td>
</tr>
<tr>
<td>Allocation class</td>
<td>1</td>
</tr>
<tr>
<td>Operations completed</td>
<td>186763</td>
</tr>
<tr>
<td>Owner UIC</td>
<td>[1,1]</td>
</tr>
<tr>
<td>Dev Prot</td>
<td>S:RWED,O:RWED,G:RWED,W:RWED</td>
</tr>
<tr>
<td>Default buffer size</td>
<td>512</td>
</tr>
<tr>
<td>Sectors per track</td>
<td>51</td>
</tr>
<tr>
<td>Tracks per cylinder</td>
<td>14</td>
</tr>
<tr>
<td>Host type, available</td>
<td>HS50, yes</td>
</tr>
<tr>
<td>Alternate host type, avail</td>
<td>HS50, yes</td>
</tr>
<tr>
<td>Volume label</td>
<td>&quot;STUDENT_COM&quot;</td>
</tr>
<tr>
<td>Relative volume number</td>
<td>0</td>
</tr>
<tr>
<td>Cluster size</td>
<td>3</td>
</tr>
<tr>
<td>Free blocks</td>
<td>618132</td>
</tr>
<tr>
<td>Extend quantity</td>
<td>6</td>
</tr>
<tr>
<td>Mount status</td>
<td>System</td>
</tr>
<tr>
<td>Extent cache size</td>
<td>63</td>
</tr>
<tr>
<td>File ID cache size</td>
<td>63</td>
</tr>
<tr>
<td>Quota cache size</td>
<td>0</td>
</tr>
<tr>
<td>Transaction count</td>
<td>1</td>
</tr>
<tr>
<td>Maximum files allowed</td>
<td>148512</td>
</tr>
<tr>
<td>Mount count</td>
<td>4</td>
</tr>
<tr>
<td>Cache name</td>
<td>&quot;$15DUA0:XXPCACHE&quot;</td>
</tr>
<tr>
<td>Maximum blocks in extent cache</td>
<td>61813</td>
</tr>
<tr>
<td>Blocks currently in extent cache</td>
<td>28014</td>
</tr>
<tr>
<td>Maximum buffers in FCP cache</td>
<td>311</td>
</tr>
</tbody>
</table>

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

FILE HEADER (SIZE = 1 BLOCK)

1. HEADER
2. IDENTIFICATION
3. MAP POINTERS
4. ACCESS
5. RESERVED
6. CHECKSUM

FILE DATA BLOCKS

LB 221 VB 1
LB 226 VB 6
LB 405 VB 7
LB 406 VB 8
LB 407 VB 9

EXTENT 1
EXTENT 2

CLUSTERSIZE = 3

TK-7986

3-10
### 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

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

#### Identification area

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>File name:</td>
<td>FILE.TXT;1</td>
</tr>
<tr>
<td>Revision number:</td>
<td>2</td>
</tr>
<tr>
<td>Creation date:</td>
<td>19-JUN-1984 09:30:53.40</td>
</tr>
<tr>
<td>Revision date:</td>
<td>19-JUN-1984 09:30:53:90</td>
</tr>
<tr>
<td>Expiration date:</td>
<td>&lt;none specified&gt;</td>
</tr>
<tr>
<td>Backup date:</td>
<td>&lt;none specified&gt;</td>
</tr>
</tbody>
</table>

#### Map area

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>6</td>
</tr>
<tr>
<td>LBN</td>
<td>147399</td>
</tr>
<tr>
<td></td>
<td>213246</td>
</tr>
<tr>
<td>Checksum:</td>
<td>5051</td>
</tr>
</tbody>
</table>

3-11
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
MODULE 4
OVERVIEW OF RPS ORGANIZATIONS
RECORD FORMATS AND ACCESS METHODS

[Text continues on the next page]
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

<table>
<thead>
<tr>
<th>RECORD 1</th>
<th>RECORD 2</th>
<th>RECORD 3</th>
<th>…</th>
<th>RECORD n</th>
</tr>
</thead>
</table>

4-1
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

<table>
<thead>
<tr>
<th>CELL #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECORD</td>
<td>RECORD</td>
<td>RECORD</td>
<td>EMPTY</td>
<td>RECORD</td>
<td>...</td>
<td>RECORD</td>
<td></td>
</tr>
</tbody>
</table>

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 (PLGSL_MRN) and the field containing the end-of-file block number (PLGSL_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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLG$B_DBKTSIZ</td>
<td>RESERVED</td>
<td>11</td>
</tr>
<tr>
<td>PLG$B_FLAGS</td>
<td>RESERVED</td>
<td>85</td>
</tr>
<tr>
<td>PLG$B_AMAX</td>
<td>PLG$B_AVBN</td>
<td></td>
</tr>
<tr>
<td>PLG$W_DVBN</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>PLG$L_MRN</td>
<td>PLG$L_EOF</td>
<td></td>
</tr>
<tr>
<td>PLG$W_GBC</td>
<td>PLG$W_VER_NO</td>
<td></td>
</tr>
<tr>
<td>CHECK SUM</td>
<td>RESERVED</td>
<td>390</td>
</tr>
</tbody>
</table>
## Contents of a Record Prolog for a Relative File

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLGSB_DBKTSIZ</td>
<td>Data bucket size. This field is not used for relative files. The FAT$B_BTKSIZ$ field defines the bucket size for a relative file.</td>
</tr>
<tr>
<td>PLGSB_FLAGS</td>
<td>Flag bits. This field contains a bit vector specifying characteristics of the file that this prolog defines. The following field is defined within PLGSB_FLAGS.</td>
</tr>
<tr>
<td></td>
<td>PLGSV_NOEXTEND If set, the file cannot be extended. This field is 1 bit long, and starts at bit 0.</td>
</tr>
<tr>
<td>PLGSB_AVBN</td>
<td>This field is not used for relative files.</td>
</tr>
<tr>
<td>PLGSB_AMAX</td>
<td>This field is not used for relative files.</td>
</tr>
<tr>
<td>PLGSW_DVBN</td>
<td>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.</td>
</tr>
<tr>
<td>PLGS_L_MRN</td>
<td>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)*.</td>
</tr>
<tr>
<td></td>
<td>This field can be set with the RMS field FAB$S_L_MRN and the FDL attribute FILE MAX_RECORD_NUMBER.</td>
</tr>
<tr>
<td>PLGS_L_EOF</td>
<td>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.</td>
</tr>
<tr>
<td>PLGSW_VER_NO</td>
<td>This field is not used for relative files.</td>
</tr>
<tr>
<td>PLGSW_GBC</td>
<td>This field is not used for relative files.</td>
</tr>
<tr>
<td>Checksum</td>
<td>Additive checksum.</td>
</tr>
</tbody>
</table>

* 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.

<table>
<thead>
<tr>
<th>DATA</th>
<th>CTRL BYTE</th>
</tr>
</thead>
</table>

CELL SIZE = MAXIMUM RECORD SIZE + 1

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**

<table>
<thead>
<tr>
<th>File Organization</th>
<th>Fixed</th>
<th>Variable</th>
<th>VFC</th>
<th>Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Relative</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Indexed</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

---

4-6
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)

<table>
<thead>
<tr>
<th>Record Access Method</th>
<th>File Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Sequential, Relative, Indexed</td>
</tr>
<tr>
<td>Random by relative record number</td>
<td>Yes*, Yes, No</td>
</tr>
<tr>
<td>Random by key value</td>
<td>No, Yes, No</td>
</tr>
<tr>
<td>Random by record's file address</td>
<td>Yes**, Yes, Yes</td>
</tr>
</tbody>
</table>

* 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:

```
RFA

+-----------+-----------+
| 4 BYTES   | 2 BYTES   |
+-----------+-----------+

SEQUENTIAL VBN # | BYTE OFFSET |
RELATIVE         |             |
INDEXED          |             |
```  

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

<table>
<thead>
<tr>
<th>Record Operation</th>
<th>File Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read (Get)</td>
<td>Sequential, Relative, Indexed</td>
</tr>
<tr>
<td>Write (Put)</td>
<td>Yes*</td>
</tr>
<tr>
<td>Find</td>
<td>Yes**</td>
</tr>
<tr>
<td>Update***</td>
<td>Yes***</td>
</tr>
</tbody>
</table>

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

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

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

** Empty buckets (all recor's 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 (PLGSB_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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESERVED (11 BYTES)</td>
<td></td>
</tr>
<tr>
<td>PLGSB_DBKTSIZ</td>
<td>Contains info about 1st key</td>
</tr>
<tr>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>RESERVED (85 BYTES)</td>
<td></td>
</tr>
<tr>
<td>PLGSB_AMAX</td>
<td></td>
</tr>
<tr>
<td>PLGSB_AVBN</td>
<td></td>
</tr>
<tr>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>PLGW_DVBI</td>
<td></td>
</tr>
<tr>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>PLGW_GBC</td>
<td></td>
</tr>
<tr>
<td>PLGW_VER_NO</td>
<td></td>
</tr>
<tr>
<td>RESERVED (390 BYTES)</td>
<td></td>
</tr>
<tr>
<td>CHECK SUM</td>
<td></td>
</tr>
</tbody>
</table>
### Contents of a Record Prolog for an Indexed Sequential File

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

- **PROLOG DESCRIPTOR**
  - PRIMARY KEY DESCRIPTOR
  - PLG$B_AMAX
  - PLG$B_AVBN

  VBN 1

- **AREA DESCRIPTORS**

  VBN 2

- **INDEX AND DATA BUCKETS**

  VBN 3 - N

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.

\[ \text{PLG\$B\_AMAX} + \text{PLG\$B\_AVBN} \]

8 (truncated)

**Format of a Prolog for a File with Multiple Keys**

```
PROLOG DESCRIPTOR
  PRIMARY KEY DESCRIPTOR
      PLG\$B\_AMAX  PLG\$B\_AVBN

UP TO 5 KEY DESCRIPTORS
  KEY 5 DESCRIPTOR

KEY DESCRIPTORS (IF MORE THAN 5)

AREA DESCRIPTORS

INDEX AND DATA BUCKETS
```
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 KEY$B_IANUM, KEY$B_LANUM, and KEY$B_DANUM fields.
### Format of a Key Descriptor

<table>
<thead>
<tr>
<th>KEY$B_LANUM</th>
<th>KEY$B_JANUM</th>
<th>KEY$W_NOFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY$BDATBKTZ</td>
<td>KEY$BIDXBKTSZ</td>
<td>KEY$B_ROOTLEV</td>
</tr>
<tr>
<td>KEY$B_ROOTVBN</td>
<td>KEY$B_ROOTVBN</td>
<td>KEY$B_ROOTVBN</td>
</tr>
<tr>
<td>KEY$B_NULLCHAR</td>
<td>KEY$B_SEGMENTS</td>
<td>KEY$B_DATATYPE</td>
</tr>
<tr>
<td>KEY$W_MINRECSZ</td>
<td>KEY$B_KEYREF</td>
<td>KEY$B_KEYSZ</td>
</tr>
<tr>
<td>KEY$W_DATFILL</td>
<td>KEY$WIDXFILL</td>
<td>KEY$WIDXFILL</td>
</tr>
<tr>
<td>KEY$W_POSITION1</td>
<td>KEY$W_POSITION</td>
<td></td>
</tr>
<tr>
<td>KEY$W_POSITION3</td>
<td>KEY$W_POSITION2</td>
<td></td>
</tr>
<tr>
<td>KEY$W_POSITION5</td>
<td>KEY$W_POSITION4</td>
<td></td>
</tr>
<tr>
<td>KEY$W_POSITION7</td>
<td>KEY$W_POSITION6</td>
<td></td>
</tr>
<tr>
<td>KEY$B_SIZE3</td>
<td>KEY$B_SIZE2</td>
<td></td>
</tr>
<tr>
<td>KEY$B_SIZE6</td>
<td>KEY$B_SIZE5</td>
<td></td>
</tr>
<tr>
<td>KEY$B_SIZE7</td>
<td>KEY$B_SIZE4</td>
<td></td>
</tr>
<tr>
<td>KEY$T_KEYNAM (32 BYTES)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KEY$W_LOADBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY$B_TYPE3</td>
</tr>
<tr>
<td>KEY$B_TYPE2</td>
</tr>
<tr>
<td>KEY$B_TYPE1</td>
</tr>
<tr>
<td>KEY$B_TYPE</td>
</tr>
<tr>
<td>KEY$B_TYPE7</td>
</tr>
<tr>
<td>KEY$B_TYPE6</td>
</tr>
<tr>
<td>KEY$B_TYPE5</td>
</tr>
<tr>
<td>KEY$B_TYPE4</td>
</tr>
</tbody>
</table>

* 96 bytes

* at most 5 to a block

* Prolog holds 1st Key Descriptor
### Contents of a Key Descriptor

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY$B_ROOTLEV</td>
<td>Root level. This field contains the level number of the root bucket for the key. In other words, this field contains the height of the index tree. Levels are numbered from 0 to n, where 0 indicates the data level and n indicates the root level. This field sets the RMS field XAB$B_LVL after an Open or Display service.</td>
</tr>
<tr>
<td>KEY$B_IDXBKTSZ</td>
<td>Index bucket size. This field contains the size (number of virtual blocks) of the index-level buckets (level 1 to n) for the key. This field sets the RMS field XAB$B_IBS after an Open or Display service.</td>
</tr>
<tr>
<td>KEY$B_DATBKTSZ</td>
<td>Data bucket size. This field contains the size (number of virtual blocks) of the data-level buckets (level 0) for the key. This field sets the RMS field XAB$B_DBS after an Open or Display service.</td>
</tr>
<tr>
<td>KEY$L_ROOTVBN</td>
<td>VBN of the root bucket. Contains the virtual block number of the index root bucket for the key. After an Open or Display service, this field sets the RMS field XAB$L_RVB.</td>
</tr>
<tr>
<td>KEY$B_FLAGS</td>
<td>Flag bits. This field may be set with the RMS field XAB$B_FLG. The following 1-bit fields are defined within KEY$B_FLAGS.</td>
</tr>
<tr>
<td>KEY$V_DUPKEYS</td>
<td>Set if duplicate key values are allowed. This field starts at bit 0. It may be set with the RMS field XAB$V_DUP.</td>
</tr>
<tr>
<td>KEY$V_CHGKEYS</td>
<td>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$V_CHG.</td>
</tr>
<tr>
<td>KEY$V_NULKEYS</td>
<td>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.</td>
</tr>
</tbody>
</table>
## Contents of a Key Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY$B_FLAGS</td>
<td><strong>KEY$V_IDX_COMPR</strong> Set if the index is compressed. This field starts at bit 3. This field may be cleared with the RMS field XAB$V_IDX_NCMPR.</td>
</tr>
<tr>
<td>(Cont.)</td>
<td><strong>KEY$V_INITIDX</strong> Set if the index must be initialized. This field is used only when RMS creates the index for this key. Because area numbers 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.</td>
</tr>
<tr>
<td></td>
<td><strong>KEY$V_KEY_COMPR</strong> Set if the key is compressed in data records. The key must be a Prolog 3 string key. This field starts at bit 6. This field may be overridden with the RMS field XAB$V_KEY_NCMPR.</td>
</tr>
<tr>
<td></td>
<td><strong>KEY$V_REC_COMPR</strong> Set if the data portion of the record is compressed. This bit applies only to Prolog 3 files. This field starts at bit 7. This field may be overridden with the RMS field XAB$V_DAT_NCMPR.</td>
</tr>
</tbody>
</table>
Contents of a Key Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYSB_DATATYPE</td>
<td>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 XABSB_DTP. The following constants may be specified.</td>
</tr>
<tr>
<td>KEYSB_STRING</td>
<td>Left-justified string of unsigned, 8-bit bytes. This is the default.</td>
</tr>
<tr>
<td>KEYSB_SGNWORD</td>
<td>Signed binary word.</td>
</tr>
<tr>
<td>KEYSB_UNSGNWORD</td>
<td>Unsigned binary word.</td>
</tr>
<tr>
<td>KEYSB_SGNLONG</td>
<td>Signed binary longword.</td>
</tr>
<tr>
<td>KEYSB_UNSGNLONG</td>
<td>Unsigned binary longword.</td>
</tr>
<tr>
<td>KEYSB_PACKED</td>
<td>Packed decimal string.</td>
</tr>
<tr>
<td>KEYSB_SGNQUAD</td>
<td>Signed binary quadword.</td>
</tr>
<tr>
<td>KEYSB_UNSGNQUAD</td>
<td>Unsigned binary quadword.</td>
</tr>
<tr>
<td>KEYSB_DSTRING</td>
<td>String data type for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td>KEYSB_DSGNWORD</td>
<td>Signed binary word data type for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td>KEYSB_DUNSGNWORD</td>
<td>Unsigned binary word data type for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td>KEYSB_DSGNLONG</td>
<td>Signed binary longword data type for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td>KEYSB_DUNSGNLONG</td>
<td>Unsigned binary longword data type for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td>KEYSB_DPACKED</td>
<td>Packed decimal string data type for descending keys (as of VMS 4.4).</td>
</tr>
</tbody>
</table>
### Contents of a Key Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>KEY$B_DATATYPE</strong> (Cont.)</td>
<td><strong>KEY$C_DSGNQUAD</strong> Signed binary quadword for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td></td>
<td><strong>KEY$C_DUNSGNQUAD</strong> Unsigned binary quadword for descending keys (as of VMS 4.4).</td>
</tr>
<tr>
<td><strong>KEY$B_SEGMENTS</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>After an Open or Display service, this field sets the RMS field XAB$B_NSNG.</td>
</tr>
<tr>
<td><strong>KEY$B_NULLCHAR</strong></td>
<td>Null character. This field contains any user-selected ASCII character value.</td>
</tr>
<tr>
<td></td>
<td>This field is set with the RMS field XAB$B_NUL.</td>
</tr>
<tr>
<td><strong>KEY$B_KEYSZ</strong></td>
<td>Total key size. This field contains the sum (in bytes) of the values in the fields KEY$B_SIZE through KEY$B_SIZ7.</td>
</tr>
<tr>
<td></td>
<td>After an Open or Display service, this field sets the RMS field XAB$B_TKS.</td>
</tr>
<tr>
<td><strong>KEY$B_KEYREF</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>This field is set with the RMS field XAB$B_REF.</td>
</tr>
<tr>
<td><strong>KEY$W_MINRECSZ</strong></td>
<td>Minimum record length. This field contains the minimum length (in bytes) needed to hold the key.</td>
</tr>
<tr>
<td></td>
<td>After an Open or Display service, this field sets the RMS field XAB$W_MRL.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| KEYSW_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 XABSW_IFL. |
| KEYSW_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 XABSW_DFL. |
| KEYSW_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 XABSW_POS0. |
| KEYSW_POSITION1     | Position 1. This field marks the beginning position of the second key segment. It is set with the RMS field XABSW_POS1. |
| KEYSW_POSITION2     | Position 2. This field marks the beginning position of the third key segment. It is set with the RMS field XABSW_POS2. |
| KEYSW_POSITION3     | Position 3. This field marks the beginning position of the fourth key segment. It is set with the RMS field XABSW_POS3. |
| KEYSW_POSITION4     | Position 4. This field marks the beginning position of the fifth key segment. It is set with the RMS field XABSW_POS4. |
| KEYSW_POSITION5     | Position 5. This field marks the beginning position of the sixth key segment. It is set with the RMS field XABSW_POS5. |
| KEYSW_POSITION6     | Position 6. This field marks the beginning position of the seventh key segment. It is set with the RMS field XABSW_POS6. |
| KEYSW_POSITION7     | Position 7. This field marks the beginning position of the eighth (and last) key segment. It is set with the RMS field XABSW_POS7. |
### Contents of a Key Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY$B_SIZE</td>
<td>Segment size. This field contains the size of the first key segment. It is set with the RMS field XAB$B_SIZE0.</td>
</tr>
<tr>
<td>KEY$B_SIZE1</td>
<td>Size 1. This field contains the size of the second key segment. It is set with the RMS field XAB$B_SIZE1.</td>
</tr>
<tr>
<td>KEY$B_SIZE2</td>
<td>Size 2. This field contains the size of the third key segment. It is set with the RMS field XAB$B_SIZE2.</td>
</tr>
<tr>
<td>KEY$B_SIZE3</td>
<td>Size 3. This field contains the size of the fourth key segment. It is set with the RMS field XAB$B_SIZE3.</td>
</tr>
<tr>
<td>KEY$B_SIZE4</td>
<td>Size 4. This field contains the size of the fifth key segment. It is set with the RMS field XAB$B_SIZE4.</td>
</tr>
<tr>
<td>KEY$B_SIZE5</td>
<td>Size 5. This field contains the size of the sixth key segment. It is set with the RMS field XAB$B_SIZE5.</td>
</tr>
<tr>
<td>KEY$B_SIZE6</td>
<td>Size 6. This field contains the size of the seventh key segment. It is set with the RMS field XAB$B_SIZE6.</td>
</tr>
<tr>
<td>KEY$B_SIZE7</td>
<td>Size 7. This field contains the size of the eighth key segment. It is set with the RMS field XAB$B_SIZE7.</td>
</tr>
<tr>
<td>KEY$T_KEYNAM</td>
<td>Optional key name. This 32-byte field contains the name of the key as an ASCII string. It is set with the RMS field XACL_KNM.</td>
</tr>
<tr>
<td>KEY$L_LDVBN</td>
<td>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.</td>
</tr>
</tbody>
</table>
## Contents of a Key Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYSB_TYPE1</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
<tr>
<td>KEYSB_TYPE2</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
<tr>
<td>KEYSB_TYPE3</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
<tr>
<td>KEYSB_TYPE4</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
<tr>
<td>KEYSB_TYPE5</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
<tr>
<td>KEYSB_TYPE6</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
<tr>
<td>KEYSB_TYPE7</td>
<td>This field is not supported by VAX/VMS Version 4.4.</td>
</tr>
</tbody>
</table>
Indexed File With and Without Areas

WITHOUT AREAS

START OF FILE

PROLOGUE

WITH AREAS

PRIMARY INDEX

DATA RECORDS

ALTERNATE INDEX

PI = PRIMARY INDEX
DR = DATA RECORDS
AI = ALTERNATE INDEX
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

<table>
<thead>
<tr>
<th>AREA$B_ARBKTSZ</th>
<th>AREA$B_AOID</th>
<th>AREA$B_FLAGS (Reserved)</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA$B_AOP</td>
<td>AREA$B_ALN</td>
<td>AREA$W_VOLUME</td>
<td></td>
</tr>
<tr>
<td>AREA$L_AVAIL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_CVBN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_CNBLK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_USED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_NXTVBN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_NXT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_NXBLK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESERVED</td>
<td>AREA$W_DEQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_LOC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$L_TOTAL_ALLOC</td>
<td>3 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$W_RFI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AREA$W_CHECK</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**not boundary aligned**
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREASB_AREAID</td>
<td>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$B_AID or the FDL attribute AREA n, where n indicates the area number.</td>
</tr>
<tr>
<td>AREASB_ARBKTSZ</td>
<td>Area bucket size. This field contains the bucket size for the area, which can range from 1 to 63 blocks. It represents the granularity of the allocation. This field is set by the RMS field XAB$B_BKZ and the FDL attribute AREA BUCKET_SIZE.</td>
</tr>
<tr>
<td>AREASW_VOLUME</td>
<td>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$W_VOL or the FDL attribute AREA VOLUME.</td>
</tr>
<tr>
<td>AREASB_ALN</td>
<td>Extent allocation alignment. This field indicates the type of alignment for the area to be allocated. It allows placement control to be specified for the file. The following options are valid for this field. If no value is set for this field, RMS assumes that placement control was not requested.</td>
</tr>
<tr>
<td>AREASC_CYL</td>
<td>This option indicates that the alignment starts at the specified cylinder number. It is set with the RMS field XAB$L_LOC or the FDL attribute AREA POSITION CYLINDER.</td>
</tr>
</tbody>
</table>

5-22
### Contents of an Area Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREASB_ALN</td>
<td>This option indicates that the alignment starts at the specified logical block. It may be set with the RMS field XAB$S_L_LOC or the FDL attribute AREA POSITION LOGICAL.</td>
</tr>
<tr>
<td>AREASC_LBN</td>
<td>This option indicates that the alignment starts as close as possible to the file specified by the related file identification field (XAB$W_RFI), at the virtual block number of the file specified in the location field (XAB$S_L_LOC). It is also set with the FDL attribute AREA POSITION FILE_ID or the FDL attribute AREA POSITION FILE_NAME.</td>
</tr>
<tr>
<td>AREASC_RFI</td>
<td>Virtual block alignment. This option indicates that the alignment starts at the specified virtual block. It is set with the RMS field XAB$S_L_LOC or the FDL attribute AREA POSITION VIRTUAL. This field is set with the RMS field XAB$S_ALN or the FDL attribute AREA POSITION.</td>
</tr>
<tr>
<td>AREASB_AOP</td>
<td>Alignment options. This field is a binary bit field where each allocation option is defined by a certain bit. Each option is identified by a symbolic bit offset and has a corresponding mask value. The following fields (or masks) are defined for this field.</td>
</tr>
<tr>
<td>AREASV_HARD</td>
<td>This option specifies absolute alignment. If the requested alignment cannot be performed, an error is returned. This option starts at bit 24. It is set with the RMS field XAB$S_AOP (the XAB$S_V_HRD option) or the FDL attribute AREA EXACT_POSITIONING.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AREA$B_AOP</td>
<td>This option requests that RMS locate the allocation on any available cylinder boundary.</td>
</tr>
<tr>
<td>AREA$V_ONC</td>
<td>This option starts at bit 25. It is set with the RMS field XAB$B_AOP (the XAB$V_ONC option) or the FDL attribute AREA POSITION ANY_POSITION.</td>
</tr>
<tr>
<td>AREA$V_CBT</td>
<td>This option indicates that the allocation or a later extension should occupy contiguous blocks, if possible.</td>
</tr>
<tr>
<td>AREA$V_CTG</td>
<td>This option starts at bit 29. It is set with the RMS field XAB$B_AOP (the XAB$V_CBT option) or the FDL attribute AREA BEST_TRY_CONTIGUOUS.</td>
</tr>
<tr>
<td>AREA$L_AVAIL</td>
<td>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.</td>
</tr>
<tr>
<td>AREA$L_CVBN</td>
<td>Available buckets. This field contains the 32-bit virtual block number of the first available bucket in a chain of reclaimed buckets (from the CONVERT/RECLAIM utility). The rest of the buckets on the chain are linked via the BKT$L_NXTBKKT field in each bucket header.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA$L_CNBLK</td>
<td>Number of blocks in current extent. This field contains the number of blocks that were allocated to this extent. The AREA$L_CVBN and AREA$L_CNBLK fields describe the result of an Extend operation for the current extent.</td>
</tr>
<tr>
<td>AREA$L_USED</td>
<td>Number of blocks used. This field contains the number of blocks that have been allocated from the current extent.</td>
</tr>
<tr>
<td>AREA$L_NXTVBN</td>
<td>Next VBN to use. This field contains the virtual block number of the starting block number of the next bucket allocated from the current extent.</td>
</tr>
<tr>
<td>AREA$L_NXT</td>
<td>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.</td>
</tr>
<tr>
<td>AREA$L_NXBLK</td>
<td>Number of blocks in next extent. This field contains the number of blocks that were allocated to the next extent. The AREA$L_NXT and AREA$L_NXBLK fields describe the result of an Extend operation for the next extent.</td>
</tr>
<tr>
<td>AREA$W_DEQ</td>
<td>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$W_DEQ or the FDL attribute AREA EXTENSION.</td>
</tr>
<tr>
<td>AREA$L_LOC</td>
<td>Starting LBN on volume. This field contains the starting logical block number of the last extent of the area.</td>
</tr>
</tbody>
</table>
### Contents of an Area Descriptor (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA$W_RFI</td>
<td>Related file ID. This field contains the file ID of a related file.</td>
</tr>
<tr>
<td>AREA$TOTAL_ALLOC</td>
<td>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 XABS$ALO field during Open, Create, Display, and Extend services.</td>
</tr>
<tr>
<td>AREA$W_CHECK</td>
<td>Checksum. This field allows the standard Files-11 checksum value to be stored in the last word of the area descriptor block.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>BKT$W_ADRSAMPLE</th>
<th>BKT$B_AREANO</th>
<th>BKT$B_CHECKCHAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKT$W_NXTRECID</td>
<td>BKT$W_FREESPACE</td>
<td></td>
</tr>
<tr>
<td>BKT$L_NXTBKT</td>
<td></td>
<td>BKT$B_BKTCB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BKT$B_LEVEL</td>
</tr>
</tbody>
</table>

'BKT$B_INDEXNO FOR PROLOG 3 FILES

BU-2434

5-27
Contents of a Data Bucket

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKTSB_CHECKCHAR</td>
<td>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.</td>
</tr>
<tr>
<td>BKTSB_AREANO</td>
<td>Area number or index number (BKTSB_INDEXNO).</td>
</tr>
<tr>
<td>BKTSB_INDEXNO</td>
<td>For Prolog 1 and 2 files, BKTSB_AREANO contains the area number of the area from which the bucket was allocated. For Prolog 3 files, BKTSB_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.</td>
</tr>
<tr>
<td>BKTSW_ADRSAMPLE</td>
<td>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.</td>
</tr>
<tr>
<td>BKTSW_FREESPACE</td>
<td>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.</td>
</tr>
<tr>
<td>BKTSW_NXTRECID</td>
<td>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.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BKT$W NXTRECID</td>
<td>For Prolog 3 files, the way the record ID is assigned depends on whether the bucket is newly created or a reclaimed bucket. If the bucket is new, the first record is assigned an ID of 1, and subsequent record IDs are assigned in order. If the bucket is reclaimed, the record ID is assigned from the value in the BKT$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.</td>
</tr>
<tr>
<td>BKT$L NXTBKT</td>
<td>VBN of the next bucket. This field contains the starting virtual block number of the next bucket at this level of data buckets; it is a horizontal link to the next bucket. This pointer always points to a bucket of the same size. The last bucket in a level points to the first bucket in the level.</td>
</tr>
<tr>
<td>BKT$B LEVEL</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Contents of a Data Bucket (Cont.)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKT$B_BKTCB</td>
<td>Bucket control bits. This field is a bit-encoded byte field and is used when RMS processes a bucket. The following bits are defined within this field.</td>
</tr>
<tr>
<td>BKT$V_LASTBKT</td>
<td>Last bucket in the horizontal chain for that level. This 1-bit field starts at bit 0.</td>
</tr>
<tr>
<td>BKT$V_ROOTBKT</td>
<td>Root bucket. This 1-bit field starts at bit 1.</td>
</tr>
<tr>
<td>BKT$V_PTR_SZ</td>
<td>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.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>BUCKET HEADER (14 BYTES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKT$W_FREESPACE</td>
</tr>
<tr>
<td>BKT$L_NXTBKT</td>
</tr>
<tr>
<td><strong>USED PORTION OF THE RECORD STORAGE AREA</strong></td>
</tr>
<tr>
<td><strong>UNUSED PORTION OF THE RECORD STORAGE AREA</strong></td>
</tr>
<tr>
<td>CHECK BYTE</td>
</tr>
</tbody>
</table>

Overhead = 15 bytes

| NEXT AVAILABLE BUCKET |
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

<table>
<thead>
<tr>
<th>DATA</th>
<th>KEY</th>
<th>RECORD LENGTH</th>
<th>RFA</th>
<th>RECORD ID</th>
<th>CTRL BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 BYTES</td>
<td>6 BYTES</td>
<td>2 BYTES</td>
<td>1 BYTE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DATA</th>
<th>KEY</th>
<th>RECORD LENGTH</th>
<th>RFA</th>
<th>REC ID</th>
<th>CTRL BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 BYTES</td>
<td>5 BYTES</td>
<td>1 BYTE</td>
<td>1 BYTE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

5-32
The fields of a data record are described in the following table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control byte</td>
<td>Record control byte. The bits are defined as follows:</td>
</tr>
<tr>
<td></td>
<td>0-1 VBN pointer size</td>
</tr>
<tr>
<td></td>
<td>2 Deleted record flag</td>
</tr>
<tr>
<td></td>
<td>3 Record reference vector record flag</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Record identifier. This field occupies two bytes.*</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFA</td>
<td>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.**</td>
</tr>
<tr>
<td>Record length</td>
<td>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.</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Format Details</th>
<th>Record Details</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFA FIELD</td>
<td>VBN ID RECORD ID CONTROL BYTE</td>
<td>9</td>
</tr>
</tbody>
</table>
Simple Bucket Splits

Example 1. Simple Bucket Split

Record B is Added to the File

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

Record E is Moved

5-35
Records C and D are Added to the File

Record E is Moved to a New Bucket
Multibucket Splits

Example 2. Multibucket Split

Record B is added to the file:

```
  B
   C (INDEX POINTER)
```

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:

```
  B
```

```
  100
  2
  A 100.1
  C 100.2
```

```
  100
  200
  300
  A 100.1
  B 200.1
  C 100.2
```

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$N_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  * only works on string keys *

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

<table>
<thead>
<tr>
<th>DATA</th>
<th>KEY</th>
<th>CNT</th>
<th>LEN</th>
<th>RECORD LENGTH</th>
<th>RFA</th>
<th>RECORD ID</th>
<th>CTRL BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BYTE</td>
<td>1 BYTE</td>
<td>2 BYTES</td>
<td>6 BYTES</td>
<td>2 BYTES</td>
<td>1 BYTE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Original Key</th>
<th>Length</th>
<th>Count</th>
<th>Compressed Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barren*</td>
<td>7</td>
<td>0</td>
<td>Barren*</td>
</tr>
<tr>
<td>Barret*</td>
<td>2</td>
<td>5</td>
<td>t*</td>
</tr>
<tr>
<td>Barrett</td>
<td>1</td>
<td>6</td>
<td>t</td>
</tr>
<tr>
<td>Barron*</td>
<td>3</td>
<td>4</td>
<td>on*</td>
</tr>
<tr>
<td>Benson*</td>
<td>6</td>
<td>1</td>
<td>enson*</td>
</tr>
</tbody>
</table>

Total Length = LEN + CNT + X

<table>
<thead>
<tr>
<th># bytes</th>
<th># bytes</th>
<th># bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>physically stored</td>
<td>front</td>
<td>rear compressed</td>
</tr>
</tbody>
</table>

X = Total - (LEN + CNT)

# bytes rear end compressed
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

UNCOMPRESSED DATA:

| MANCHESTER***NH03105 | **********777 MAIN STREET | ****JONES | *****JANE |

COMPRESSED DATA:

| 0 | MANCHESTER***NH03105 | 20 | 8 | 777 MAIN STREET | 16 | 4 | ****JONES | 6 | 5 | *****JANE | 5 |
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$B_INDEXNO field becomes the BKT$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$B_NXTBKT field) points back to the first bucket. The BKT$W_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.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>2-byte pointers. Three bytes for Prolog 1 and 2.</td>
</tr>
<tr>
<td>01</td>
<td>3-byte pointers. Four bytes for Prolog 1 and 2.</td>
</tr>
<tr>
<td>10</td>
<td>4-byte pointers. Five bytes for Prolog 1 and 2.</td>
</tr>
</tbody>
</table>
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

```
KEY   KEY   KEY
```

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

<table>
<thead>
<tr>
<th>KEY VALUE</th>
<th>FRONT COUNT</th>
<th>LENGTH</th>
</tr>
</thead>
</table>

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 (latter only applies if index compression is used)
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**

<table>
<thead>
<tr>
<th>KEY VALUE</th>
<th>VBN</th>
<th>CONTROL BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BYTE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>2-byte pointer size</td>
</tr>
<tr>
<td>01</td>
<td>3-byte pointer size</td>
</tr>
<tr>
<td>10</td>
<td>4-byte pointer size</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Original Key</th>
<th>After Compression</th>
<th>Compressed Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Total length = 11)</td>
<td>Len  Cnt  End</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barren*****</td>
<td>7     0    4</td>
<td>Barren*</td>
<td>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 characters and no leading characters were compressed.</td>
</tr>
<tr>
<td>Barret*****</td>
<td>2     5    4</td>
<td>t*</td>
<td>The 5 leading characters of this key were compressed, which reduced the length of the key to 2 characters.</td>
</tr>
<tr>
<td>Barrette***</td>
<td>3     6    2</td>
<td>te*</td>
<td>The 6 leading characters of this key were compressed, which reduced the length of the key to 3 characters.</td>
</tr>
<tr>
<td>Barron*****</td>
<td>3     4    4</td>
<td>on*</td>
<td>The 4 leading characters of this key were compressed, which reduced the length of the key to 3 characters.</td>
</tr>
<tr>
<td>Benson*****</td>
<td>6     1    4</td>
<td>enson*</td>
<td>Only the initial leading character of this key was compressed, which reduced the length of the key to 6 characters.</td>
</tr>
<tr>
<td>Johns******</td>
<td>6     0    5</td>
<td>Johns*</td>
<td>This key has no characters in common with the preceding 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.</td>
</tr>
<tr>
<td>Johnson****</td>
<td>3     5    3</td>
<td>on*</td>
<td>The 5 leading characters of this key were compressed, which reduced the length of the key to 3 characters.</td>
</tr>
<tr>
<td>Johnston***</td>
<td>4     5    2</td>
<td>ton*</td>
<td>The 5 leading characters of this key were compressed, which reduced the length of the key to 4 characters.</td>
</tr>
<tr>
<td>Johnstone**</td>
<td>2     8    1</td>
<td>e*</td>
<td>The 8 leading characters of this key were compressed, which reduced the length of the key to 2 characters.</td>
</tr>
</tbody>
</table>
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 (BKS\$B_INDEXNO) replaces the area number (BKS\$B_AREANO), and the next record ID (BKS\$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

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

Three bits are defined within the control byte field:

- **DELETED** Bit 2: If set, the record has been deleted.
- **NOPTRSZ** 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

<table>
<thead>
<tr>
<th>FIELD</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>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.</td>
</tr>
<tr>
<td>Key</td>
<td>Key field. This field contains the alternate key value. It's length is specified by the user.</td>
</tr>
<tr>
<td>Key ptr</td>
<td>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:</td>
</tr>
<tr>
<td>Ctrl byte</td>
<td>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).</td>
</tr>
<tr>
<td>Record ID</td>
<td>This word contains the record ID of the primary data record that contains the given secondary key.</td>
</tr>
<tr>
<td>VBN</td>
<td>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.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl byte</td>
<td>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.</td>
</tr>
<tr>
<td>Rec ID</td>
<td>Record ID. This field is 1 byte long. Prolog 3 SIDRs do not have this field.</td>
</tr>
<tr>
<td>Dup count</td>
<td>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.</td>
</tr>
<tr>
<td>Size</td>
<td>Size of the rest of the array. This field is 2 bytes long, and the size does not include itself.</td>
</tr>
<tr>
<td>Key</td>
<td>Key field. This field contains the alternate key value. It’s length is specified by the user.</td>
</tr>
</tbody>
</table>
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)

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

The bucket will contain:

1 byte checksum

14 bytes bucket header

112 bytes data

5-54
Example 1. Entering Data Records into a New Indexed File in Random Order from Program Control

Indexed File After Four Records Added

VAIL Inserted

Split occurs because records not added in sorted order (RMS bases this on order of last two records in bucket)
Example of inserting data records into a new random order from program control.

BUSH Inserted

EVANS Inserted
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:

- **Primary Index Level 1**
  - **JONES**
  - **MAYO**
  - **RAKOS**
  - **SACK**
  - **SMITH

- **Primary Index Level 0**
  - **ASHE**
  - **BUSH**
  - **EVANS**
  - **JONES**
  - **RAKOS**
  - **SACK**
  - **SMITH**
  - **TODD**
  - **VAIL**
  - **WOOGS**

**Random Data Entry**

**Sorted* Data Entry**

- # Index Levels: 2
- # Index Buckets - Primary: 3
- # Data Buckets - Primary: 5
- # RRVs: 5

* 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

DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER11.DAT:1

FILE HEADER

File Spec: DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER11.DAT:1
File ID: (27936,3,0)
Owner UIC: [010,007]
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'00000')

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'00400')

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

5-60
Example 2 (Sheet 2 of 3)

RMS File Statistics

DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER1.DAT]

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: 0
Index Bucket Size: 1, Data Bucket Size: 1
Root VBN: 9
Key Flags:
(0) KEYSV_DUPKEYS 0
(3) KEYSV_IDX_COMPR 0
(4) KEYSV_INITIDX 0
(6) KEYSV_KEY_COMPR 0
(7) KEYSV_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

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 (0) KEYSV_DUPKEYS
0 (1) KEYSV_CKDSKEY
0 (2) KEYSV_NULKEYS
0 (3) KEYSV_IDX_COMPR
0 (4) KEYSV_INITIDX
0 (6) KEYSV_KEY_COMPR

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%
Mean Index Bucket Compression: 0%

The analysis uncovered NO errors.
Alternate Index Tree

LEVEL 0 (SIDRs)

LEVEL 1

RFA's*  SEQ-NQ  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 #
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)
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.

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

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 SYSSOUTPUT 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.
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 !
11 ! This program illustrates the use of the RTL performance measurement routines.
12 !
13 EXTERNAL LONG FUNCTION LIB$INIT_TIMER, LIB$SHOW_TIMER
14 DECLARE LONG result, j, k
15 DIMENSION LONG array (99%, 99%)
16 !
17 result = LIB$INIT_TIMER()
18 CALL LIB$STOP (result BY VALUE) IF (result AND 1%) = 0%
19 !
20 FOR k = 0% TO 99% STEP 1%
21 FOR j = 0% TO 99% STEP 1%
22 array (j,k) = k
23 NEXT j
24 NEXT k
25 result = LIB$SHOW_TIMER()
26 PRINT 'Usage values after array initialization'
27 CALL LIB$STOP (result BY VALUE) IF (result AND 1%) = 0%
28 END

$ BASIC LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER
Elapsed: 00:00:00.36 CPU: 00:00:00.21 BUFIO: 0 DIRIO: 0 FAULTS: 1
Usage values after array initialization
```
COBOL Example 1 (Sheet 2 of 5)

- 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.

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID. LIBTIMER.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 ARRAY.
  02 DIM0 OCCURS 100 TIMES.
  03 DIM2 OCCURS 100 TIMES.
  05 IARRAY PIC 99(9) COMP.
  01 I PIC 99(9) COMP.
  01 J PIC 99(9) COMP.
  01 RESULT PIC 99(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 J > 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.
```
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)

```fortran
C** LIBTIMER.FOR

C This program illustrates the use of the RTL performance measurement routines.
C
IMPLICIT INTEGER*4 (A-Z)
DIMENSION IARRAY(100, 100)

RESULT = LIB$INIT TIMER()
IF (.NOT. RESULT) CALL LIB$STOP($VAL(RESULT))

C Initialize the array
DO J=1, 100
   DO K=1, 100
      IARRAY(J,K) = 1
   ENDDO
ENDDO

RESULT= LIB$SHOW TIMER()
IF (.NOT. RESULT) CALL LIB$STOP($VAL(RESULT))

C 'Usage values after array initialization'

END
```

$ FORTRAN LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER

ELAPSED: 00:00:00.21 CPU: 00:00:00.19 BUFIO: 0 DIRIO: 0 FAULTS: 80

Usage values after array initialization
MACRO

- 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

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42

$ MACRO LIBTIMER
$ LINK LIBTIMER
$ RUN LIBTIMER

ELAPSED: 00:00:00.56 CPU: 00:00:00.21 BUFIO: 0 DIRIO: 0 FAULTS: 22
Usage values after computation:

6-6
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)

```pascal
PROGRAM libtimer (INPUT, OUTPUT);

(" LIBTIMER.PAS
(* This program illustrates the use of the RTL
(* performance measurement routines.

VAR iarray: ARRAY [1..100, 1..100] OF INTEGER;
  count1, count2: INTEGER;
  lib_stat: INTEGER;

FUNCTION LIB$INIT_TIMER( VAR handle adr: INTEGER
  := %IMMED 0); INTEGER; EXTERN;

FUNCTION LIB$SHOW_TIMER(
  handler adr: INTEGER := %IMMED 0;
  code: INTEGER := %IMMED 0; %IMMED [UNBOUND]
  function action_rtn :INTEGER := %IMMED 0;
  %IMMED user_arg: INTEGER := %IMMED 0);
  INTEGER; EXTERN;

PROCEDURE LIB$STOP( %IMMED cond_value: INTEGER); EXTERN;

BEGIN
  lib_stat:= LIB$INIT_TIMER;
  IF NOT ODD( lib_stat) THEN LIB$STOP( lib_stat);

  (* Initialize the array *)
  FOR count1:= 1 TO 100 DO
    FOR count2:= 1 TO 100 DO
      iarray[count1,count2]:= 1;

  lib_stat:= LIB$SHOW_TIMER;
  WRITELN('Usage values after array initialization');
  IF NOT ODD( lib_stat) THEN LIB$STOP( lib_stat)
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

6-7
# PART 4. EVALUATING/UTILIZING

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

(Sheet 1 of 2)

### RMS_Puts

<table>
<thead>
<tr>
<th>Initial</th>
<th>Additional</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FILE</th>
<th>NAME</th>
<th>ORGANIZATION</th>
<th>RECORD</th>
<th>CARRIAGE_CONTROL</th>
<th>FORMAT</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX/VMS</td>
<td>&quot;INDXBACK.DAT&quot;</td>
<td>indexed</td>
<td>carriage_return</td>
<td>fixed</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AREA 0</th>
<th>ALLOCATION</th>
<th>180</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEST_TRY_CONTIGUOUS</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>BUCKET_SIZE</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>EXTENSION</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AREA 1</th>
<th>ALLOCATION</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEST_TRY_CONTIGUOUS</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>BUCKET_SIZE</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>EXTENSION</td>
<td>3</td>
</tr>
</tbody>
</table>

### FAST_CONVERT

<table>
<thead>
<tr>
<th>Initial</th>
<th>Additional</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FILE</th>
<th>NAME</th>
<th>ORGANIZATION</th>
<th>RECORD</th>
<th>CARRIAGE_CONTROL</th>
<th>FORMAT</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX/VMS</td>
<td>&quot;INDXBACK.DAT&quot;</td>
<td>indexed</td>
<td>carriage_return</td>
<td>fixed</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AREA 0</th>
<th>ALLOCATION</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEST_TRY_CONTIGUOUS</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>BUCKET_SIZE</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>EXTENSION</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AREA 1</th>
<th>ALLOCATION</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEST_TRY_CONTIGUOUS</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>BUCKET_SIZE</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>EXTENSION</td>
<td>3</td>
</tr>
</tbody>
</table>

### KEY 0

<table>
<thead>
<tr>
<th>CHANGES</th>
<th>DATA_AREA</th>
<th>DATA_FILL</th>
<th>DATA_KEY_COMPRESSION</th>
<th>DATA_RECORD_COMPRESSION</th>
<th>DUPLICATES</th>
<th>INDEX_AREA</th>
<th>INDEX_COMPRESSION</th>
<th>INDEX_FILL</th>
<th>LEVEL1_INDEX_AREA</th>
<th>NAME</th>
<th>SEGNO_LENGTH</th>
<th>SEGNO_POSITION</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>0</td>
<td>100</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>1</td>
<td>no</td>
<td>100</td>
<td>1</td>
<td>&quot;SEQ_NO&quot;</td>
<td>5</td>
<td>0</td>
<td>string</td>
</tr>
</tbody>
</table>

6-8
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$INSTRUCTOR:[WOODS.RMS.COURSE]BACK1.FDL;1
18
19 ALLOCATION 180
19 BEST_TRY_CONTIGUOUS yes
20 BUCKET_SIZE 3
21 EXTENSION 45
22
**********
File DISK$INSTRUCTOR:[WOODS.RMS.COURSE]BACK2.FDL;1
18
19 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: 00:00:15.90 CPU Time: 00:00:04.82

$DIR/FULL INDXBACK.DAT

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

INDXBACK.DAT:1 File ID: (31322,14,0)
Size: 128/128 Owner: [VMS,WOODS]
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
Record format: Fixed length 50 byte records
Record attributes: Carriage return carriage control
File protection: F, item:R, owner:RWED, group:R, world:
Access Cntrl List: One

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: 00:00:03.33  CPU Time: 00:00:02.20

$DIR/FULL INDXFRONT.DAT

Directory DISKINSTRUCTOR:[WOODS.RMS.COURSE]
INDXFRONT.DAT;1  File ID: (31155,15,0)
Size: 128/128  Owner: [VMS,WOODS]
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
Access Ctrl List: None

Total of 1 file, 128/128 blocks.
Example 5. Loading BACKWARDS Data Using RMS_Puts in Program Control

$SSH STATUS

Status on 1-JAN-1986 20:46:30.46 Elapsed CPU : 00:16:36.68

$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: 00:00:13.89 BUFIO: 15 DIRIO: 1868 FAULTS: 110

VS FAST_CONVERT 0:04.82 29 221 264

$SSH STATUS

Status on 1-JAN-1986 20:47:34.54 Elapsed CPU : 00:16:51.09
Buff. I/O : 41520 Cur. ws. : 300 Open files : 2

$DIR/FULL INDXBACK.DAT

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

INDXBACK.DAT:2 File ID: (31581,39,0)
Size: 368/368 Owner: [VMS,WOODS]
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
Access Ctrl List: None

6-12
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:[WORES.RMS.COURSE]INDXBACK.DAT;2

FILE HEADER

File Spec: DISK$INSTRUCTOR:[WORES.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

6-13
Example 6 (Sheet 2 of 2)

RMS File Statistics  1-JAN-1986  20:52:05.41
DISK:INSTRUCTOR:[WOODS,RMS,COURSE]INDXBACK.DAT: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)  KEYSV_DUPKEYS  0
(3)  KEYSV_IDX_COMPR  0
(4)  KEYSV_INITIDX  0
(6)  KEYSV_KEY_COMPR  0
(7)  KEYSV_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
Example 7. Analyze Statistics for FRONTWARDS RMS_Puts Data Load

(Sheet 1 of 2)

$ANALYZE/RMS_FILE/STATISTICS/OUT=INDEXFRONT.ANL INDEXFRONT.DAT

RMS File Statistics
1-JAN-1986 20:50:03.56
DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDEXFRONT.DAT;2

FILE HEADER

File Spec: DISK$INSTRUCTOR:[WOODS.RMS.COURSE]INDEXFRONT.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]INDEXFRONT.DAT: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=INDEXFRONT.ANL INDEXFRONT.DAT

6-16
Example 8. Comparison of Statistics for RMS_Puts Data Load -- BACKWARDS versus FRONTWARDS RMS_Puts Data Load

ANALYZE STATISTICS FOR KEY #0

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

ANALYZE FILE SUMMARY

<table>
<thead>
<tr>
<th></th>
<th>INDXBACK</th>
<th>INDXFRONT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket size</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Number blocks allocated</td>
<td>368</td>
<td>140</td>
</tr>
<tr>
<td>Default extent</td>
<td>45</td>
<td>33</td>
</tr>
<tr>
<td>Area 0 - Number unused</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>Area 1 - Number unused</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

RTL LIB$SHOW_TIMER STATISTICS for LOADINDX:

<table>
<thead>
<tr>
<th></th>
<th>INDXBACK</th>
<th>INDXFRONT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time</td>
<td>0:01:49</td>
<td>0:46:54</td>
</tr>
<tr>
<td>CPU time</td>
<td>0:15:89</td>
<td>0:11:94</td>
</tr>
<tr>
<td>Buffered I-O</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>Direct I-O</td>
<td>1868</td>
<td>978</td>
</tr>
<tr>
<td>Page faults</td>
<td>110</td>
<td>112</td>
</tr>
</tbody>
</table>

Deferred write

6-17
Calculation of percent difference in DIO between INDXBACK_DFW and INDXFRONT_DFW

\[
\text{Percent Difference} = \frac{(\text{Poorer way} - \text{Better way})}{\text{Poorer way}} \times 100.0
\]

\[
\frac{978 - 395}{978} \times 100.0 = 78.85\% \text{ reduction in DIO}
\]
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]
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 specified
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 0x'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) $KEYV_DUPKEYS: 0
(3) $KEYV_IDX_COMPR: 1
(4) $KEYV_INITIDX: 0
(6) $KEYV_KEY_COMPR: 1
(7) $KEYV_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

6-20
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**

<table>
<thead>
<tr>
<th>Initial Accessor Access</th>
<th>Subsequent Accessor Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS GET*</td>
<td>SHARING GET*</td>
</tr>
<tr>
<td>ACCESS DELETE</td>
<td>SHARING DELETE</td>
</tr>
<tr>
<td>ACCESS PUT</td>
<td>SHARING PUT</td>
</tr>
<tr>
<td>ACCESS UPDATE</td>
<td>SHARING UPDATE</td>
</tr>
<tr>
<td>ACCESS TRUNCATE**</td>
<td>No access allowed</td>
</tr>
</tbody>
</table>

* 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

BASIC
ACCESS
READ
WRITE
MODIFY
SCRATCH
APPEND (sequential)

ALLOW
READ
WRITE (locks against delete or scratch)
MODIFY (unlimited access)
NONE
DEFAULT - MODIFY
If access READ - READ and other access - NONE

DEFAULT
No default - must specify one of above keywords.

COBOL
INPUT
OUTPUT
EXTEND
I-O

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

PASCAL
History
OLD
NEW
UNKNOWN
READONLY

Sharing
READONLY
READWRITE
NONE

DEFAULT - NEW
(except if external file opened using RESET or EXTEND - OLD)

DEFAULT
If History = READONLY - READONLY
If History ≠ READONLY - NONE
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 SHARE 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 file level.

- In the case of shared sequential files, bucket 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 RABSV_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 another 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  
   RABSM_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
FORTRAN - N/A
PASCAL - N/A
MACRO - RABSL_ROP = RABSV_ULK
ALTERNATIVE RECORD LOCKING CONTROLLING OPTIONS

There are several record locking options available that can be set in the RAB$SL_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$V_TMO field contains the maximum time value, in seconds, to be
allowed for a record input wait caused by a locked record. If the
timeout period expires and the record is still locked, RMS will
abort the record operation with the RMS$_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)
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= NOPOBUF. 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 $S$FLUSH).

Write-Behind

When a process switches from one buffer to another (for example, a ninth $S$PUT in a series of sixteen $S$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 $S$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 $S$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 $S$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$MASY bit in the RAB$LRSP

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 SPUTs 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

![Primary Index Tree Diagram]

BU-2468
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**

```
1  2  3
```

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**

```
1  2  3
VBN 9 | VBN 7 | VBN 6
2    | 1    | 0
```

8-7
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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBN 9</td>
<td>VBN 7</td>
<td>VBN 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Update 3. $PUT Record with LAST_NAME = PIG

This $PUT involves following buckets 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**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VBN 9</td>
<td>VBN 5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>VBN 9</th>
<th>VBN 5</th>
<th>VBN 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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.

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).

Multi-blocks
Sequential
Relative
Indexed
Bucket
Bucket
maximum size = 63 blocks

# 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
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:

\[ \text{I/O TIME} = \text{SEEK} + \text{LATENCY} + \text{DATA TRANSFER} + \text{BUCKET SEARCH} \]

- 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

<table>
<thead>
<tr>
<th>Bucket Size</th>
<th>Seek Time (ms)</th>
<th>Rotational Latency (ms)</th>
<th>Data Transfer Time (ms)</th>
<th>Total (ms)</th>
<th>% Data Transfer of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28</td>
<td>8.3</td>
<td>.2</td>
<td>36.5</td>
<td>0.6%</td>
</tr>
<tr>
<td>10</td>
<td>28</td>
<td>8.3</td>
<td>2.3</td>
<td>38.6</td>
<td>6.0%**</td>
</tr>
<tr>
<td>15</td>
<td>28</td>
<td>8.3</td>
<td>3.5</td>
<td>39.8</td>
<td>8.8%</td>
</tr>
<tr>
<td>20</td>
<td>28</td>
<td>8.3</td>
<td>4.7</td>
<td>41.0</td>
<td>11.5%</td>
</tr>
<tr>
<td>32</td>
<td>28</td>
<td>8.3</td>
<td>7.4</td>
<td>43.7</td>
<td>16.9%</td>
</tr>
<tr>
<td>63</td>
<td>28</td>
<td>8.3</td>
<td>14.7</td>
<td>51.0</td>
<td>28.8%</td>
</tr>
</tbody>
</table>

* 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.

8-14
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. Buffers are locked into 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

* Unless local buffers, they 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 and subsequent processes may either:
(a) Ignore global buffers
(b) Accept what the first 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  
FABS_B_SHR = FABS_M_SHRGET  
+  
SHARING MULTISTREAMING  
FABS_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:

<table>
<thead>
<tr>
<th>ACCESS</th>
<th>WRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHARING</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

# 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)

8-19
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 FABSW_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)

Example of Global Buffer Table Mode Application

Process A opens the file for shared access write with:

<table>
<thead>
<tr>
<th>WRITE</th>
<th>ACCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE</td>
<td>READ</td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>READ</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

Set global buffers:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessed</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Available</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Delete</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Caution on global buffer:

* Do not access the global buffer without permission.

* Access the global buffer during program execution only.

* Do not modify or delete the global buffer.

* Do not access the global buffer that is not in use.

* Do not access the global buffer that is no longer required.

Extra key of reference = 0 (primary key)
Alternate Index Tree

VBN #14

LEVEL 1

HIGH VALUES

VBN #13

LEVEL 0 (SDR's)

1 2 3 4 5 6 7 8 9 10 11

1 2 3 4 5 6 7 8 9 10 11

LEVEL 0

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

8-21
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) |
|-------------------------------|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

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)
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)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBN 9</td>
<td>VBN 7</td>
<td>VBN 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Retrieval 2

Process A issues FIND ** WOODS ** followed by a GET.

Internal Table at the End of Retrieval 2 (WOODS)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBN 9</td>
<td>VBN 7</td>
<td>VBN 6</td>
<td>VBN 8</td>
<td>VBN 16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Retrieval 3

Process B issues FIND ** SEQ_NO = 9 (MAYO) ** followed by GET.

Internal Table After the Root Alternate Key - Bucket Brought In

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBN 9</td>
<td>VBN 7</td>
<td>VBN 6</td>
<td>VBN 8</td>
<td>VBN 14</td>
<td>VBN 16</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>VBN 9</th>
<th>VBN 7</th>
<th>VBN 13</th>
<th>VBN 8</th>
<th>VBN 16</th>
<th>VBN 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

SEARCH POINTER

BU-2479
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 owned. 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)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>VBN 9</td>
<td>VBN 7</td>
</tr>
<tr>
<td>VBN 13</td>
<td>VBN 8</td>
</tr>
</tbody>
</table>

BU-2480

Process A's Local Buffers at the End of Retrieval 4 (No Bucket Split)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>VBN 4</td>
<td></td>
</tr>
</tbody>
</table>

BU-2481

8-26
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)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>VBN 9</td>
<td>VBN 7</td>
<td>VBN 13</td>
<td>VBN 14</td>
<td>VBN 4</td>
<td>VBN 5</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

WEIGHTS RESTORED

BU-2482

8-27
Summary – Global Buffer Performance (as of VAX/VMS Version 4.4)

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.
VAXCLUSTERS

- 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.
CALCULATING THE NUMBER OF BUFFERS NEEDED TO CACHE INDEX

Example 4. Real-Data ANALYZE/RMS/STAT Output
(Sheet 1 of 2)

RMS File Statistics
Page 1

FILE HEADER

File Spec: DISKXY2:[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

8-32
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) KEYSV_DUPKEYS 0
- (3) KEYSV_IDX_CMPTR 1
- (4) KEYSV_INITIDX 0
- (6) KEYSV_KEY_CMPTR 1
- (7) KEYSV_REC_CMPTR 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

\[ 1 + \left( \frac{48}{3} \right) = 14 \]

buffer to cache entire index for primary key

allows 1 data bucket

8-33
Example 5. Run-Time Statistics for REALDATA1.DAT  
Varying the Number of Buffers

<table>
<thead>
<tr>
<th># BUFFERS</th>
<th>ELAPSED</th>
<th>CPU</th>
<th>BIO</th>
<th>DIO</th>
<th>PG FAULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7:13.23</td>
<td>4:20.67</td>
<td>3</td>
<td>2742</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4:49.64</td>
<td>4:11.32</td>
<td>3</td>
<td>1637</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>4:40.88</td>
<td>4:08.60</td>
<td>3</td>
<td>1397</td>
<td>26</td>
</tr>
<tr>
<td>9</td>
<td>4:31.11</td>
<td>4:03.99</td>
<td>3</td>
<td>582</td>
<td>25</td>
</tr>
<tr>
<td>20</td>
<td>4:18.30</td>
<td>4:02.27</td>
<td>3</td>
<td>138</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>4:06.69</td>
<td>4:00.21</td>
<td>3</td>
<td>137</td>
<td>49</td>
</tr>
<tr>
<td>40</td>
<td>4:05.36</td>
<td>4:00.76</td>
<td>3</td>
<td>137</td>
<td>95</td>
</tr>
</tbody>
</table>

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
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

```
Indexed Data File
    | ANALYZE/RMS_FILE
    | FDL File
    | (with ANALYSIS Sections)
    | Original FDL File
    | EDIT/FDL
    | FDL File (Revised)
    | CONVERT
    | Tuned Indexed Data File
```
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/DDL=ddl-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

- CONVERSION / CLARION

- ACF/3500

- ACF/2500

- ACF/1000

- ACF/500

- ACF/250

- ACF/250

- ACF/25

- ACF/10

- ACF/5

- ACF/2

- ACF/1

- ACF/0.5

- ACF/0.2

- ACF/0.1

- ACF/0.05

- ACF/0.02

- ACF/0.01

- ACF/0.005

- ACF/0.002

- ACF/0.001

- ACF/0.0005

- ACF/0.0002

- ACF/0.0001

- ACF/0.00005

- ACF/0.00002

- ACF/0.00001

- ACF/0.000005

- ACF/0.000002

- ACF/0.000001

- ACF/0.0000005

- ACF/0.0000002

- ACF/0.0000001

- ACF/0.00000005

- ACF/0.00000002

- ACF/0.00000001

- ACF/0.000000005

- ACF/0.000000002

- ACF/0.000000001

- ACF/0.0000000005

- ACF/0.0000000002

- ACF/0.0000000001

- ACF/0.00000000005

- ACF/0.00000000002

- ACF/0.00000000001

- ACF/0.000000000005

- ACF/0.000000000002

- ACF/0.000000000001

- ACF/0.0000000000005

- ACF/0.0000000000002

- ACF/0.0000000000001

- ACF/0.00000000000005

- ACF/0.00000000000002

- ACF/0.00000000000001

- ACF/0.000000000000005

- ACF/0.000000000000002

- ACF/0.000000000000001

- ACF/0.0000000000000005

- ACF/0.0000000000000002

- ACF/0.0000000000000001
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 SUMMARIES

Major Topics
- Designing file specifications
- Tuning file performance

Source:
Guide to Microsoft File Applications — Chapter 4 (Section 4.1–4.2)
# DESIGN -- FILE CREATION PARAMETERS

<table>
<thead>
<tr>
<th>SEQ</th>
<th>REL</th>
<th>INDX</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1. Contiguous disk allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-- Initial file allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-- Extend size</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>2. Block spanning (sequential files)</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>3. Bucketsize</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>4. Primary Key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-- Unique value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-- Position 0</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>5. Number of alternate keys</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>6. Multiple areas</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>7. Bucket fill factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8. Compression index (more important)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-- data</td>
</tr>
<tr>
<td>SEQ</td>
<td>REL</td>
<td>INDX</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

1. Buffer size (sequential only)

2. Number of buffers

3. READ-AHEAD or WRITE-BEHIND (sequential only)

4. DEFERRED WRITE

5. Window size

6. Global buffers

7. Record locking options

8. Fast delete

SDA> show process /ims

static display - show local & global buffers
### Existing Keywords Available in Higher-Level Languages for Implementing Optimizing Features

<table>
<thead>
<tr>
<th>Optimizing Features</th>
<th>BASIC</th>
<th>COBOL</th>
<th>FORTRAN</th>
<th>PASCAL</th>
<th>DCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best-try-contiguous</td>
<td>Contiguous</td>
<td>I-O Control</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contiguous-best-try</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contiguous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contiguous</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial alloc.</td>
<td>FILESIZE n</td>
<td>I-O PREALLOCATION n</td>
<td>INITIALSIZE=n</td>
<td>--</td>
<td>SET FILE</td>
</tr>
<tr>
<td>Extend</td>
<td>EXTENDSIZE n</td>
<td>I-O EXTENSION n</td>
<td>EXTENDSIZE=n</td>
<td>--</td>
<td>SET RMS_DEFAULT</td>
</tr>
<tr>
<td>Buffer size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiblocks (sequential)</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bucket size (rel., indx)</td>
<td>BUCKETSIZES n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blockspan (sequential)</td>
<td>NOSPAN, SPAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NOSPANBLOCKS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(default span)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple areas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill factor</td>
<td>--</td>
<td>FILL-SIZE*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global buffers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SET FILE</td>
</tr>
<tr>
<td>Number buffers</td>
<td>BUFFER n</td>
<td>File Control</td>
<td>BUFFERCOUNT=n</td>
<td>--</td>
<td>SET RMS_DEFAULT</td>
</tr>
<tr>
<td>Window size</td>
<td>WINDOWSIZE n</td>
<td>I-O WINDOW n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-ahead (sequential)</td>
<td>--</td>
<td></td>
<td>(enabled RAB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-behind (sequential)</td>
<td>--</td>
<td></td>
<td>(enabled RAB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferred write (relative, index)</td>
<td>(enabled FAB)**</td>
<td>I-O DEFERRED-WRITE</td>
<td>(enabled FAB)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Causes it to use fill size specified at file creation else uses 100% fill.

** Enabled if ALLOW not equal to WRITE or MODIFY.
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 II
PERMANENT FILE ATTRIBUTES VERSUS RUN-TIME FILE CHARACTERISTICS

Major Topics
- Permanent file attributes
- Run-time file characteristics
- Datafile selection for RMS control blocks for different-level languages
- DCL commands for implementing run-time features
- SET FILE
- SET RMS_DEFAULT
- Specifying run-time options

Source
- UNIX File Attributes -- Chapter 3
- RMS Reference -- Chapter 3, (FAD), Chapter 1 (RAD)
- UNIX and the Definition Language "Unix 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
- Number of local buffers
- Retrieval window size
- File sharing options
- File processing options which can be set in FAB$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$ROP (for example, fast delete)
Run-time file open and record connection options that apply to file performance are summarized below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous record processing*</td>
<td>Specifies that record I/O for this record stream will be done asynchronously.</td>
</tr>
<tr>
<td>FDL:</td>
<td>CONNECT ASYNCHRONOUS</td>
</tr>
<tr>
<td>RMS:</td>
<td>RAB$L_ROP RAB$V_ASY</td>
</tr>
<tr>
<td>Deferred write*</td>
<td>Allows records to be accumulated in a buffer and written only when the buffer is needed or when the file is closed.</td>
</tr>
<tr>
<td>FDL:</td>
<td>FILE DEFERRED WRITE</td>
</tr>
<tr>
<td>RMS:</td>
<td>FAB$L_POP FAB$V_DFW</td>
</tr>
<tr>
<td>Default extension quantity</td>
<td>Specifies the number of blocks to be allocated to a file when more space is needed.</td>
</tr>
<tr>
<td>FDL:</td>
<td>FILE EXTENSION</td>
</tr>
<tr>
<td>RMS:</td>
<td>RAB$W_DEQ</td>
</tr>
<tr>
<td>Fast delete*</td>
<td>Postpones certain internal operations associated with deleting indexed file records until the record is accessed again. This allows records to be deleted rapidly, but may affect the performance of subsequent accessors reading the file until its next-scheduled convert.</td>
</tr>
<tr>
<td>FDL:</td>
<td>CONNECT FAST_DELETE</td>
</tr>
<tr>
<td>RMS:</td>
<td>RAB$L_ROP RAB$V_FDL</td>
</tr>
<tr>
<td>Global buffer count</td>
<td>Specifies whether global buffers will be used and the number to be used if the record stream is the first to connect to the file.</td>
</tr>
<tr>
<td>FDL:</td>
<td>CONNECT GLOBAL_BUFFER_COUNT</td>
</tr>
<tr>
<td>RMS:</td>
<td>FAB$W_GBC</td>
</tr>
<tr>
<td>Locate mode*</td>
<td>Allows the use of locate mode, not move mode, when reading records.</td>
</tr>
<tr>
<td>FDL:</td>
<td>CONNECT LOCATE_MODE</td>
</tr>
<tr>
<td>RMS:</td>
<td>RAB$L_ROP RAB$V_LOC</td>
</tr>
</tbody>
</table>
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$B_ROP RAB$V_RA

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$B_FOP FAB$V_SOO

Write-behind*

Alternates buffer use between two buffers; as of VMS 4.4 for nonshared sequential files only.

FDL: CONNECT WRITE_BEHIND
RMS: RAB$B_ROP RAB$V_WBH

* Indicates on option that can be specified for each record-processing operation.
<table>
<thead>
<tr>
<th>Optimizing Features</th>
<th>BASIC</th>
<th>COBOL</th>
<th>FORTRAN</th>
<th>PASCAL</th>
<th>DCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best-try-contiguous</td>
<td>Contiguous</td>
<td>I-O Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contiguous-best-try</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contiguous</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contiguous</td>
<td></td>
<td>FILESIZE n</td>
<td></td>
<td>INITIALSIZE=n</td>
<td></td>
</tr>
<tr>
<td>Initial alloc.</td>
<td></td>
<td>EXTENDSIZE n</td>
<td></td>
<td>EXTENDSIZE=n</td>
<td></td>
</tr>
<tr>
<td>Extend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer size</td>
<td></td>
<td>BLOCKSIZE=n</td>
<td></td>
<td></td>
<td>SET FILE</td>
</tr>
<tr>
<td>Multiblocks (sequential)</td>
<td></td>
<td>NOSPAN, SPAN</td>
<td></td>
<td></td>
<td>SET RMS_DEFAULT</td>
</tr>
<tr>
<td>Bucket size (rel., index)</td>
<td>BUCKETSIZE n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blockspan (sequential)</td>
<td>NOSPAN, SPAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOSPANBLOCKS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(default span)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple areas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill factor</td>
<td></td>
<td>FILL-SIZE*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global buffers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SET FILE</td>
</tr>
<tr>
<td>Number buffers</td>
<td></td>
<td>BUFFER n</td>
<td></td>
<td>BUFFERCOUNT=n</td>
<td>SET RMS_DEFAULT</td>
</tr>
<tr>
<td>Window size</td>
<td></td>
<td>RESERVE n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-ahead (sequential)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-behind (sequential)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferred write (relative, index)</td>
<td>(enabled FAB)**</td>
<td>I-O DEFERRED-WRITE</td>
<td></td>
<td>(enabled FAB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(enabled FAB)</td>
<td></td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Name</th>
<th>BASIC OPEN</th>
<th>FORTRAN OPEN</th>
<th>PASCAL OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Keyword and Value</td>
<td></td>
<td>Parameters and Value</td>
</tr>
<tr>
<td>FARS_LAL0</td>
<td>Allocation quantity</td>
<td>As specified by FILESIZE</td>
<td>n if INITIALIZE = n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bucket size (# blocks)</td>
<td>(see Footnote 1.)</td>
<td>MIN((x + 511)/512, 12)</td>
<td></td>
</tr>
<tr>
<td>FARS_RKSS</td>
<td></td>
<td>Magtape BLOCKSIZEx</td>
<td>where x = MAX(RECL + 24, BLOCKSIZE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Initialized to zero</td>
<td>n if BLOCKSIZE = n</td>
<td></td>
</tr>
<tr>
<td>FARS_BLS</td>
<td>Block size</td>
<td>As specified by EXTENDSIZE</td>
<td>Initialized to zero</td>
<td></td>
</tr>
<tr>
<td>FARS_CTX</td>
<td>Context</td>
<td>Returned by RMS</td>
<td>n if EXTENDSIZE = n</td>
<td></td>
</tr>
<tr>
<td>FARS_DX3</td>
<td>Default file</td>
<td>DEFAULTNAME file specification</td>
<td>Returned by RMS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extension quantity</td>
<td>address</td>
<td>UNIT = nnn.DAT</td>
<td></td>
</tr>
<tr>
<td>FARS_DGE</td>
<td>Device characteristics</td>
<td></td>
<td>set to FORnnn.DAT</td>
<td></td>
</tr>
<tr>
<td>FARS_DGA</td>
<td>Default file specification</td>
<td></td>
<td>FORREAD.DAT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>string address</td>
<td></td>
<td>FORACCEPT.DAT, FORTYPE.DAT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FORPRINT.DAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or DEFAULTFILE</td>
<td></td>
</tr>
<tr>
<td>FARS_DNS</td>
<td>Default file specification</td>
<td></td>
<td>Set to length of default</td>
<td></td>
</tr>
<tr>
<td></td>
<td>string size</td>
<td></td>
<td>file name string</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_BAC</td>
<td>File access options</td>
<td>1 if not ACCESS MODIFY, Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_DBL</td>
<td>Allow deletions</td>
<td>1 if not ACCESS READ, MODIFY, SCRATCH, Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_DBE</td>
<td>Allow reads</td>
<td>1 if ACCESS WRITE, MODIFY, SCRATCH, APPEND, Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_PUT</td>
<td>Allow writes</td>
<td>1 if ACCESS SCRATCH, Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_TRN</td>
<td>Allow truncates</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_UPD</td>
<td>Allow updates</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_FNA</td>
<td>File specification</td>
<td>File name if FILE present if NAME if specified, else name of file variable if external file, else zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>string address</td>
<td>else to FORnnn, FOR$READ, FORACCEPT, FORTYPE, FORPRINT, SYS$INPUT, or SYS$OUTPUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_FNS</td>
<td>File specification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>string size</td>
<td></td>
<td>Set to length of file name</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_FOP</td>
<td>File processing options</td>
<td>1 if CONTIGUOUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_CBT</td>
<td>Contiguous best try</td>
<td>1 if not FOR INPUT/OUTPUT, or MODIFY else 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_CIF</td>
<td>Create if nonexistent</td>
<td>initialized to zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_CCG</td>
<td>Contiguous allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FARS_DFW</td>
<td>Deferred write</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Settings of VAX RMS FAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Default Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABSV_DLT</td>
<td>Delete on close service</td>
<td>Initialized to zero</td>
<td>Set at FORTRAN CLOSE, depending on keywords DISP (OPEN or CLOSE) or STATUS (CLOSE)</td>
</tr>
<tr>
<td>FABSV_ESCAPE</td>
<td>Escape, nonstandard processing</td>
<td>Initialized to zero</td>
<td>Set when file is closed, depending on DISPOSITION</td>
</tr>
<tr>
<td>FABSV_INPUT</td>
<td>Input, make this SYS$INPUT</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_KFO</td>
<td>Known file open</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_MNV</td>
<td>Maximize version number</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_NAME</td>
<td>Name block inputs</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_NEF</td>
<td>Not positioned at end of file</td>
<td>0 if ACCESS APPEND else 1</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_NFS</td>
<td>Not file structured</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_OFP</td>
<td>Output file parse</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_POS</td>
<td>Current position (after closed file)</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_PPF</td>
<td>Process permanent file</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_RCK</td>
<td>Read check</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_RWC</td>
<td>Rewind on close service</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_RWO</td>
<td>Rewind on open service</td>
<td>0 if NOREWIND else 1</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_SCF</td>
<td>Submit command file (when closed)</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_SPL</td>
<td>Spool to printer</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_SQO</td>
<td>Sequential only</td>
<td>1 if terminal format file</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_SUP</td>
<td>Supersede</td>
<td>1 if FOR OUTPUT</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
<tr>
<td>FABSV_TEF</td>
<td>Truncate at end-of-file</td>
<td>Initialized to zero</td>
<td>l if terminal file reopened to enable prompting</td>
</tr>
</tbody>
</table>
### Settings of VAX RMS FAB Fields by Higher-Level Language

**Compilers by Default at FILE OPEN (Cont.)**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABS_TMD</td>
<td>Temporary (marked for deletion)</td>
<td>1 if TEMPORARY</td>
</tr>
<tr>
<td>FABS_TNP</td>
<td>Temporary (file with no directory entry)</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FABS_UMF</td>
<td>User file mode</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FABS_UFO</td>
<td>User file open or create file only</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FABS_WCK</td>
<td>Write check</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FABS_FSZ</td>
<td>Fixed control area size</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FABS_IFI</td>
<td>Internal file identifier</td>
<td>Returned by RMS</td>
</tr>
<tr>
<td>FABS_HRN</td>
<td>Maximum record number</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FABS_HRS</td>
<td>Maximum record size</td>
<td>132 if terminal-format file or ORGANIZATION = SEQUENTIAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set to address of name block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(the expanded and resultant string areas are set up, but the related file name string is not)</td>
</tr>
<tr>
<td>FABS_Nam</td>
<td>Name block address</td>
<td>FABSC REL if relative</td>
</tr>
<tr>
<td>FABS_ORG</td>
<td>File organization</td>
<td>FABSC IDX if indexed</td>
</tr>
<tr>
<td>FABS_RAT</td>
<td>Record attributes</td>
<td>FABSC SEQ if SEQUENTIAL, VIRTUAL, UNDEFINED, or omitted</td>
</tr>
<tr>
<td>FABS_FTM</td>
<td>FORTRAN carriage control</td>
<td>1 if RECORDTYPE = FORTRAN</td>
</tr>
<tr>
<td>FABS_CR</td>
<td>Add LF and CR</td>
<td>1 if default, ANY, OR LIST</td>
</tr>
<tr>
<td>FABS_BLK</td>
<td>Do not span blocks</td>
<td>1 if normal</td>
</tr>
<tr>
<td>FABS_FNM</td>
<td>Print file format</td>
<td>FABSC_FIX if FIXED or ORGANIZATION = VIRTUAL, FABSC VAR if VARIABLE or omitted</td>
</tr>
<tr>
<td>FABS_RFM</td>
<td>Record format</td>
<td>FABSC_FIX if FIXED or ORGANIZATION = VIRTUAL, FABSC VAR if VARIABLE or omitted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FABSC STM if STREAM, FABSC VPC if VPC</td>
</tr>
</tbody>
</table>

- **TEMPORARY**:
  - 1 if STATUS = SCRATCH else 0
- **nonexternal file**:
  - With no FILE NAME specified and DISPOSITION="DELETE" specified or implied
  - 2 if terminal file
- **terminal file enabled**:
  - For prompting
  - 1 if CRARRIAGE\_CONTROL = FORTRAN, LIST
  - Default for text files
  - 1 if terminal file enabled for prompting
<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>FARB_SCRIPT</td>
<td>Retrieval window size has the setting specified by the script</td>
<td>Set to WINDOWSIZE</td>
</tr>
<tr>
<td>FARS1_CODEC</td>
<td>Spooling device characteristics set to 0</td>
<td>Returned by RMS</td>
</tr>
<tr>
<td>FARS1_SRH</td>
<td>File sharing is set to more than zero but not less than the window size</td>
<td>As specified by ALLOW</td>
</tr>
<tr>
<td>FARS1.PUT</td>
<td>Allow PUTs to be modified or written</td>
<td>1 if MODIFY OF WRITE</td>
</tr>
<tr>
<td>FARS1.GET</td>
<td>Allow GETs to be modified or read</td>
<td>1 if MODIFY OF READ</td>
</tr>
<tr>
<td>FARS1.DEL</td>
<td>Allow DELETEEs to be deleted or read</td>
<td>1 if MODIFY</td>
</tr>
<tr>
<td>FARS1.UPDATE</td>
<td>Allow UPDATEs to be updated</td>
<td>1 if MODIFY</td>
</tr>
<tr>
<td>FARS1.NIL</td>
<td>Allow no other operations at all</td>
<td>NONE OR SCRATCH</td>
</tr>
<tr>
<td>FARS1.UPI</td>
<td>User-provided interlock</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>FARS1.MSR</td>
<td>Multistream allowed</td>
<td>1 if INDEXED else 0</td>
</tr>
<tr>
<td>FARS1_STS</td>
<td>Completion Status Code</td>
<td>Returned by RMS</td>
</tr>
<tr>
<td>FARS1_XAB</td>
<td>Extended Attribute Block Address</td>
<td>Set to FILE HEADER CHARACTERISTICS (FHC) extended attribute block address to get the longest record length.</td>
</tr>
</tbody>
</table>
RAB Default Settings

Setting of VAX RMS RAB Fields by Higher-Level Language Compilers by Default at FILE OPEN

<table>
<thead>
<tr>
<th>Field</th>
<th>Name</th>
<th>BASIC OPEN</th>
<th>FORTRAN OPEN</th>
<th>PASCAL OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>RABSL_BKT</td>
<td>Bucket code</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>RABSL_CTX</td>
<td>Context</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>RABSL_FAB</td>
<td>FAB address</td>
<td>Set to address of FAB</td>
<td>Set to address of FAB</td>
<td>Set to address of FAB</td>
</tr>
<tr>
<td>RABSL_ISI</td>
<td>Internal stream identifier</td>
<td>Returned by RMS</td>
<td>Returned by RMS</td>
<td>Returned by RMS</td>
</tr>
<tr>
<td>RABSL_KBF</td>
<td>Key buffer address</td>
<td>Address of location containing current or next logical record number for sequential access segmented files</td>
<td>Address of longword containing a 1 if ACCESS = 'DIRECT'</td>
<td>May be modified for individual file operations after the file is opened</td>
</tr>
<tr>
<td>RABSR_KRF</td>
<td>Key of reference</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
<td>May be modified for individual file operations after the file is opened</td>
</tr>
<tr>
<td>RABSR_KSZ</td>
<td>Key size</td>
<td>4 (changed for INDEX.)</td>
<td>Initialized to zero</td>
<td>May be modified for individual file operations after the file is opened</td>
</tr>
<tr>
<td>RABSR_MBC</td>
<td>Multiblock count</td>
<td>Initialized to zero</td>
<td>If blocksize = n use ((n + 511)/512) (n) if BUFFERCOUNT = (n)</td>
<td>Initialzed to zero</td>
</tr>
<tr>
<td>RABSR_MBF</td>
<td>Multibuffer count</td>
<td>As specified by BUFFER</td>
<td>n if BUFFERCOUNT = (n)</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>RABSL_PBF</td>
<td>Prompt buffer address</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>RABSR_PSZ</td>
<td>Prompt buffer size</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
<td>Initialized to zero</td>
</tr>
<tr>
<td>RABSR_RAC</td>
<td>Record access mode</td>
<td>Initialized to zero</td>
<td>RABSK KEY if DIRECT or KEYED</td>
<td>May be modified for individual file operations after the file is opened</td>
</tr>
<tr>
<td>RABSL_RBF</td>
<td>Record address</td>
<td>Initialized to zero</td>
<td>RABSK SEQ if SEQUENTIAL, APPEND, or ACCESS omitted</td>
<td>May be modified for individual file operations after the file is opened</td>
</tr>
<tr>
<td>RABSL_RHB</td>
<td>Record header buffer</td>
<td>Initialized to zero</td>
<td>Set later</td>
<td>May be modified for individual file operations after the file is opened</td>
</tr>
<tr>
<td>RABSL_ROP</td>
<td>Record processing options</td>
<td>Initialized to zero</td>
<td>Set to address of two-byte carriage-control information for terminal files enabled for prompting</td>
<td>Set to address of two-byte carriage-control information for terminal files enabled for prompting</td>
</tr>
</tbody>
</table>
### Setting of VAX RMS RAB Fields by Higher-Level Language Compilers by Default at FILE OPEN (Cont.)

<table>
<thead>
<tr>
<th>Field</th>
<th>Default Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>RABSV_ASY</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>RABSV_BIO</td>
<td>Block I/O</td>
</tr>
<tr>
<td>RABSV_CCO</td>
<td>Cancel CTRL/O</td>
</tr>
<tr>
<td>RABSV_CVT</td>
<td>Convert to uppercase</td>
</tr>
<tr>
<td>RABSV_EOF</td>
<td>End-of-file</td>
</tr>
<tr>
<td>RABSV_FDL</td>
<td>Fast delete</td>
</tr>
<tr>
<td>RABSV_KGE</td>
<td>Key &gt; or =</td>
</tr>
<tr>
<td>RABSV_KGT</td>
<td>Key &gt;</td>
</tr>
<tr>
<td>RABSV_LIM</td>
<td>Limit</td>
</tr>
<tr>
<td>RABSV_LOC</td>
<td>Locate mode</td>
</tr>
<tr>
<td>RABSV_NLK</td>
<td>No lock</td>
</tr>
<tr>
<td>RABSV_NXR</td>
<td>Nonexistent record</td>
</tr>
<tr>
<td>RABSV_PMT</td>
<td>Prompt</td>
</tr>
<tr>
<td>RABSV_PTA</td>
<td>Purge type-ahead</td>
</tr>
<tr>
<td>RABSV_RAH</td>
<td>Read-ahead</td>
</tr>
<tr>
<td>RABSV_RLK</td>
<td>Read locked record allowed</td>
</tr>
<tr>
<td>RABSV_RNE</td>
<td>Read no echo</td>
</tr>
<tr>
<td>RABSV_RNF</td>
<td>Read no filter</td>
</tr>
<tr>
<td>RABSV_TMO</td>
<td>Timeout</td>
</tr>
<tr>
<td>RABSV_TPT</td>
<td>Truncate file after PUT</td>
</tr>
<tr>
<td>RABSV_UIF</td>
<td>Update if record exists</td>
</tr>
<tr>
<td>RABSV_ULK</td>
<td>Manual unlocking</td>
</tr>
<tr>
<td>RABSV_WBH</td>
<td>Write-behind</td>
</tr>
<tr>
<td>RABSV_RSZ</td>
<td>Record size</td>
</tr>
</tbody>
</table>

- Initialized to zero
- 1 if ACCESS APPEND
- 1 if APPEND
- May be modified for individual file operations after the file is opened

**NOTE:**

- The table above lists the default settings for VAX RMS RAB fields as used by higher-level language compilers at FILE OPEN time. Each field and its default setting are listed, with exceptions noted for specific conditions or operations. Some fields may be modified for individual file operations after the file is opened, as indicated in the table.

---

11-12
| RAB$1_STS | Completion status code | Returned by RMS |
| RAB$8_TMO | Timeout period | Initialized to zero |
| RAB$1_UBF | User record area address | Set to address of buffer |
| RAB$1_USZ | User record area size | Same as RAB$M_MRS |
| | | Returned by RMS |
| | | Initialized to zero |
| | | Set to buffer address after file is opened |
| | | Set to size of buffer after file is opened |
## Setting of VAX-11 RMS Extended Attribute Block (XAB) Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Name*</th>
<th>BASIC OPEN Keyword and Value</th>
<th>FORTRAN and PASCAL OPEN Keyword and Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>XABKEY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XAB$B_DTP</td>
<td>Data type of the key</td>
<td>Set to data type of the key</td>
<td>Set to data type of the key</td>
</tr>
<tr>
<td>XAB$B_FLG</td>
<td>Key options flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XAB$V_CHG</td>
<td>Changes allowed</td>
<td>As specified by CHANGES</td>
<td>0 if key is 0, else 1</td>
</tr>
<tr>
<td>XAB$V_DUP</td>
<td>Duplicates allowed</td>
<td>As specified by DUPLICATES</td>
<td>0 if key is 0, else 1</td>
</tr>
<tr>
<td>XAB$W_POSO</td>
<td>Key position</td>
<td>Position of key in indexed file record</td>
<td>Position of key in indexed file record</td>
</tr>
<tr>
<td>XAB$B_REF</td>
<td>Key of reference</td>
<td>Primary key is zero, first alternate key is one, second alternate key is two, and so on...</td>
<td>Primary key is zero, first alternate key is one, second alternate key is two, and so on...</td>
</tr>
<tr>
<td>XAB$B_SIZ0</td>
<td>Key size</td>
<td>Size of key</td>
<td>Size of key</td>
</tr>
<tr>
<td>XABFHC</td>
<td></td>
<td>Initialized</td>
<td>Initialized</td>
</tr>
<tr>
<td>XABSUM</td>
<td></td>
<td>Initialized</td>
<td>Initialized</td>
</tr>
<tr>
<td>NAM block</td>
<td></td>
<td>Initialized</td>
<td>Initialized</td>
</tr>
</tbody>
</table>
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.
- 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 selective on copy of file attributes
  backup always copies all attributes

11-15
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

<table>
<thead>
<tr>
<th>MULTI-BLOCK COUNT</th>
<th>MULTIBUFFER COUNTS</th>
<th>NETWORK BLOCK COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 0</td>
<td>Indexed 0</td>
<td>Disk 0</td>
</tr>
<tr>
<td>System 16</td>
<td>Relative 0</td>
<td>Magtape 0</td>
</tr>
<tr>
<td></td>
<td>Sequential 0</td>
<td>Unit 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Record 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

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

- 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

<table>
<thead>
<tr>
<th>MULTI-BLOCK COUNT</th>
<th>MULTIBUFFER COUNTS</th>
<th>NETWORK BLOCK COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed Relative</td>
<td>Disk Magtape Unit</td>
<td>0</td>
</tr>
<tr>
<td>Sequential</td>
<td>Record</td>
<td>0</td>
</tr>
<tr>
<td>Process 24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>System 16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>Extend Quantity</td>
<td>0</td>
</tr>
<tr>
<td>Process 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>System 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$ SET RMS_DEFAULT/EXTEND=50/INDEXED/BUFFER_COUNT=24
$ SHOW RMS_DEFAULT

<table>
<thead>
<tr>
<th>MULTI-BLOCK COUNT</th>
<th>MULTIBUFFER COUNTS</th>
<th>NETWORK BLOCK COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed Relative</td>
<td>Disk Magtape Unit</td>
<td>0</td>
</tr>
<tr>
<td>Sequential</td>
<td>Record</td>
<td>0</td>
</tr>
<tr>
<td>Process 24</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>System 16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prolog</td>
<td>Extend Quantity</td>
<td>0</td>
</tr>
<tr>
<td>Process 0</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>System 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

11-18
SPECIFYING RUN-TIME OPTIONS

All RMS options are set within the RMS control blocks that are maintained in the user PO 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 FDLSPARSE 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATION</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>BEST_TRY_CONTIGUOUS</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>BUCKET_SIZE</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>CONTIGUOUS</td>
<td>yes/no</td>
<td>Not worth considering on public volume but may be useful for dedicated drives</td>
</tr>
<tr>
<td>Exact_Positioning</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>EXTENSION</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>POSITION</td>
<td>qualifier</td>
<td></td>
</tr>
<tr>
<td>VOLUME</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYNCHRONOUS</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>BLOCK_IO</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>BUCKET_CODE</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>CONTEXT</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>END_OF_FILE</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>FAST_DELETE</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>FILL_BUCKETS</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>KEY_GREATER_EQUAL</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>KEY_GREATER_THAN</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>KEY_LIMIIT</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>KEY_OF_REFERENCE</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>LOCATE_MODE</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>LOCK_ON_READ</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>LOCK_ON_WRITE</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>MANUAL_UNLOCKING</td>
<td>yes/no</td>
<td></td>
</tr>
<tr>
<td>MULTIBLOCK_COUNT</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>MULTIBUFFER_COUNT</td>
<td>number</td>
<td></td>
</tr>
</tbody>
</table>

(NONEXISTENT_RECORD READ_AHEAD READ_REGARDLESS TIMEOUT_ENABLE TIMEOUT_PERIOD TRUNCATE_ON_PUT TT_CANCEL_CONTROL_0 TT_PROMPT TT_PURGE_TYPE_AHEAD TT_READ_NOECHO TT_READ_NOFILTER TT_UPCASE_INPUT UPDATE_IF WAIT_FOR_RECORD WRITE_BEHIND)

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)[-] : ?

11-22
Legal KEY 0 Secondary Attributes

<table>
<thead>
<tr>
<th>CHANGES</th>
<th>yes/no</th>
<th>LEVEL1_INDEX_AREA</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_AREA</td>
<td>number</td>
<td>NAME</td>
<td>string</td>
</tr>
<tr>
<td>DATA_FILL</td>
<td>number</td>
<td>NULL_KEY</td>
<td>yes/no</td>
</tr>
<tr>
<td>DATA_KEY_COMPRESSION</td>
<td>yes/no</td>
<td>NULL_VALUE</td>
<td>char/num</td>
</tr>
<tr>
<td>DATA_RECORD_COMPRESSION</td>
<td>yes/no</td>
<td>POSITION</td>
<td>number</td>
</tr>
<tr>
<td>DUPLICATES</td>
<td>yes/no</td>
<td>PROLOG</td>
<td>number</td>
</tr>
<tr>
<td>INDEX_AREA</td>
<td>yes/no</td>
<td>TYPE</td>
<td>keyword</td>
</tr>
<tr>
<td>INDEX_COMPRESSION</td>
<td>yes/no</td>
<td>SEGn_LENGTH</td>
<td>number</td>
</tr>
<tr>
<td>INDEX_FILL</td>
<td>number</td>
<td>SEGn_POSITION</td>
<td>number</td>
</tr>
<tr>
<td>LENGTH</td>
<td>number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>yes/no</td>
</tr>
<tr>
<td>GET</td>
<td>yes/no</td>
</tr>
<tr>
<td>MULTISTREAM</td>
<td>yes/no</td>
</tr>
<tr>
<td>PROHIBIT</td>
<td>yes/no</td>
</tr>
<tr>
<td>PUT</td>
<td>yes/no</td>
</tr>
<tr>
<td>UPDATE</td>
<td>yes/no</td>
</tr>
<tr>
<td>USER_INTERLOCK</td>
<td>yes/no</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVICE</td>
<td>string</td>
</tr>
<tr>
<td>SOURCE</td>
<td>keyword</td>
</tr>
<tr>
<td>TARGET</td>
<td>keyword</td>
</tr>
</tbody>
</table>

(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) (Keyword)[-]: ?

Legal CONNECT Secondary Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYNCHRONOUS</td>
<td>yes/no</td>
<td>NOLOCK</td>
</tr>
<tr>
<td>BLOCK_IO</td>
<td>yes/no</td>
<td>NONEXISTENT_RECORD</td>
</tr>
<tr>
<td>BUCKET_CODE</td>
<td>number</td>
<td>READ_AHEAD</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>number</td>
<td>READ REGARDLESS</td>
</tr>
<tr>
<td>END_OF_FILE</td>
<td>yes/no</td>
<td>TIMEOUT_ENABLE</td>
</tr>
<tr>
<td>FAST_DELETE</td>
<td>yes/no</td>
<td>TIMEOUT PERIOD</td>
</tr>
<tr>
<td>FILL_BUCKETS</td>
<td>yes/no</td>
<td>TRUNCATE ON PUT</td>
</tr>
<tr>
<td>KEY_GREATER_EQUAL</td>
<td>yes/no</td>
<td>TT_CANCEL_CONTROL_0</td>
</tr>
<tr>
<td>KEY_GREATER_THAN</td>
<td>yes/no</td>
<td>TT_PROMPT</td>
</tr>
<tr>
<td>KEY_LIMIT</td>
<td>yes/no</td>
<td>TT_PURGE_TYPE_AHEAD</td>
</tr>
<tr>
<td>KEY_OF_REFERENCE</td>
<td>number</td>
<td>TT_READ_NOECHO</td>
</tr>
<tr>
<td>LOCATE MODE</td>
<td>yes/no</td>
<td>TT_READ_NOFILTER</td>
</tr>
<tr>
<td>LOCK_ON_READ</td>
<td>yes/no</td>
<td>TT_UPCASE_INPUT</td>
</tr>
<tr>
<td>LOCK_ON_WRITE</td>
<td>yes/no</td>
<td>UPDATE IF</td>
</tr>
<tr>
<td>MANUAL_UNLOCKING</td>
<td>yes/no</td>
<td>WAIT_FOR_RECORD</td>
</tr>
</tbody>
</table>
MULTIBLOCK_COUNT: number
MULTIBUFFER_COUNT: number
WRITE_BEHIND: yes/no

Enter CONNECT Attribute (Keyword){-} : LOCK_ON_WRITE

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: 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:

\[ \text{RMSS}_{-\text{fff}}[\ldots\text{f}] \]

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

\[ \text{RMSS}_{-\text{EOF}} \quad 0001827A \quad \text{(Hex)} \]
\[ \quad 98938 \quad \text{(Decimal)} \]

The symbolic names and associated numeric values are defined in an $RMSDEF$ module, the location of which varies among the programming languages.
<table>
<thead>
<tr>
<th>Language</th>
<th>SYS$LIBRARY:</th>
<th>External Reference to Symbolic Name Resolved at</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>STARLET.OLB(1)</td>
<td>Link --- for V3.0 and above use: SYS$LIBRARY:STARLET.TLB</td>
</tr>
<tr>
<td>COBOL</td>
<td>STARLET.OLB(1)</td>
<td>Link</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>FORSYSDEF.TLB(2)</td>
<td>Compile</td>
</tr>
<tr>
<td>PASCAL</td>
<td>STARLET.PEN(3)</td>
<td>Compile</td>
</tr>
<tr>
<td>MACRO</td>
<td>STARLET.MLB(1)</td>
<td>Assemble</td>
</tr>
</tbody>
</table>

Use the following DCL library utilities to extract a listing from one of these libraries.

(1) `S$LIBRARY/EXTRACT = S$RMSDEF/OUT=filespec - SYS$LIBRARY:STARLET.MLB`

(2) `S$LIBRARY/EXTRACT=$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.
.MACRO $RMSDEF,$GBL
$DEFINI RMS,$GBL
ADD RMS$ BUSY
ADD RMS$ FILEPURGED

$SEQU RMS$_FACILITY 1
$SEQU RMS$SV_STVSTATUS 14
$SEQU RMS$_SUC 65537
$SEQU RMS$_NORMAL 65537
$SEQU RMS$_STALL 98305
$SEQU RMS$_PENDING 98313
$SEQU RMS$_OK_DUP 98321
$SEQU RMS$_OK_IDX 98329
$SEQU RMS$_OK_RLK 98337
$SEQU RMS$_OK_RRL 98345

; OK_RRV
$SEQU RMS$_KFF 98353
$SEQU RMS$_OK_ALK 98361
$SEQU RMS$_OK_DEL 98369
$SEQU RMS$_OK_RNF 98377
$SEQU RMS$_OK_LIM 98385
$SEQU RMS$_OK_NOP 98393
$SEQU RMS$_OK_WAT 98401
$SEQU RMS$_CRE_STS 98409
$SEQU RMS$_OK_RULK 98417
$SEQU RMS$_CONTROL 67153
$SEQU RMS$_CONTROLO 67081
$SEQU RMS$_CONTROLY 67089
$SEQU RMS$_CREATED 67097
$SEQU RMS$_SUPERSEDE 67121
$SEQU RMS$_OVRDSKQUOTA 67177
$SEQU RMS$_FILEPURGED 67193
$SEQU RMS$_BOF 98712
$SEQU RMS$_RNL 98720
$SEQU RMS$_RTB 98728
$SEQU RMS$_TMO 98736
$SEQU RMS$_TNS 98744
$SEQU RMS$_BES 98752
$SEQU RMS$_PES 98760
$SEQU RMS$_ACT 98906
$SEQU RMS$_DEL 98914
$SEQU RM $INCOMPSHR 98922
$SEQU R$S[tmp] 98930
$SEQU RMS$_EOA 98938
$SEQU RMS$_FE 98946
$SEQU RMS$_FLK 98954
$SEQU RMS$_FNF 98962
$SEQU RMS$_PRV 98970
$SEQU RMS$_REX 98978
$SEQU RMS$_RLK 98986
$SEQU RMS$_RNF 98994
$SEQU RMS$_WLK 99002
!!** 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$_FACILITY = '00000001'X
PARAMETER RMS$_STVSTATUS = '00000000E'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$_INCOMP$HR = '0001826A'X
PARAMETER RMS$_DNR = '00018272'X
PARAMETER RMS$_EOF = '0001827A'X
PARAMETER RMS$_FEX = '00018282'X
PARAMETER RMS$_PLK = '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 **

```pascal
(** 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.) *)

. . .
```

---

12-6
Testing Completion Status

Testing Completion Status -- Calling RMS Procedures

1. Call procedure as Function
2. Declare function name EXTERNAL LONG function name
3. Declare variable to receive status code as function result LONG STAT
4. If you wish to test for specific status code (optional) EXTERNAL LONG CONSTANT symbol_name

BASIC

COBOL Function
EXTERNAL LONG function name
LONG STAT
EXTERNAL LONG CONSTANT symbol_name

FORTRAN Function
INTEGER*4 name
INTEGER*4 STAT
01 STAT COMP PIC S9(9)
INCLUDE '($RMSDEF)' [INHERIT ('SYSLIBRARY:STARLET')]

PASCAL Function
[INHERIT ('SYSLIBRARY:STARLET')]
STAT : INTEGER;
01 symbol_name PIC S9(9)
COMP VALUE IS EXTERNAL symbol_name
A programmer can test for:

1. Success or failure \(\iff\) overall test -- binomial evaluation
2. Specific conditions \(\iff\) test of specific condition codes (symbolic names)

Format of Status Value

<table>
<thead>
<tr>
<th>CONTROL</th>
<th>FACILITY NUMBER</th>
<th>MESSAGE NUMBER</th>
<th>SEVERITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>27</td>
<td>16 15</td>
<td>03 02 00</td>
</tr>
</tbody>
</table>

Severity Codes

<table>
<thead>
<tr>
<th>Severity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Warning</td>
</tr>
<tr>
<td>1</td>
<td>Success</td>
</tr>
<tr>
<td>2</td>
<td>Error</td>
</tr>
<tr>
<td>3</td>
<td>Information</td>
</tr>
<tr>
<td>4</td>
<td>Severe Error</td>
</tr>
<tr>
<td>5-7</td>
<td>Reserved for Future Use</td>
</tr>
</tbody>
</table>
Example 1. Testing Specific Condition Codes

(Basic)

```basic
01 EXTERNAL LONG FUNCTION SYSSGET
02 EXTERNAL LONG CONSTANT RMS$_EOF
03 DECLARE LONG STAT
04 .
05 .
06 STAT = SYSSGET(...)
07 handle error IF (STAT = RMS$_EOF)
08 .
09 .
```

(Cobol)

```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 'SYSSGET' USING ... GIVING STAT.
19 IF STAT IS EQUAL TO RMS$_EOF THEN handle error.
20 .
```

(Fortran)

```fortran
21 INTEGER*4 SYSSGET, STAT
22 INCLUDE '($RMSDEF)'
23 .
24 .
25 STAT = SYSSGET(...)  
26 IF (STAT .EQ. RMS$_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, RO
43 BEQLU handle_error
44 ...

12-10
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 CALL 'SYSSGET' USING ... GIVING STAT.
18 IF STAT IS FAILURE THEN  handle error.
19 .
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 .
```

12-11
Example 2 (Sheet 2 of 2)

PASCAL

```
31 (INHERIT ('SYSLIBRARY: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 ...
```

12-12
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$SETSF) 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 RO 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

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TYPE</th>
<th>LENGTH OF STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ADDRESS OF STRING</td>
</tr>
</tbody>
</table>

:ABC

12-15
**Format of an Argument List**

![Diagram showing the format of an argument list]

**Default Argument Passing and Receiving Mechanisms**

<table>
<thead>
<tr>
<th>Language</th>
<th>Numeric</th>
<th>Character</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>Reference</td>
<td>Descriptor</td>
<td>Descriptor</td>
</tr>
<tr>
<td>COBOL</td>
<td>Reference</td>
<td>Reference</td>
<td>N/A</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>Reference</td>
<td>Descriptor</td>
<td>Reference</td>
</tr>
<tr>
<td>MACRO</td>
<td>No Default</td>
<td>No Default</td>
<td>No Default</td>
</tr>
<tr>
<td>PASCAL</td>
<td>Reference</td>
<td>Reference</td>
<td>Reference*</td>
</tr>
</tbody>
</table>

* Except conformant arrays
### Argument Passing Specifiers

#### Mechanism Specifier

<table>
<thead>
<tr>
<th>BASIC</th>
<th>COBOL</th>
<th>FORTRAN</th>
<th>PASCAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BY VALUE</td>
<td>BY VALUE</td>
<td>%VAL</td>
<td>%IMMED</td>
</tr>
<tr>
<td>BY REF</td>
<td>BY REFERENCE</td>
<td>%REF</td>
<td>%REF</td>
</tr>
<tr>
<td>OR</td>
<td>BY CONTENT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BY DESC</td>
<td>BY DESCRIPTOR</td>
<td>%DESCR</td>
<td>%DESCR</td>
</tr>
<tr>
<td>BY DESC</td>
<td>BY DESCRIPTOR</td>
<td>%DESCR</td>
<td>%STDESCR</td>
</tr>
</tbody>
</table>

Passed by Value

Passed by Reference

Passed by Descriptor

Passed by String Descriptor
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; EXTER:

    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;

abc = '35';
STAT. = SUB1 (abc);

MACRO:  
abc: .ASCIID /35/

PUSHAQ abc
CALLS #1, SUB1
RMS routines -- The passing mechanism is identified in the second sentence of the argument description.

<table>
<thead>
<tr>
<th>Passing Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate Value</td>
<td>&quot;The value of&quot; or omission of the word &quot;address&quot;</td>
</tr>
<tr>
<td>Reference</td>
<td>&quot;Address of&quot;</td>
</tr>
<tr>
<td>Descriptor</td>
<td>&quot;Address of a descriptor&quot;</td>
</tr>
</tbody>
</table>

Example 6. Passing Mechanism for SYS$SETDDIR Arguments

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Passing Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_dir_addr</td>
<td></td>
</tr>
<tr>
<td>length_addr</td>
<td></td>
</tr>
<tr>
<td>cur_dir_addr</td>
<td></td>
</tr>
</tbody>
</table>
Trailing Optional Arguments -- RMS Services

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, SYSSETDDIR), 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 SYSS$SETDDIR

(Basic)

```
10  This program calls the RMS procedure $SETDDIR to
    change the default directory for the process.
```

(Cobol)

```
IDENTIFICATION DIVISION.

PROGRAM-ID. SETDDIR.

*  This program calls the RMS procedure $SETDDIR to change
*  the default directory for the process.

DATA DIVISION.
WORKING-STORAGE SECTION.

01 DIRECTORY PIC X(17)
   VALUE '[COURSE.PROG.COB]'.

01 STAT
   PIC S9(9)
   VALUE 0.

PROCEDURE DIVISION.
BEGIN.
    CALL 'SYSS$SETDDIR' USING BY DESCRIPTOR DIRECTORY
        BY VALUE 0
        BY VALUE 0
    GIVING STAT.
    IF STAT IS FAILURE DISPLAY 'ERROR'.
    STOP RUN.
```

(Fortran)

```
This program calls the RMS procedure $SETDDIR to change
the default directory for the process.

IMPLICIT INTEGER (A - Z) OR INTEGER SYSS$SETDDIR, STAT
CHARACTER*17 DIR '/[COURSE.PROG.FDR]'/
STAT = SYSS$SETDDIR (DIR,)
IF (.NOT. STAT) TYPE '*', 'ERROR'
END
```
Example 7 (Sheet 2 of 2)

PASCAL

PROGRAM setddir( OUTPUT);
(* $SETDDIR.PAS *)
(* This program calls the RMS procedure $SETDDIR to *)
(* change the default directory for the process. *)

TYPE word_integer = [WORD] 0..65535;

VAR dir_status: INTEGER;

FUNCTION SYS$SETDDIR(
  new_dir: [CLASS_S] PACKED ARRAY [1..u:INTEGER] OF CHAR;
  old_dir_len: word integer := $IMMED 0;
  old_dir: VARYING [line2] OF CHAR := $IMMED 0);
  INTEGER: EXTERN;
BEGIN
  dir_status := SYS$SETDDIR('([COURSE.V4PROG.PAS]');
  IF NOT ODD( dir_status )
  THEN WRITELN( 'Error in SYS$SETDDIR call.' )
END.

MACRO

SETDDIR.MAR

.TITLE SETDDIR

This program calls the RMS procedure $SETDDIR to change
the default directory for the process.

.PSEC Nonshared Data Pic, Noexe, Long

.DIR: ./COURSE_V4PROG.MAC/

.PROC Code Pic, Shr, Nowrt, Long

.ENTER start,"m"

.PUSHL $0

.RET

.END start

12-24
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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Open Keyword</th>
<th>Value</th>
<th>Language Function May Be Written In</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>USEROPEN</td>
<td>function-name</td>
<td>BASIC</td>
</tr>
<tr>
<td>COBOL</td>
<td>not available</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>USEROPEN</td>
<td>function-name</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>PASCAL</td>
<td>USER_ACTION</td>
<td>function-name</td>
<td>PASCAL</td>
</tr>
</tbody>
</table>

The USEROPEN function has three arguments:

1. Arg 1 -- FAB address passed to user by RMS
2. Arg 2 -- RAB address
3. Arg 3 -- File # - BASIC
   Unit # - FORTRAN
   Filename identifier - PASCAL

   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.
None provided in system library. Set up your own and 'include' it in the source file.

```
%INCLUDE 'FABRABDEF.BAS'
```

```
FORTRAN

INCLUDE '($FABDEF)'

INCLUDE '($RABDEF)'

SYS$LIBRARY: FORSYSDEF.TLB
```

```
PASCAL

(INHERIT ('SYS$LIBRARY: STARLET'))
```

```
Now included in BASIC $STARTLET.TLB
```

For BASIC and FORTRAN USE OF INCLUDE Function only the NAME of the LIBRARY is specified for INDEX FILE.

```
FOR BASIC USE OF INCLUDE Function only the NAME of the LIBRARY is specified for INDEX FILE.
```

Also included in the DIGITAL FORTRAN UPER ACTION Parameter differences from the BASIC UPER ACTION.

```
The DIGITAL FORTRAN USE OF INCLUDE Function only the NAME of the LIBRARY is specified for INDEX FILE.
```

To access library of a library or keep catalog in a library to the
```
Example 1. USEROPEN Alternative (if available) or Regular I/O

BASIC (Sheet 1 of 5)

1 10 OPTION TYPE = EXPLICIT  

2 !  
3 ! This BASIC program uses USEROPEN function to open  
4 ! an indexed file. Thereafter, all access to the file is  
5 ! done using regular BASIC I/O. User is prompted for  
6 ! SEQ_NO of record to be retrieved randomly and given  
7 ! the option of deleting any record retrieved.  
8 !  
9 ! Program assumes FILE SHARING and RECORD LOCKING  
10 !  
11 ON ERROR GO TO err_check  
12  
13 DECLARE SINGLE CONSTANT TIME_WAIT=10.0  
14 DECLARE STRING CONSTANT RIGHT_JUSTIFY = "########"  
15  
16 RECORD ACCOUNTTYPE  
17 STRING SEQ_NO=7  
18 STRING LAST_NAME=15  
19 STRING FIRST_NAME=10  
20 STRING SOC_SEC=9  
21 STRING STREET=18  
22 STRING CITY=14  
23 STRING STATE=2  
24 STRING ZIP_CODE=5  
25  
26 END RECORD ACCOUNTTYPE  
27  
28 MAP (INREC) ACCOUNTTYPE IN_REC  
29 MAP (FILE_NAME) STRING FILENAME=80  
30 MAP (KEYVALUE) STRING KEY_VALUE=7  
31 MAP (DELEFLAG) STRING DELETE_FLAG=1  
32  
33 DECLARE LONG KEY_IN  
34 EXTERNAL SUB LIB$INIT_TIMER,&  
35 LIB$SHOW_TIMER,&  
36 LIB$WAIT  
37  
38 start:  
39  
40 CALL LIB$INIT_TIMER()  
41 INPUT 'Enter filename';filename  
42 OPEN filename FOR INPUT AS FILE #1, &  
43 ORGANIZATION Indexed, &  
44 MAP INREC, &  
45 ACCESS MODIFY, &  
46 ALLOW MODIFY, &  
47 USEROPEN OPENFILE  
48
PRINT "***************
PRINT " Hit <CR> or enter zero to stop run"
PRINT "***************

INPUT 'Enter SEQ_NO':KEY_IN
WHILE (KEY_IN > 0)
  KEY_VALUE = FORMATS(KEY_IN,RIGHT_JUSTIFY,100)
  GET #1%, KEY #0% EQ KEY_VALUE
  PRINT IN_REC::SEQ_NO;TAB(9);IN_REC::SOC_SEC;TAB(20);
  IN_REC::LAST_NAME
  INPUT 'Do you wish to delete this record? (Y/<CR>)': &
  DELETE_FLAG
  IF (DELETE_FLAG = 'Y' OR DELETE_FLAG = 'y') THEN
    DELETE #1%
  NEXT

CLEANUP:
CLOSE #1%
CALL LIB$SHOW_TIMER()
GOTO done

ERR_CHECK:
! ! ERR = 155 ---- record not found on random get
! ! ERR = 154 ---- record/bucket locked
! ! SELECT ERR
CASE 155
  PRINT "Record NOT FOUND with SEQ_NO = ":KEY_VALUE
  RESUME 150
CASE 154
  PRINT "Record currently LOCKED", &
  'will try again shortly"
  CALL LIB$WAIT (TIME_WAIT)
  RESUME 100
CASE ELSE
  ON ERROR GO TO 0
END SELECT

done: END
FUNCTION LONG OPENFILE (FAB$Type FAB,&
RAB$Type RAB,&
LONG channel)

EXTERNAL LONG FUNCTION SYS$OPEN
EXTERNAL LONG FUNCTION SYS$CONNECT

! Include record definitions of FAB$Type and RAB$Type
! and external constants

%INCLUDE 'FABRABDEF.BAS'

DECLARE LONG ret_stat

! 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$M_PUT OR FAB$W_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 inserted in source after
! SYS$OPEN call but prior to SYS$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)

ret_stat = SYS$OPEN(FAB,)
CALL LIB$STOP(ret_stat BY VALUE) IF (ret_stat AND 1%)=0%

! Specify number of local buffers you want RMS to allocate on CONNECT
RAB::RAB$B_MBF = 3

! Enable any record processing options to be used for entire run
RAB::RAB$L_ROP = (RAB::RAB$L_ROP OR RAB$M_FDL) \ fast delete

ret_stat = SYS$CONNECT(RAB,)
CALL LIB$STOP(ret_stat BY VALUE) IF (ret_stat AND 1%)=0%

OPENFILE = ret_stat

END FUNCTION
LONG rab$1_stv
RFA rab$sr_rfa
WORD fill
LONG rab$1_ctx
WORD fill
BYTE rab$b_rac
BYTE rab$b_tmo
WORD rab$w_usz
WORD rab$w_rsz
LONG rab$1_ubf
LONG rab$1_rbf
LONG rab$1_thb
VARIANT
CASE
LONG rab$1_kbf
BYTE rab$b_ksz
CASE
LONG rab$1_pbff
BYTE rab$b_pszz
END VARIANT
BYTE rab$b_krf
BYTE rab$b_mbf
BYTE rab$b_mbc
VARIANT
CASE
LONG rab$1_bkt
CASE
LONG rab$1_dct
END VARIANT
LONG rab$1_fab
LONG rab$1_rab
END RECORD
+ declarations of FAB and RAB CONSTANTS
+ EXTERNAL BYTE CONSTANT FAB$M_DEL,&
FAB$M_GET,&
FAB$M_PUT,&
FAB$M_UPD,&
FAB$M_DFW,&
FAB$M_SHRPUT,&
FAB$M_SHRGET,&
FAB$M_SHRDEL,&
FAB$M_SHRUPD,&
FAB$M_MSE
EXTERNAL BYTE CONSTANT RAB$C_KEY,&
RAB$C_SEQ,&
RAB$C_RFA,&
RAB$M_FDL
COBOL (Sheet 1 of 3)

* COBOL PROGRAM which opens an indexed file. All access
to the file is done using regular COBOL I-O. User
is prompted for SEQ_NO of record to be retrieved
randomly and given option of deleting any record
retrieved.

* COBOL currently does not support a USEROPEN function call
from its language.

* Program assumes FILE SHARING and RECORD LOCKING

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOPEN_IDX.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-COMTROL.
SELECT INDEX1 ASSIGN FILENAME
ORGANIZATION IS INDEXED
ACCESS MODE IS RANDOM RESERVE 3
RECORD KEY IS SEQ_NO
ALTERNATE RECORD KEY IS LAST_NAME
WITH DUPLICATES.

I-O-CONTROL.
APPLY DEFERRED-WRITE
EXTENSION 10
WINDOW 7 ON INDEX1.

DATA DIVISION.
FILE SECTION.
FD INDEX1 VALUE OF ID IS FILENAME.

01 INREC.
  02 SEQ_NO PIC X(7).
  02 LAST_NAME PIC X(15).
  02 FIRST_NAME PIC X(10).
  02 SOC_SEC PIC X(9).
  02 STREET PIC X(18).
  02 CITY PIC X(14).
  02 STATE PIC X.
  02 ZIP_CODE PIC X(5).

WORKING-STORAGE SECTION.
* CONSTANTS
  01 TIME_WAIT USAGE COMP-1 VALUE IS 10.0.
* VARIABLES
  01 KEY_VALUE PIC X(7) JUSTIFIED RIGHT.
  01 DELLETE_FLAG PIC X VALUE 'N'.
  01 RET_STATUS PIC S9(9) COMP.
  01 PROG_STAT PIC 9.
  88 NO_ERROR VALUE 1.
  88 SOME_ERROR VALUE 2.
  88 WAIT_READ_AGAIN VALUE 3.
  88 END_OF_INPUT VALUE 4.
  01 FILENAME PIC X(80) VALUE 'XXXX'.
  01 RMSS_RNF PIC 9(9) COMP VALUE EXTERNAL RMSS_RNF.
01  RMSS_RLK  PIC 9(9) COMP VALUE EXTERNAL RMSS_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+2 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 RMSS_RNF
      DISPLAY "NO RECORD WITH SEQ_NO = ", KEY_VALUE
    WHEN RMSS_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     IF PROG_STAT IS EQUAL TO 1 THEN
108           DISPLAY SEQ_NO," ",SOC_SEC," ",LAST_NAME
109           PERFORM 200-DELETE-RECORD
110         END-IF.
111
112         PROCEDURE DIVISION.
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 IDX1 INVALID KEY
119         DISPLAY "Bad DELETE".
120         MOVE 0 TO PROG_STAT.
121         CALL 'LIB$SHOW_TIMER'.
122         CLOSE IDX1.
123         STOP RUN.

124         250-CLEANUP.
125         PERFORM 200-CLEANUP.
126         STOP RUN.
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_INDEX

IMPLICIT NONE

REAL TIME_WAIT
PARAMETER (TIME_WAIT = 10.0)

STRUCTURE /ACCOUNT_STRUCT/
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_STRUCT/ 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/
1 VALUE 1 = no error
1 VALUE 2 = some error
1 VALUE 3 = wait-read-again

EXTERNAL OPENFILE

INCLUDE '($RMSDEF)'

CALL LIB$INIT_TIMER()
WRITE (6,1)
READ (5,2) len_filename,FILENAME

OPEN (UNIT=IUNIT, 1
FILE=FILENAME(len_filename), 2
STATUS='OLD', 3
ORGANIZATION='INDEXED', 4
ACCESS='KEYED', 5
FORM='UNFORMATTED', 6
RECORDTYPE='FIXED', 7
SHARED,
USEROPEN=OPENFILE)

WRITE (6,6) ! directions to stop run
WRITE (6,5)
WRITE (6,6)

WRITE (6,3)
READ (5,4) KEY_IN

DO WHILE (KEY_IN .GT. 0)
  ! Convert integer KEY_IN to right justified character string KEY_VALUE
  ! using internal read
  IF (PROG_STAT .NE. 3)
    WRITE (UNIT=KEY_VALUE,FMT='(I7)') KEY_IN
  1
  READ (IUNIT,KEY=KEY_VALUE,KEYID=0,IOSTAT=RET_STATUS) IN_REC
  IF (RET_STATUS .GT. 0) THEN
    PROG_STAT = 2
    CALL ERRSNS(RMS_STS,RMS_STV,IUNIT,)
    IF (RMS_STS .E . RMS$ RNF) THEN
      WRITE (6,11) KEY_VALUE
    ELSE
      IF (RMS_STS .EQ. RMS$ RLK) THEN
        WRITE (6,12)
        CALL LIB$WAIT (TIME_WAIT)
        PROG_STAT = 3
      ELSE
        CALL LIB$STOP($VAL(RMS_STS),$VAL(RMS_STV))
      END IF
    END IF
    END IF
  ELSE
    PROG_STAT = 1
  END IF
FORTRAN (Sheet 3 of 4)

104 IF (PROG_STAT .EQ. 1) THEN
105 WRITE (6,*) IN_REC.SEQ_NO, ',IN_REC.SOC_SEC, ',
106 IN_REC.LAST_NAME
107 1
108 WRITE (6,14)
109 READ (5,15) DELETE_FLAG
110 IF (DELETE_FLAG .EQ. 'Y' .OR.
111 DELETE_FLAG .EQ. 'y') THEN
112 DELETE (UNIT=IUNIT)
113 END IF
114 END IF  ! prog_stat = 1
115
116 IF (PROG_STAT .NE. 3) THEN
117 WRITE (6,3)
118 READ (5,4) KEY_IN
119 END IF  ! prog_stat .ne. 3
120 END DO
121
122 CLOSE (IUNIT)
123 CALL LIB$SHOW_TIMER()
124 CALL EXIT
125
126 1 FORMAT ('$Enter filename: ')
127 2 FORMAT (A)
128 3 FORMAT ('$Enter SEQ_NO: ')
129 4 FORMAT (I)
130 5 FORMAT (' Hit <CR> or enter zero to stop run')
131 6 FORMAT ('******************************************
132 11 FORMAT (' Record NOT FOUND with SEQ_NO = ',A)
133 12 FORMAT (' Record currently LOCKED ',
134 1 14 FORMAT ('$Do you wish to delete this record? (Y/<CR>): ')
135 15 FORMAT (A)
136 END
INTEGER FUNCTION OPENFILE (FAB, RAB, IUNIT)

IMPLICIT NONE

INCLUDE '($FABDEF)'
INCLUDE '($RABDEF)'
INCLUDE '($RMSDEF)'

RECORD /FABDEF/   FAB
RECORD /RABDEF/   RAB

INTEGER IUNIT,
1 RET_STATUS,
2 SYS$OPEN,
3 SYS$CONNECT,

! Extension $ blocks if file is extended
FAB.FABS$W_DEQ = 10

! File access desired for USER
FAB.FABS$B_FAC = FAB.FABS$B_FAC .OR. FABS$M_DEL .OR. FABS$M_GET
1 .OR. FABS$M_PUT .OR. FABS$M_UPD

! File options desired
FAB.FABS$L_FOP = FAB.FABS$L_FOP .OR. FABS$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 ith global buffers enabled
! The following statement must be inserted in source after
! SYS$OPEN call but prior to SYS$CONNECT call.
! FAB.FABS$W_GBC = ?

! Sharing attributes - what others can do or set TO FABS$M_SHRNIL
FAB.FABS$B_SHR = FAB.FABS$B_SHR .OR. FABS$M_SHRNIL
1 .OR. FABS$M_SHRGET .OR. FABS$M_SHRDEL
2 .OR. FABS$M_SHRUPD

! RET_STATUS = SYS$OPEN(FAB,)

! Specify number of local buffers you want RMS to allocate on CONNECT
RAB.RABS$B_MBF = 3

! Enable any record processing options to be used for entire run
RAB.RABS$L_ROP = RAB.RABS$L_ROP .OR. RABS$M_FDL  ! fast delete

IF (RET_STATUS) RET_STATUS = SYS$CONNECT(RAB,)

IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))

OPENFILE = RET_STATUS

RETURN

END
MACRO (Sheet 1 of 6)

MACRO program using RMS services to open an INDEX file
and terminal INPUT and OUTPUT files.

All I-O is done using RMS services.

User is prompted for SEQ_NO of a record to be
retrieved randomly and given option of deleting
any record retrieved.

Program assumes FILE SHARING and RECORD LOCKING

.TITLE MAROPEN

.MACRO LIB_ERROR ?NO_ERROR
BLBS R0,NO_ERROR
PUSHL R0
CALLS $1,G^LIB$STOP
NO_ERROR:
.ENDM

LIB_ERROR

.RMS_ERROR device, prefix, ?NO_ERROR
BLBS R0,NO_ERROR
MOVAL device, R6
PUSHL prefix$SL_STV(R6)
PUSHL prefix$SL_STS(R6)
PUSHL R0
CALLS $3,G^LIB$STOP
NO_ERROR:
.ENDM

.RMS_ERROR

.PSELECT NONSHARED_DATA NOEXE,WRT

TIME_WAIT: .F_FLOATING 10.0 ; # seconds wait if record locked

IN_REC:

"SEQ_NO: .BLKB 7
"LAST_NAME: .BLKB 15
"FIRST_NAME: .BLKB 10
"SOC_SEC .BLKB 9
"STREET: .BLKB 18
"CITY: .BLKB 14
"STATE: .BLKB 2
"ZIP_CODE: .BLKB 5

INREC_LENGTH = . - IN_REC

INREC_BUFF: .BLKB 80

KEY_VALUE: .BLKB 7

KEY_LENGTH = . - KEY_VALUE
MACRO (Sheet 2 of 6)

IN_PROMPT1: .ASCII /Enter filename: /

IN_PMTSIZE1 = . . IN_PROMPT1

IN_PROMPT2: .ASCII /Enter SEQ_NO: /

IN_PMTSIZE2 = . . IN_PROMPT2

IN_PROMPT3: .ASCII /Do you wish to DELETE this record? (Y or <CR>): /

IN_PMTSIZE3 = . . IN_PROMPT3

DIRECTIONS: .ASCII / AT SEQ_NO PROMPT - HIT <CR> TO STOP RUN/

DIRECTIONS_LENGTH = . . DIRECTIONS

HIGHLIGHT: .ASCII /*********************************************************************************/

HIGHLIGHT_LENGTH = . . HIGHLIGHT

OUT_BUFF: .BLKB 80

OUTBUFF_LENGTH = . . OUT_BUFF

IN_BUFF: .BLKB 80

INBUFF_LENGTH = . . IN_BUFF

MSG_RNF: .ASCII / RECORD NOT FOUND/

MSGRNF_LENGTH = . . MSG_RNF

MSG_RLK: .ASCII /RECORD CURRENTLY LOCKED - WILL TRY AGAIN SHORTLY/

MSGRLK_LENGTH = . . MSG_RLK

.PSECT SHARED_DATA  PIC, NOEXE, LONG

INDX_INFAB:  $FAB

FAC=<GET,DEL>,-  S003881178 2A

FOP=<DFW>,- ; deferred write

ORG=<IDX>,-  S003881178 2A

RAT=<CR>,-

RFM <FIX>,-  8C23308 4420 262880013 8288

SH =<SHRGET,SHRPUT,SHRDEL,SHRPD>

INDX_INRAB:  $RAB

FAB=INDX_INFAB,-

KBF=KEY VALUE,-

KSZ=KEY_LENGTH,-

KRF=0,- ; primary key of reference

MBF=3,- ; multibuffer count

RAC=<KEY>,-

ROP=<FDL>,- ; fast delete

RBF=INREC_BUFF,-

RSZ=INREC_LENGTH,-

UBF=INREC_BUFF,-

USZ=INREC_LENGTH
MACRO (Sheet 3 of 6)

103 TTIN_FAB:
104 $FAB
105 FN=SYS$INPUT,-
106 RAT=CR,-
107 FAC=<GET>
108 TTIN_RAB:
109 $RAB
110 FAB=TTIN_FAB,-
111 UBF=IN_BUFF,-
112 USZ=80,-
113 ROP=PMT,-
114 PBF=IN_PROMPT1,-
115 PSZ=IN_PMTSIZEL
116 TTOUT_FAB:
117 $FAB
118 FN=SYS$OUTPUT,-
119 RAT=CR,-
120 FAC=<PUT>
121 TTOUT_RAB:
122 $RAB
123 FAB=TTOUT_FAB,-
124 RBF=OUT_BUFF,-
125 RSZ=80
126 .PSECT CODE SHR, NOWRT, EXE
127 ;******************************************************************************
128 .ENTRY MAROPEN_INDEX ^M>
129 CALLS ^0, G^LIB$INIT_TIMER
130 LIB_ERROR
131 ; open SYS$INPUT
132 $OPEN
133 FAB=TTIN_FAB
134 RMS_ERROR TTIN_FAB, FAB
135 $CONNECT
136 RAB=TTIN_RAB
137 RMS_ERROR TTIN_RAB, RAB
138 ; input name of indexed file
139 $GET
140 RAB=TTIN_RAB
141 RMS_ERROR TTIN_RAB, RAB
142 ; open indexed file
143 $FAB_STORE
144 FAB=INDX_INFAB,-
145 FMA=IN_BUFF,-
146 FNS=TTIN_RAB+RAB$W_RSZ ; filename
147 ; filename length
148 $OPEN
149 FAB=INDX_INFAB
150 RMS_ERROR INDX_INFAB, FAB
151 $CONNECT
152 RAB=INDX_INRAB
153 RMS_ERROR INDX_INRAB, RAB
154 13-17
MACRO (Sheet 4 of 6)

156 ; open SYS$OUTPUT
157
158 $OPEN
159 RMS_ERROR
160 $CONNECT
161 RMS_ERROR
162
163 ; display directions to stop run and prompt for first key value
164
165 MOVCS
166 $PUT
167 RMS_ERROR
168
169 MOVCS
170 $PUT
171 RMS_ERROR
172
173 MOVCS
174 $PUT
175 RMS_ERROR
176
177 ; input key value
178
179 KEY_IN:
180
181 $RAB_STORE
182 $GET
183
184 STOP_RUN_CHK:
185
186 MOVC5
187 CVTWL
188 SUBL3
189 ADDL
190 MOVC3
191
192 ; right justify key value
193
194 CONV_KEY:
195
196 MOVCS
197 CVTWL
198 SUBL3
199 ADDL
200 MOVC3

13-18
MACRO (Sheet 5 of 6)

; do indexed keyed retrieval of record
201 READ_REC:                   RAB=INDX_INRAB
202    $GET    R0,SUCCESSFUL_READ ; record found
203    BLBS    R0,$RMS$RNF      ; record NOT found
204    CMPL    RNF
205    CMPL    R0,$RMS$RLK     ; record LOCKED
206    BEQL    RLK
207    RMS_ERROR    INDX_INRAB,RAB
208 SUCCESSFUL_READ:            JMP    TYPEOUT_REC
209 ; record not found - print out message and input new key value
210 RNF:
211    MOVCS  #$MSGRNF_LENGTH,MSG_RNF,"/","80",OUT_BUFF
212    $PUT    RAB=TTOUT_RAB
213    RMS_ERROR    TTOUT_RAB,RAB
214    JMP    KEY_IN
215 ; record LOCKED - print out message, wait 10 seconds and try again
216 RLK:
217    MOVCS  #$MSGRLK_LENGTH,MSG_RLK,"/","80",OUT_BUFF
218    $PUT    RAB=TTOUT_RAB
219    RMS_ERROR    TTOUT_RAB,RAB
220    PUSHAL    TIME_WAIT
221    CALLS    #$1,C$LIB$WAIT
222    LIB_ERROR
223 WAIT_OVER:                   JMP    READ_REC
224 ; record found - type out record on terminal
225 TYPEOUT_REC:                JMP    READ_REC
226    MOVCS  #$INREC_LENGTH,INREC_BUFF,OUT_BUFF
227    $PUT    RAB=TTOUT_RAB
228    RMS_ERROR    TTOUT_RAB,RAB
229 ; ask user whether he or she wishes to DELETE record displayed
230 DELETE_FLAG:                JMP    READ_REC
231    $RAB_STORE    RAB=TTIN_RAB,-
232    PBF=IN_PMT3,-
233    PSZ=#IN_PMTSIZE3
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
MACRO (Sheet 6 of 6)

250  SGET
251  TSTW
252  BNEQ
253  JMP
254  CHECK_FLAG_UC:
255  CMPB
256  BNEQ
257  JMP
258  CHECK_FLAG_UC:
259  IN_BUFF, $\sim$A/y/
260  BNEQ
261  JMP
262  CHECK_FLAG_UC:
263  IN_BUFF, $\sim$A/y/
264  BNEQ
265  JMP
266  FLAG_NE:
267  JMP
268  DELETE_REC:
269  $DELETE
270  RMS_ERROR
271  GET_ANOTHER_KEY:
272  DONE:
273  CALLS
274  LIB_ERROR
275  $EXIT_S
276  .END  MAROPEN_INDEX
PASCAL (Sheet 1 of 6)

{PASCOPEN_INDEX.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 PASCOPEN_INDEX(INPUT,OUTPUT,INDX1);

CONST
    TIME_WAIT = 10.0;  
    [ Number of seconds wait
      if record locked ]

TYPE

    ACCOUNT_STRUCT = 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_STRUCT;
UNSAR_FILE = [UNSAFE] FILE OF CHAR;
RAB_PTR = 'RAB$TYPE';

VAR

    INDX1 : ACCOUNT_REC;
    IN_REC : ACCOUNT_STRUCT;
    RMS_STS : UNSIGNED;
    REI_STATUS, 
    LTI_FILENAME, 
    KEYLEN,
    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 ]

13-21
FUNCTION PAS$RAB (VAR anyname : Unsafe_file):
  RAB_PTR; EXTERN;
PROCEDURE LIBSWAIT (num_secs : REAL); EXTERN;
PROCEDURE LIB$STOP (IMMED cond_value : INTEGER); EXTERN;
PROCEDURE LIB$INIT_TIMER (VAR HANDLER_ADR : INTEGER := %IMMED 0); EXTERN;
PROCEDURE LIB$SHOW TIMER (HANDLER_ADR : INTEGER := %IMMED 0;
  CODE : INTEGER := %IMMED 0;
  %IMMED USER_ARG : INTEGER := %IMMED 0); EXTERN;
FUNCTION OPENFILE (VAR FAB : FAB$TYPE;
  VAR RAB : RAB$TYPE;
  VAR FNAME : ACCOUNT_REC); INTEGER;
BEGIN [ openfile function ]
  [ Extension # blocks if file is extended ]
  FAB.FAB$W_DEQ := 10;
  [ File access desired for USER ]
  FAB.FAB$B_FAC := FAB$M_DEL + FAB$M_GET + FAB$M_PUT + FAB$M_UPD;
  [ File options desired ]
  FAB.FAB$L_FOP := FAB$M_DFW; [ deferred write ]
  [ Wipe 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 inserted in source after
  SYSS$OPEN call but prior to SYSS$CONNECT call. ]
  FAB.FAB$W_GBC := #; ]
  [ Sharing attributes - what others can do or set TO FAB$M_SHRNIL ]
  FAB.FAB$B_SHR := FAB$M_SHRPUT + FAB$M_SHRGET + FAB$M_SHRDDEL
  + FAB$M_SHRPUD;
  RET_STATUS := $OPEN (FAB,);
  [ Specify number of local buffers you want RMS to allocate on CONNECT ]
  RAB.RAB$B_MBF := 3;
[ Enable any record processing options to be used for entire run ]

RAB.RABSL.ROP := RAB$M.FDL; [ fast delete ]

IF ODD(RET_STATUS) THEN
  RET_STATUS := SCONNECT(RAB,);
ELSE
  LIB$STOP(RET_STATUS);

OPENFILE := RET_STATUS;

END; [ function openfile ]

BEGIN [ MAIN ]

LIB$INIT_TIMER;

WRITE ('Enter filename: ');
READLN (filename);

OPEN (file_variable := INDX1,
  file_name := filename,
  organization := INDEXED,
  access_method := KEYED,
  history := OLD,
  user_action := OPENFILE);

RAB.START := PASS$RAB (INDX1);
RESETK (INDX1,0); [ Retrieval by primary key ]

WRITELN ('***********************************');
WRITELN ('Enter ZERO to stop run');
WRITELN ('***********************************');

WRITE ('Enter SEQ_NO: ');
READLN (KEY_IN);

WHILE (KEY_IN > 0) DO
  BEGIN [ while loop ]
    Convert integer KEY_IN to right justified character string KEY_VALUE
    IF (PROG_STAT <> 3) THEN
      WRITEV (KEY_VALUE,KEY_IN:7);
    FINDK (INDX1,0,KEY_VALUE,EQL,ERROR:=CONTINUE);

    [ Rest of the code goes here ]
PASCAL (Sheet 4 of 6)

155 | NOTE: PASCAL FINDK does not return status code RMS$_RNF to RAB$T_STS
156 | Can test for this condition with UFB function
157 |
158 | IF (STATUS(INDX1) > 0) OR UFB(INDX1) THEN
159 | BEGIN
160 | PROG_STAT := 2;
161 | RMS_STS := RAB$T_STS;
162 | IF UFB(INDX1) THEN
163 | WRITELN ('Record NOT FOUND with SEQ_NO = ,KEY_VALUE)
164 | ELSE
165 | IF (RMS_STS = RMS$_RLK) THEN
166 | BEGIN
167 | WRITELN ('Record currently LOCKED ',
168 | ' - will try again shortly');
169 | LIB$WAIT (TIME_WAIT);
170 | PROG_STAT := 3;
171 | END
172 | END
173 | ELSE
174 | LIB$STOP (%IMMED RMS_STS);
175 | END
176 | ELSE PROG_STAT := 1;
177 | IF (PROG_STAT = 1) THEN
178 | BEGIN
179 | IN_REC := INDX1;
180 | WRITELN (IN_REC.SEQ_NO, ',IN_REC.SOC_SEC', ',
181 | IN_REC.LAST_NAME);
182 | WRITE ('Do you wish to delete this record?',
183 | '(Y/any char): ');
184 | READLN (DELETE_FLAG);
185 | IF ((DELETE_FLAG = 'Y') OR (DELETE_FLAG = 'y')) THEN
186 | DELETE (INDEX1);
187 | END; [ if prog_stat = 1 ]
188 | IF (PROG_STAT <> 3) THEN
189 | BEGIN
190 | READLN ('Enter SEQ_NO: ');
191 | READLN (KEY_IN);
192 | END; [ if prog_stat <> 3 ]
193 | END; [ while loop ]
194 | CLOSE (INDEX1);
195 | LIB$SHOW_TIMER;
196 | 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. PASS$FAB(f)

The PASS$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.

PASS$FAB is an external function that must be explicitly declared by a declaration such as the following:

```pascal
TYPE
  Unsafe_File = [UNSAFE] FILE OF CHAR;
  Ptr_to_FAB = ^FAB$TYPE;

FUNCTION PASS$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 PASS$FAB. The type FAB$TYPE is defined in the VAX PASCAL environment file STARLET:PEN, which your program or module can inherit.

2. PASS$RAB(f)

The PASS$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.
PASCAL (Sheet 6 of 6)

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 = P(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 interfere 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 FDL$PARSE routine parses an FDL specification, allocates RMS control blocks (FABs, RABs, or XABs), and fills in the relevant fields.

Format

FDL$PARSE fdl_spec, fdl_fab_pointer, fdl_rab_pointer [,flags] [,dfIt_fdl_sp] [,stmt_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_spcl 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.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDL$V_DEFAULT_STRING</td>
<td>Interprets the dflt_fdl_spcl argument as an FDL specification in string form. By default, the dflt_fdl_spcl argument is interpreted as a file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_FDL_STRING</td>
<td>Interprets the fd1_spec argument as an FDL specification in string form. By default, the fd1_spec argument is interpreted as a file name of an FDL file.</td>
</tr>
<tr>
<td>FDL$V_SIGNAL</td>
<td>Signals any error. By default, the status code is returned to the calling image.</td>
</tr>
</tbody>
</table>

This argument is optional. By default, an error status is returned rather than signaled.

dflt_fdl_spcl

The dflt_fdl_spcl 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 fd1_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.

<table>
<thead>
<tr>
<th>Condition Values Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS$ NORMAL</td>
<td>Normal successful completion.</td>
</tr>
<tr>
<td>LIB$ BADBLOADR</td>
<td>Bad block address.</td>
</tr>
<tr>
<td>LIB$ BADBLOSIZ</td>
<td>Bad block size.</td>
</tr>
<tr>
<td>LIB$ INSVIRMEM</td>
<td>Insufficient virtual memory.</td>
</tr>
<tr>
<td>RMS$ DNF</td>
<td>Directory not found.</td>
</tr>
<tr>
<td>RMS$ DNR</td>
<td>Device not ready or mounted.</td>
</tr>
<tr>
<td>RMS$ WCC</td>
<td>Invalid wildcard context (WCC) value.</td>
</tr>
</tbody>
</table>
Typical Sequence of RMS Calls Initiated by FDL$PARSE

FDL$PARSE

YES

NEW FILE?

NO

$CREATE

$OPEN

$CONNECT

READ

READ, WRITE, ETC.

WRITE

$FIND

$GET

$CLOSE

$PUT

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.
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

COBOL

01 x-ptr USAGE IS POINTER
   VALUE REFERENCE variable-name.

FORTRAN

INTEGER*4 x-pointer

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

13-34
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

<table>
<thead>
<tr>
<th>BASIC Statement</th>
<th>RMS Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>$GET</td>
</tr>
<tr>
<td>PUT</td>
<td>$PUT</td>
</tr>
<tr>
<td>FIND</td>
<td>$FIND</td>
</tr>
<tr>
<td>DELETE</td>
<td>$DELETE</td>
</tr>
<tr>
<td>UPDATE</td>
<td>$UPDATE</td>
</tr>
<tr>
<td>RESTORE</td>
<td>$REWIND</td>
</tr>
<tr>
<td>SCRATCH</td>
<td>$TRUNCATE</td>
</tr>
<tr>
<td>FREE</td>
<td>$FREE</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>$RELEASE</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>COBOL Statement</th>
<th>RMS Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>$GET</td>
</tr>
<tr>
<td>CLOSE</td>
<td>$CLOSE, $DISCONNECT, $NXTVOL</td>
</tr>
<tr>
<td>DELETE</td>
<td>$FIND, $DELETE (See Note)</td>
</tr>
<tr>
<td>OPEN</td>
<td>$OPEN or $CREATE, $CONNECT</td>
</tr>
<tr>
<td>READ</td>
<td>$GET</td>
</tr>
<tr>
<td>REWRITE</td>
<td>$FIND, $UPDATE (See Note)</td>
</tr>
<tr>
<td>WRITE</td>
<td>$PUT</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>$PUT</td>
</tr>
<tr>
<td>START</td>
<td>$FIND</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>$RELEASE, $FREE</td>
</tr>
</tbody>
</table>

**NOTE**

$FIND is done only when the DELETE or REWRITE is being performed during random access.
FORTRAN I/O Statements and RMS Routines

<table>
<thead>
<tr>
<th>FORTRAN Statement</th>
<th>RMS Routine</th>
<th>See Note Number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>$GET</td>
<td>5</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>$REWIND, followed by one or more $GET operations if target record is not the first record</td>
<td></td>
</tr>
<tr>
<td>CLOSE</td>
<td>$CLOSE</td>
<td></td>
</tr>
<tr>
<td>DEFINE FILE</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>DELETE(u)</td>
<td>$DELETE</td>
<td>5</td>
</tr>
<tr>
<td>DELETE(u, REC = r)</td>
<td>$FIND, $DELETE</td>
<td>5</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>$PUT</td>
<td>2,5</td>
</tr>
<tr>
<td>FIND</td>
<td>$FIND</td>
<td>1.5</td>
</tr>
<tr>
<td>INQUIRE (by file)</td>
<td>$PARSE, $SEARCH, $OPEN, $CLOSE</td>
<td></td>
</tr>
<tr>
<td>INQUIRE (by unit)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>OPEN</td>
<td>$OPEN or $CREATE, $CONNECT</td>
<td></td>
</tr>
<tr>
<td>OPEN (with USEROPEN)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>OPEN (on connected unit)</td>
<td>$PARSE, $SEARCH, $CLOSE, $OPEN or $CREATE, $CONNECT</td>
<td>8</td>
</tr>
<tr>
<td>PRINT</td>
<td>$PUT</td>
<td>4,5</td>
</tr>
<tr>
<td>READ</td>
<td>$GET</td>
<td>1,5,7</td>
</tr>
<tr>
<td>READ (internal file)</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>REWIND</td>
<td>$REWIND</td>
<td></td>
</tr>
<tr>
<td>REWRITE</td>
<td>$UPDATE</td>
<td>5</td>
</tr>
<tr>
<td>TYPE</td>
<td>$PUT</td>
<td>4,5</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>$FREE</td>
<td>5</td>
</tr>
<tr>
<td>WRITE</td>
<td>$PUT</td>
<td>2,5,7</td>
</tr>
<tr>
<td>WRITE (internal file)</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

13-36
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

<table>
<thead>
<tr>
<th>PASCAL Statement</th>
<th>RMS Routine</th>
<th>See Note Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE</td>
<td>$CLOSE</td>
<td>1, 3</td>
</tr>
<tr>
<td>DELETE</td>
<td>$DELETE</td>
<td>1</td>
</tr>
<tr>
<td>EOF</td>
<td>None</td>
<td>1, 2</td>
</tr>
<tr>
<td>EOLN</td>
<td>None</td>
<td>1, 2</td>
</tr>
<tr>
<td>FIND</td>
<td>$GET</td>
<td>1</td>
</tr>
<tr>
<td>FINDK</td>
<td>$GET</td>
<td>1</td>
</tr>
<tr>
<td>GET</td>
<td>$GET</td>
<td>1, 2</td>
</tr>
<tr>
<td>LINELIMIT</td>
<td>None</td>
<td>1, 2</td>
</tr>
</tbody>
</table>
LOCATE None

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 VPC 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

<table>
<thead>
<tr>
<th>Field</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FABSL_ALQ</td>
<td>Allocation quantity</td>
</tr>
<tr>
<td>FABSB_BKS</td>
<td>Bucket size</td>
</tr>
<tr>
<td>FABSW_BLS</td>
<td>Block size for tape</td>
</tr>
<tr>
<td>FABSL_CTX</td>
<td>Context</td>
</tr>
<tr>
<td>FABSW_DEQ</td>
<td>Default file extension quantity</td>
</tr>
<tr>
<td>FABSL_DEV</td>
<td>Device characteristics</td>
</tr>
<tr>
<td>FABSL_DNA</td>
<td>Default file specification string address</td>
</tr>
<tr>
<td>FABSL_DNS</td>
<td>Default file specification string size</td>
</tr>
<tr>
<td>FABSB_FAC</td>
<td>File access</td>
</tr>
<tr>
<td>FABSL_FNA</td>
<td>File specification string address</td>
</tr>
<tr>
<td>FABSL_FNS</td>
<td>File specification size</td>
</tr>
<tr>
<td>FABSL_FOP</td>
<td>File processing options</td>
</tr>
<tr>
<td>FABSV_CBT</td>
<td>Contiguous best try</td>
</tr>
<tr>
<td>FABSV_CIF</td>
<td>Create if nonexistent</td>
</tr>
<tr>
<td>FABSV_CTX</td>
<td>Contiguous allocation</td>
</tr>
<tr>
<td>FABSV_DFW</td>
<td>Deferred write</td>
</tr>
<tr>
<td>FABSV_DLT</td>
<td>Delete on close service</td>
</tr>
<tr>
<td>FABSV_MKV</td>
<td>Maximize version number</td>
</tr>
<tr>
<td>FABSV_NAM</td>
<td>Name block inputs</td>
</tr>
<tr>
<td>FABSV_NEF</td>
<td>Not position at end of file (tape)</td>
</tr>
<tr>
<td>FABSV_NFS</td>
<td>Not file structured</td>
</tr>
<tr>
<td>FABSV_OPF</td>
<td>Output file parse</td>
</tr>
<tr>
<td>FABSV_POS</td>
<td>Current position (after closed file) (tape)</td>
</tr>
<tr>
<td>FABSV_RCK</td>
<td>Read check</td>
</tr>
<tr>
<td>FABSV_RKC</td>
<td>Rewind on close service (tape)</td>
</tr>
<tr>
<td>FABSV_RMC</td>
<td>Rewind on open service (tape)</td>
</tr>
<tr>
<td>FABSV_RWO</td>
<td>Rewind on close service (tape)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAB$V_SCF</td>
<td>Submit command</td>
</tr>
<tr>
<td>FAB$V_SPL</td>
<td>Spool to printer</td>
</tr>
<tr>
<td>FAB$V_SQQ</td>
<td>Sequential only</td>
</tr>
<tr>
<td>FAB$V_SUP</td>
<td>Supersede</td>
</tr>
<tr>
<td>FAB$V_TEF</td>
<td>Truncate at end-of-file</td>
</tr>
<tr>
<td>FAB$V_TMD</td>
<td>Temporary, marked for delete</td>
</tr>
<tr>
<td>FAB$V_TMP</td>
<td>Temporary (file with no directory entry)</td>
</tr>
<tr>
<td>FAB$V_UFO</td>
<td>User file open or create file only</td>
</tr>
<tr>
<td>FAB$V_WCK</td>
<td>Write check</td>
</tr>
<tr>
<td>FABSB_FZ5</td>
<td>Fixed control area size</td>
</tr>
<tr>
<td>FABSW_GBC</td>
<td>Global buffer count</td>
</tr>
<tr>
<td>FABSW_IFI</td>
<td>Internal file identifier</td>
</tr>
<tr>
<td>FABSL_MRN</td>
<td>Maximum record number</td>
</tr>
<tr>
<td>FABSW_MRS</td>
<td>Maximum record size</td>
</tr>
<tr>
<td>FABSL_NAM</td>
<td>Name block address</td>
</tr>
<tr>
<td>FABSB_ORG</td>
<td>File organization</td>
</tr>
<tr>
<td>FABSB_RAT</td>
<td>Record attributes</td>
</tr>
<tr>
<td>FAB$V_FTN</td>
<td>FORTRAN carriage control</td>
</tr>
<tr>
<td>FAB$V_CR</td>
<td>Print LF and CR</td>
</tr>
<tr>
<td>FAB$V_BLK</td>
<td>Do not cross block boundaries</td>
</tr>
<tr>
<td>FABSB_RFN</td>
<td>Record format</td>
</tr>
<tr>
<td>FABSB_RTV</td>
<td>Retrieval window size</td>
</tr>
<tr>
<td>FABSL_SDC</td>
<td>Spooling device</td>
</tr>
<tr>
<td>FABSB_SHR</td>
<td>Characteristics</td>
</tr>
<tr>
<td>FABSL_STS</td>
<td>File sharing</td>
</tr>
<tr>
<td>FABSL_STV</td>
<td>Completion status code</td>
</tr>
<tr>
<td>FAB$L_XAB</td>
<td>Status value (if any)</td>
</tr>
<tr>
<td></td>
<td>Extended attribute block address</td>
</tr>
</tbody>
</table>
### Summary of RAB Fields and Options

<table>
<thead>
<tr>
<th>Field</th>
<th>Name</th>
<th>Field</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAB$L_BKT</td>
<td>Bucket code</td>
<td>RAB$V_KGE</td>
<td>Key greater than or equal to</td>
</tr>
<tr>
<td>RAB$L_CTX</td>
<td>Context</td>
<td>RAB$V_KGT</td>
<td>Key greater than</td>
</tr>
<tr>
<td>RAB$L_FAB</td>
<td>FAB address</td>
<td>RAB$V_LIM</td>
<td>Limit</td>
</tr>
<tr>
<td>RAB$W_ISI</td>
<td>Internal stream ID</td>
<td>RAB$V_LOA</td>
<td>Load according to fill factor</td>
</tr>
<tr>
<td>RAB$L_KBF</td>
<td>Key buffer address</td>
<td>RAB$V_LOC</td>
<td>Locate mode</td>
</tr>
<tr>
<td>RAB$B_KRF</td>
<td>Key of reference</td>
<td>RAB$V_NLR</td>
<td>No lock</td>
</tr>
<tr>
<td>RAB$B_KZ2</td>
<td>Key size</td>
<td>RAB$V_NMX</td>
<td>Nonexistent record</td>
</tr>
<tr>
<td>RAB$B_MBC</td>
<td>Multiblock count</td>
<td>RAB$V_PMT</td>
<td>Prompt</td>
</tr>
<tr>
<td>RAB$B_MBF</td>
<td>Multibuffer count</td>
<td>RAB$V_PTA</td>
<td>Purge type-ahead</td>
</tr>
<tr>
<td>RAB$L_PBF</td>
<td>Prompt buffer address</td>
<td>RAB$V_RAH</td>
<td>Read-ahead</td>
</tr>
<tr>
<td>RAB$L_PSZ</td>
<td>Prompt buffer size</td>
<td>RAB$V_REA</td>
<td>Lock record for read</td>
</tr>
<tr>
<td>RAB$L_RAC</td>
<td>Record access mode</td>
<td>RAB$V_RLK</td>
<td>(others can read)</td>
</tr>
<tr>
<td>RAB$C_SEQ</td>
<td>Record address</td>
<td>RAB$V_RNE</td>
<td>Lock record for write</td>
</tr>
<tr>
<td>RAB$C_KEY</td>
<td>Record's file address</td>
<td>RAB$V_RRL</td>
<td>(others can read)</td>
</tr>
<tr>
<td>RAB$C_RFA</td>
<td>VBN #</td>
<td>RAB$V_RNN</td>
<td>Read no echo</td>
</tr>
<tr>
<td>RAB$L_RBZ</td>
<td>Byte offset or ID #</td>
<td>RAB$V_RNF</td>
<td>Read no regardless of lock</td>
</tr>
<tr>
<td>RAB$L_RBP</td>
<td>Record header buffer</td>
<td>RAB$V_TMO</td>
<td>Timeout</td>
</tr>
<tr>
<td>RAB$W_RFA</td>
<td>Record processing options</td>
<td>RAB$V_TTP</td>
<td>Truncate put</td>
</tr>
<tr>
<td>RAB$L_RFAO</td>
<td>Asynchronous</td>
<td>RAB$V_UIE</td>
<td>Update if</td>
</tr>
<tr>
<td>RAB$W_RFA4</td>
<td>Block I/O</td>
<td>RAB$V_ULK</td>
<td>Manual unlocking</td>
</tr>
<tr>
<td>RAB$L_RHB</td>
<td>Cancel CTRL/O</td>
<td>RAB$V_WAU</td>
<td>Wait if record currently locked</td>
</tr>
<tr>
<td>RAB$L_RBP</td>
<td>Convert to uppercase</td>
<td>RAB$V_WBL</td>
<td>Write-behind</td>
</tr>
<tr>
<td>RAB$L_ROP</td>
<td>End-of-file</td>
<td>RAB$W_RZ2</td>
<td>Record size</td>
</tr>
<tr>
<td>RAB$L_RSF</td>
<td>Extended terminal operation</td>
<td>RAB$L_STS</td>
<td>Completion status code</td>
</tr>
<tr>
<td>RAB$V_FDL</td>
<td>Fast delete</td>
<td>RAB$L_STV</td>
<td>Status value (if any)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAB$B_TMO</td>
<td>Timeout period</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAB$L_UBF</td>
<td>User record area address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAB$W_USZ</td>
<td>User record area size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RAB$L_XAZ</td>
<td>Next XAB address</td>
</tr>
</tbody>
</table>

13-40
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

<table>
<thead>
<tr>
<th>Record Operation</th>
<th>Record Access Mode</th>
<th>Current Record</th>
<th>Next Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>Does not apply</td>
<td>None</td>
<td>First record</td>
</tr>
<tr>
<td>RAB$S Romney</td>
<td>None</td>
<td></td>
<td>End of file</td>
</tr>
<tr>
<td>RAB$S_EOF set</td>
<td>Does not apply</td>
<td>Old next record</td>
<td>New current</td>
</tr>
<tr>
<td>Get, when last service was not a Find</td>
<td>Sequential</td>
<td>New current record+1</td>
<td></td>
</tr>
<tr>
<td>Get, when last service was a Find</td>
<td>Sequential</td>
<td>Unchanged</td>
<td>Current record+1</td>
</tr>
<tr>
<td>Get</td>
<td>Random</td>
<td>New</td>
<td>New current record+1</td>
</tr>
<tr>
<td>Put, sequential file</td>
<td>Sequential</td>
<td>None</td>
<td>End of file</td>
</tr>
<tr>
<td>Put, relative file</td>
<td>Sequential</td>
<td>None</td>
<td>Next record position</td>
</tr>
<tr>
<td>Put, indexed file</td>
<td>Sequential</td>
<td>None</td>
<td>Undefined</td>
</tr>
<tr>
<td>Put</td>
<td>Random</td>
<td>None</td>
<td>Changed</td>
</tr>
<tr>
<td>Find</td>
<td>Sequential</td>
<td>Old next record</td>
<td>New current</td>
</tr>
<tr>
<td>Find</td>
<td>Random</td>
<td>New</td>
<td>New current record+1</td>
</tr>
<tr>
<td>Update</td>
<td>Does not apply</td>
<td>None</td>
<td>Changed</td>
</tr>
<tr>
<td>Delete</td>
<td>Does not apply</td>
<td>None</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>

13-41
Record Access Stream Context (Cont.)

Record Operation

Record Access Mode Current 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__POP is set).

13-42
Example 2. FDL$PARSE Alternative

BASIC (Sheet 1 of 7)

1 10 OPTION TYPE = EXPLICIT

2 !

3 ! NOTE: $LINK BASPARSE_INDEX,BAS_SETRMS

4 ! BAS_SETRMS.BAS contains two external routines:

5 !

6 SETPB

7 SETAB

8 !

9 ! This BASIC program uses FDL$PARSE to set up RMS structures
10 ! for accessing an indexed file rather than using a USEROPEN
11 ! function. In addition, all access to the INDEXED file is done
12 ! using RMS services rather than regular BASIC I/O. Using the
13 ! program is prompted for key value for random retrieval of
14 ! records and given option to delete any record retrieved.
15 !
16 ! Program assumes FILE SHARING and RECORD LOCKING
17 !
18 ! ON ERROR GO TO err_check
19 !
20 ! DECLARE SINGLE CONSTANT TIME_WAIT=10.0
21 !
22 ! DECLARE STRING CONSTANT RIGHT_JUSTIFY = ""###"
23 !
24 ! DECLARE WORD CONSTANT INREC_LENGTH=80
25 !
26 ! DECLARE WORD CONSTANT KEY_VALUE_LENGTH=7
27 !
28 ! ACCOUNT$TYPE
29 !
30 ! MAP (IN_REC) ACCOUNT$TYPE INREC
31 !
32 ! MAP (FILE_NAME) STRING FDL_FILE=80
33 !
34 ! MAP (KEYVALUE) STRING KEY_VALUE=7
35 !
36 ! MAP (DELETER) STRING DELETE_FLAG=1
37 !
38 ! DECLARE LONG FAB_PTR, &
39 !
40 ! DECLARE LONG RAB_PTR, &
41 !
42 ! DECLARE LONG RET_STATUS, &
43 !
44 ! DECLARE LONG INREC_PTR, &
45 !
46 ! DECLARE LONG KEY_IN, &
47 !
48 ! DECLARE LONG KEY_VALUE_PTR
49 !
50 ! DECLARE LONG PROG_STAT
51 !
52 ! VALUE 1 = no error in GET
53 !
54 ! VALUE 2 = some error
55 !
56 ! VALUE 3 = wait-read-again

13-43
EXTERNAL SUB
LIB$INIT_TIMER, &
LIB$SHOW_TIMER, &
LIB$WAIT
EXTERNAL LONG FUNCTION FDL$PARSE, &
SYS$OPEN, &
SYS$CONNECT, &
SYS$GET, &
SYS$DELETE, &
SYS$CLOSE
EXTERNAL LONG CONSTANT RMS$RFN, &
RMS$RLK

! start!
CALL LIB$INIT_TIMER()
! INPUT 'Enter name of FDL file '; FDL_FILE
! RET_STATUS = FDL$PARSE(FDL_FILE,
! FAB_PTR, &
! RAB_PTR, &)
! CALL LIB$STOP(RET_STATUS BY VALUE) IF (RET_STATUS AND 16)=0%

CALL SETFAB(FAB_PTR BY VALUE)

RET_STATUS = SYS$OPEN(FAB_PTR BY VALUE, &)
CALL LIB$STOP(RET_STATUS BY VALUE) IF (RET_STATUS AND 16)=0%

INREC_PTR = LOC(INREC)  ! internal ADDR input rec buff
KEY_VALUE_PTR = LOC(KEY_VALUE)  ! internal ADDR key value
CALL SETRAB(
RAB_PTR BY VALUE, &
INREC_PTR, &
INREC_LENGTH, &
KEY_VALUE_PTR, &
KEY_VALUE_LENGTH, &)

RET_STATUS = SYS$CONNECT(RAB_PTR BY VALUE, &)
CALL LIB$STOP(RET_STATUS BY VALUE) IF (RET_STATUS AND 16)=0%

PRINT "*******************************************************************"
PRINT " Hit <CR> or enter zero to stop run"
PRINT "*******************************************************************"
INPUT 'Enter SEQ_NO'; KEY_IN
PROG_STAT = 0
BASIC (Sheet 3 of 7)

106 WHILE (KEY_IN > 0)
107 108 IF (PROG_STAT <> 3) THEN
109     KEY_VALUE = FORMATS(KEY_IN,RIGHT_JUSTIFY)  ! Convert key-in
110     ! to right justified character string key_value
111     END IF
112
113 RET_STATUS = SYS$GET(RAB_PTR BY VALUE, ,)
114
115 IF (RET_STATUS AND 18) = 0% THEN
116     PROG_STAT = 2
117     SELECT RET_STATUS
118         CASE RMS$ RNF
119             PRINT "Record not found with SEQ_NO = "; KEY_VALUE
120         CASE RMS$ RLK
121             PRINT "Record currently locked - will try again"
122             CALL LIB$WAIT(TIME_WAIT)
123     END SELECT
124     CALL LIB$STOP(RET_STATUS BY VALUE)
125 
126 ELSE
127     PROG_STAT = 1
128     END IF
129
130 IF (PROG_STAT = 1) THEN
131     PRINT INREC$;SEQ_NO;TAB(9);INREC$;SOC_SEC;TAB(20); &
132     INPUT 'Do you wish to delete this record? (Y/<CR>)'; &
133     DELETE_FLAG
134     IF (DELETE_FLAG = 'Y' OR DELETE_FLAG = 'y') THEN
135         RET_STATUS = SYS$DELETE(RAB_PTR BY VALUE, ,)
136     END IF
137 
138 INPUT 'Enter SEQ_NO',KEY_IN IF PROG_STAT <> 3
139 
140 NEXT cleanup:
141 CALL SYS$CLOSE(FAB_PTR BY VALUE, ,)
142 CALL LIB$SHOW_TIMER()
143 GOTO done
144 err_check:
145 ON ERROR GO TO 0
146 done: END
BASIC (Sheet 4 of 7)

200  SUB SETFAB (FAB$TYPE FAB)

EXTERNAL BASIC routines called by COBPARSE_INDEX.COB or
by BASPARSE_INDEX.BAS

OPTION TYPE = EXPLICIT

%INCLUDE 'FABRABDEF.BAS'

Extension # blocks if file is extended
FAB$::FAB$W_deck = 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
! Specify key of reference
RAB::RAB$B_KRF = 0 ! primary key

! Specify number of local buffers you want RMS to allocate on CONNECT
RAB::RAB$B_MBF = 3

! Specify type of mode to use for record access
RAB::RAB$B_RAC = RAB$C_KEY ! RAB$C_SEQ or RAB$C_RFA

! Provide address and length of output record buffer
RAB::RAB$L_RBF = REC_PTR
RAB::RAB$W_RSZ = REC_LENGTH

! Provide address and length of input record buffer
RAB::RAB$L_UBF = REC_PTR
RAB::RAB$W_USZ = REC_LENGTH

! Enable any record processing options to be used for entire run
RAB::RAB$L_ROP = (RAB::RAB$L_ROP OR RAB$M_FDL) ! fast delete

END SUB
RECORD fab$TYPE

BYTE fab$b_bid
BYTE fab$b_bln
WORD fab$w_ifi
LONG fab$l_top
LONG fab$l_sts
LONG fab$l_stv
LONG fab$l_alq
WORD fab$w_deq
BYTE fab$b_fac
BYTE fab$b_shr
LONG fab$l_ctx
BYTE fab$b_rtv
BYTE fab$b_org
BYTE fab$b_rat
LONG fab$l_jnl
LONG fab$l_xab
LONG fab$l_nam
LONG fab$l_fna
LONG fab$l_dna
BYTE fab$b_fns
BYTE fab$b_dns
WORD fab$w_mrs
LONG fab$l_mrn
WORD fab$w_bls
BYTE fab$b_bks
BYTE fab$b_fsz
LONG fab$l_dev
LONG fab$l_sdc
WORD fab$w_gbc
BYTE fab$b_acm
BYTE fab$b_rcf
BYTE fill(4)

END RECORD

RECORD rab$TYPE

BYTE rab$b_bid
BYTE rab$b_bln
WORD rab$b_isi
LONG rab$l_rop
LONG rab$1_sts
LONG rab$1_stv
RFA rab$w_rfa
16 bytes
WORD fill
LONG rab$1_ctx
WORD fill
BYTE rab$1_rac
BYTE rab$1_tmo
WORD rab$w_usz
WORD rab$w_rsz
LONG rab$1_ubf
LONG rab$1_rbf
LONG rab$1_rhb
VARIANT
CASE
  LONG rab$1_kbf
  BYTE rab$1_ksz
CASE
  LONG rab$1_pbf
  BYTE rab$1_psz
END VARIANT
BYTE rab$1_krf
BYTE rab$1_mbf
BYTE rab$1_mbc
VARIANT
CASE
  LONG rab$1_bkt
CASE
  LONG rab$1_dct
END VARIANT
LONG rab$1_fab
LONG rab$1_xab
END RECORD

! +
declarations of FAB and RAB CONSTANTS
! +
EXTERNAL BYTE CONSTANT FAB$M_DEL,&
FAB$M_GET,&
FAB$M_PUT,&
FAB$M_UPD,&
FAB$M_DFW,&
FAB$M_SHRPUT,&
FAB$M_SHRGET,&
FAB$M_SHRDEL,&
FAB$M_SHRUDF,&
FAB$M_MSE
EXTERNAL BYTE CONSTANT RAB$C_KEY,&
RAB$C_SEQ,&
RAB$C_RFA,&
RAB$M_FDL
COBOL 1 (Sheet 1 of 10)

*  *  *  COBOL PROGRAM which sets up RMS structures using FDL$PARSE
*  *  and calls TWO external non-COBOL routines to SET values in
*  *  FAB & RAB. The program retrieves records randomly by key value
*  *  entered by USER and gives user option to DELETE record.
*  *  *
*  *  This version is appropriate for linking in the following SETRMS
*  *  external routines:
*  *  *
11  $ LINK COBPARSE_INDEX,BAS_SETRMS
12  $ LINK COBPARSE_INDEX,FOR_SETRMS
13  $ LINK COBPARSE_INDEX,PAS_SETRMS
14  *
15  Program assumes FILE SHARING and RECORD LOCKING
16  *
17  IDENTIFICATION DIVISION.
18  *
19  PROGRAM-ID. COBPARSE_INDEX.
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  *
37  CONSTANTS.
38    01 INREC_LENGTH PIC 9(5) COMP VALUE IS 80.
39    01 KEY_VALUE_LENGTH PIC 9(5) COMP VALUE IS 7.
40    01 TIME_WAIT   USAGE COMP-1 VALUE IS 10.0.
41  *
42  VARIABLES.
43    01 FAB_PTR     USAGE IS POINTER.
44    01 RAB_PTR     USAGE IS POINTER.
45    01 INREC_PTR   POINTER VALUE REFERENCE IN_REC.
46    01 KEY_VALUE   PIC X(7) JUSTIFIED RIGHT.
47    01 KEY_VALUE_PTR POINTER VALUE REFERENCE KEY_VALUE.
48    01 FDL_FILE    PIC X(80).
49    01 RET_STATUS  PIC S9(9) COMP.
50    01 DELETE_FLAG PIC X VALUE IS "N".
51    01 PROG_STAT   PIC 9.
52    88 NO_ERROR   VALUE 1.
53    88 SOME_ERROR VALUE 2.
88 WAIT_READ_AGAIN VALUE 3.
88 END_OF_INPUT VALUE 4.

01 RMS$_RNF PIC 9(9) COMP VALUE EXTERNAL RMSS_RNF.
01 RMS$_RLK PIC 9(9) COMP VALUE EXTERNAL RMSS_RLK.

PROCEDURE DIVISION.
000-BEGIN.

CALL 'LIB$INIT_TIMER'.
DISPLAY 'Enter FDL file name: ' WITH NO ADVANCING.
ACCEPT FDL_FILE.

CALL FDL$PARSE to set up RMS structures (FAB, RAB)
CALL 'FDL$PARSE' USING BY DESCRIPTOR FDL_FILE
     BY REFERENCE FAB_PTR RAB_PTR
     BY VALUE 0 0 0
     GIVING RET_STATUS.

IF RET_STATUS IS FAILURE CALL 'LIB$STOP'.

CALL external routine SETFAB in a non-COBOL language
that supports FAB structure to set any fields
needed which were not set in FDL file
CALL 'SETFAB' USING BY VALUE FAB_PTR.

Open input data file
CALL 'SYSS$OPEN' USING BY VALUE FAB_PTR 0 0
     GIVING RET_STATUS.
IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
     USING BY VALUE RET_STATUS.

CALL external routine SETRAB WRITTEN IN BASIC,
FORTRAN or PASCAL or some higher-level language
that supports RAB structure and at a minimum
initialize addresses of record and keyvalue buffers
and their sizes

NOTE: If the external routine were written in MACRO
all the PTR arguments HAVE TO BE PASSED BY VALUE
and only the lengths BY REFERENCE.

CALL 'SETRAB' USING BY VALUE RAB_PTR
     BY REFERENCE INREC_PTR
     BY REFERENCE INREC_LENGTH
     BY REFERENCE KEY_VALUE_PTR
     BY REFERENCE KEY_VALUE_LENGTH.
* Connect record
   CALL 'SYSSCONNECT' USING BY VALUE RAB_PTR 0 0
   GIVING RET_STATUS.
   IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
   USING BY VALUE RET_STATUS.
   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.
   *
   * Prompt user for key value
   IF PROG_STAT NOT EQUAL TO 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 NOT EQUAL TO 4 THEN
      PERFORM 150-READ-BY-PRIMARY-KEY.

150-READ-BY-PRIMARY-KEY.
   CALL 'SYSGET' USING BY VALUE RAB_PTR 0 0
   GIVING RET_STATUS.
   IF RET_STATUS IS FAILURE THEN
      MOVE 2 TO PROG_STAT
      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 RET_STATUS
      END-EVALUATE
   ELSE
      MOVE 1 TO PROG_STAT
   END-IF.
COBOL 1 (Sheet 4 of 10)

154 IF PROG_STAT IS EQUAL TO 1 THEN
155 DISPLAY SEQ_NO," ",SOC_SEC," ",LAST_NAME
156 PERFORM 200-DELETE-RECORD
157 END-IF.
158
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.
SUB SETFAB (FAB$TYPE FAB)  

EXTERNAL BASIC routines called by COBPARSE_INDEX.COB or  
by BASPARSE_INDEX.BAS  

OPTION TYPE = EXPPLICIT  

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$L_POP = (FAB::FAB$L_POP 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 open with global buffers enabled  
The following statement must be moved to SETTRAB 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_SHRFUT &  
OR FAB$M_SHRGET OR FAB$M_SHRDEL &  
OR FAB$M_SHRUPD)  

END SUB  

SUB SETTRAB (RAB$TYPE RAB,&  
LONG REC_PTR, &  
WORD REC_LENGTH,&  
LONG KEY_PTR,&  
WORD KEY_LENGTH)  

OPTION TYPE = EXPPLICIT  

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
COBOL 1 (Sheet 6 of 10)

BASIC External Routines (2 of 2)

52 ! Specify key of reference
53 RAB::RAB$B_KRF = 0  ! primary key
54
55 ! Specify number of local buffers you want RMS to allocate on CONNECT
56 RAB::RAB$B_MBF = 3
57
58 ! Specify type of mode to use for record access
59 RAB::RAB$B_RAC = RAB$C_KEY    ! RAB$C_SEQ or RAB$C_RFA
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_PDL)  ! fast delete
71
72 END SUB
EXTERNAL FORTRAN routines called by COBPARSER_INDX.COB or
by FORPARSER_INDX.FOR

SUBROUTINE SETFAB (FAB)
IMPLICIT NONE

INCLUDE '(FABDEF)'
RECORD /FABDEF/ FAB

! 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
1 .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
1 .OR. FAB$M_SHRGET .OR. FAB$M_SHRDEL
2 .OR. FAB$M_SHRUPD

RETURN
END

SUBROUTINE SETRAB (RAB,REC_PTR,REC_LENGTH,KEY_PTR,KEY_LENGTH)
IMPLICIT NONE

INTEGER*4 REC_PTR,KEY_PTR
INTEGER*2 REC_LENGTH,KEY_LENGTH
INCLUDE '(RABDEF)'
RECORD /RABDEF/ RAB

! Provide address of key_value if keyed retrieval to be used
! and size of key
RAB.RABSL_KBF = KEY_PTR
RAB.RABSB_KS2 = KEY_LENGTH

! Specify key of reference
RAB.RABSB_KRF = 0 ! primary key
! Specify number of local buffers you want RMS to allocate on CONNECT
RAB.RAB$M_BF = 3

! Specify type of mode to use for record access
RAB.RAB$M_RAC = RABC_KEY ! RABC_SEQ or RABC_RFA

! Provide address and length of output record buffer
RAB.RAB$L_RBF = REC_PTR
RAB.RAB$W_RSZ = REC_LENGTH

! Provide address and length of input record buffer
RAB.RAB$L_UBF = REC_PTR
RAB.RAB$W_USZ = REC_LENGTH

! Enable any record processing options to be used for entire run
RAB.RAB$L_ROP = RAB.RAB$L_ROP .OR. RAB$M_PDL ! fast delete

RETURN
END
EXTERNAL PASCAL routines called by COBPARSE_INDEX.COB and by PASPARSE_INDEX.PAS

INHERIT ('SYS$LIBRARY: STARLET') MODULE RMS_SETTINGS;

TYPE
  WORD_INTEGER = [WORD] 0..64534;

[GLOBAL] PROCEDURE SETFAB (VAR FAB : FAB$TYPE);

BEGIN
  { Extension # blocks if file is extended }
  FAB.FAB$W_DEQ := 10;
  { File access desired for USER }
  FAB.FAB$B_FAC := FAB$M_DEL + FAB$M_GET + FAB$M_PUT + FAB$M_UPD;
  { File options desired }
  FAB.FAB$L_POP := FAB$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 SYSSCONNECT call. }
  { FAB.FAB$W_GBC := #; }
  { Sharing attributes - what others can do or set TO FAB$M_SHRNIL }
  FAB.FAB$B_SHR := FAB$M_SHRPUT + FAB$M_SHRGET + FAB$M_SHRDDEL
    + FAB$M_SHRUPD;

END;  [{ SETFAB ]

[GLOBAL] PROCEDURE SETRAB (VAR RAB : RAB$TYPE;
  REC_PTR : INTEGER;
  REC_LENGTH : WORD INTEGER;
  KEY_PTR : INTEGER;
  KEY_LENGTH : WORD INTEGER);

BEGIN
  { Provide address of key_value if keyed_retrieval to be used and size of key }
  RAB.RAB$SL_KBF := KEY_PTR;
  RAB.RAB$B_KSZ := KEY_LENGTH;
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 PARSE_INDEX.COB

COBOL PROGRAM which sets up RMS structures using FDLPARSE
and calls TWO external non-COBOL routines to SET values in FAB & RAB. The program retrieves records randomly by key value entered by USER and gives user option to DELETE record.

This version is appropriate for linking in the external routines written in MACRO:

$ LINK COBP parse_INDEX,COBMAR_SETRMS

See COBPARSE_INDEX.COB for BASIC, FORTRAN and PASCAL versions.

IDENTIFICATION DIVISION.
PROGRAM-ID. COBPARSE_INDEX.

ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 INREC.
   02 SEQ_NO PIC X(7).
   02 LAST_NAME PIC X(15).
   02 FIRST_NAME PIC X(10).
   02 SOC_SEC PIC X(9).
   02 STREET PIC X(18).
   02 CITY PIC X(14).
   02 STATE PIC XX.
   02 ZIP_CODE PIC X(5).

* CONSTANTS
   01 INREC_LENGTH PIC 9(5) COMP VALUE IS 80.
   01 KEY_VALUE_LENGTH PIC 9(5) COMP VALUE IS 7.
   01 TIME_WAIT USAGE COMP-1 VALUE IS 10.0.

* VARIABLES
   01 FAB_PTR USAGE IS POINTER.
   01 RAB_PTR USAGE IS POINTER.
   01 INREC_PTR POINTER VALUE REFERENCE INREC.
   01 KEY_VALUE PIC X(7) JUSTIFIED RIGHT.
   01 KEY_VALUE_PTR POINTER VALUE REFERENCE KEY_VALUE.
   01 FDL_FILE PIC X(80).
   01 RET_STATUS PIC S9(9) COMP.
   01 DELETE_FLAG PIC X VALUE IS "N".
   01 PROG_STAT PIC 9.
   88 NO_ERROR VALUE 1.
   88 SOME_ERROR VALUE 2.

13-60
CALL 'LIB$INIT_TIMER'.

DISPLAY 'Enter FDL file name: ' WITH NO ADVANCING.
ACCEPT FDL_FILE.

Call FDL$PARSE to set up RMS structures (FAB, RAB)
CALL 'FDL$PARSE' USING BY DESCRIPTOR FDL_FILE
    BY REFERENCE FAB_PTR RAB_PTR
    BY VALUE 0 0 0
    GIVING RET_STATUS.
IF RET_STATUS IS FAILURE CALL 'LIB$STOP'.

* call external routine SETFAB in a non-COBOL language
* that supports FAB structure to set any fields
* needed which were not set in FDL file
CALL 'SETFAB' USING BY VALUE FAB_PTR.

* Open input data file
CALL 'SYSSOPEN' USING BY VALUE FAB_PTR 0 0
    GIVING RET_STATUS.
IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
    USING BY VALUE RET_STATUS.

* call external routine SETRAB WRITTEN IN MACRO and at
* a minimum initialize addresses of record and keyvalue
* buffers and their sizes

* NOTE: If the external routine written in BASIC,
* FORTRAN or PASCAL, ONLY the RAB_PTR would be passed
* BY VALUE and all other arguments BY REFERENCE.

CALL 'SETRAB' USING BY VALUE RAB_PTR
    BY VALUE INREC_PTR
    BY REFERENCE INREC_LENGTH
    BY VALUE KEY_VALUE_PTR
    BY REFERENCE KEY_VALUE_LENGTH.

* Connect record
CALL 'SYSSCONNECT' USING BY VALUE RAB_PTR 0 0
    GIVING RET_STATUS.
IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
    USING BY VALUE RET_STATUS.
DISPLAY "**************************************************************************".
DISPLAY " HIT CNTL+2 TO STOP RUN"
DISPLAY "**************************************************************************".
MOVE 0 TO PROG_STAT.
PERFORM 100-CHOOSE-RECORD UNTIL END_OF_INPUT.
PERFORM 250-CLEANUP.

100-CHOOSE-RECORD.
* Prompt user for key value
IF PROG_STAT NOT EQUAL TO 3 THEN
  DISPLAY "Enter SEQ NO: " WITH NO ADVANCING
  ACCEPT KEY_VALUE WITH CONVERSION AT END
  MOVE 4 TO PROG_STAT
ENDIF.
IF PROG_STAT NOT EQUAL TO 4 THEN
  PERFORM 150-READ-BY-PRIMARY-KEY.

150-READ-BY-PRIMARY-KEY.
CALL 'SYS$GET' USING BY VALUE RAB_PTR 0 0 GIVING RET_STATUS.
IF RET_STATUS IS FAILURE THEN
  MOVE 2 TO PROG_STAT
  EVALUATE RET_STATUS
  WHEN RMSS_RNF
    DISPLAY "NO RECORD WITH SEQ_NO = ",KEY_VALUE
  WHEN RMSS_RLK
    DISPLAY "RECORD CURRENTLY LOCKED - ",KEY_VALUE
    "WILL TRY AGAIN SHORTLY"
    CALL 'LIB$WAIT' USING BY REFERENCE TIME_WAIT
    MOVE 3 TO PROG_STAT
  WHEN OTHER
    CALL 'LIB$STOP' USING BY VALUE RET_STATUS
  END-EVALUATE
ELSE
  MOVE 1 TO PROG_STAT
END-IF.
IF PROG_STAT IS EQUAL TO 1 THEN
  DISPLAY SEQ_NO," ",SEQ_SEC," ",LAST_NAME
  CALL 'LIB$DELETE' USING BY VALUE SEQ_NO RET_SUCCESS
  IF RET_SUCCESS IS SUCCESS THEN
    MOVE 5 TO PROG_STAT
  ELSE
    MOVE 2 TO PROG_STAT
  END-IF.
END-IF.
200-DELETE-RECORD.
   DISPLAY "Do you wish to delete this record? (Y/<CR>): "
   WITH NO ADVANCING.
   ACCEPT DELETE_FLAG.
   IF DELETE_FLAG = "Y" OR DELETE_FLAG = "y"
      CALL 'SYSS$DELETE' USING BY VALUE RAB_PTR 0 0
      GIVING RET_STATUS
      IF RET_STATUS IS FAILURE CALL 'LIB$STOP'
      USING BY VALUE RET_STATUS
   END-IF.
250-CLEANUP.
   CALL 'SYSS$CLOSE' USING BY VALUE FAB_PTR 0 0
   GIVING RET_STATUS.
   CALL 'LIB$SHOW_TIMER'.
   STOP RUN.
MACRO External Routines (1 of 1)

ENTRY SETFAB,"M>

FAB=0L'4(AP),- ; RAB address
DEQ=#10,- ; extension # blocks if file extended
FAC=DEL,GET,PUT,UPD>,- ; what you can do
FOP=DFW,- ; file options - deferred write
SHR=PUT,GET,DEL,UPD>,- ; what others can do
RET

ENTRY SETRAB,"M>

RAB=0L'4(AP),- ; RAB address
RBF=0L'8(AP),- ; output record buffer
RSZ=0W'12(AP),- ; input record buffer
UBF=0L'8(AP),- ; input record buffer
USZ=0W'12(AP),- ; input KEY buffer
KBF=0L'16(AP),- ; input KEY buffer
KSZ=0W'20(AP),- ; primary key
KRF=$0,- ; number of local buffers
MBF=$3,- ; type of record access SEQ or RFA
RAC=KEY,- ; record options for whole run - fast delete
ROP=FDL

RET

.END
FORTRAN (Sheet 1 of 6)

NOTE: $LINK FORPARSE_INDEX,FOR_SETRMS

FOR_SETRMS.FOR contains two external subroutines:

SETFAB
SETRAB

FORTRAN program using FDLPARSE to set up RMS structures
for accessing an INDEXED file rather than using
USEROPEN function. In addition, all access to the
file is done using RMS services rather than regular
FORTRAN 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

PROGRAM FORPARSE_INDEX
IMPLICIT NONE

REAL PARAMETER ((TIME_WAIT=10.0) ! No. seconds wait
             ) if record locked

INTEGER*2 INREC_LENGTH/80/
INTEGER*2 KEY_VALUE_LENGTH/7/

STRUCTURE /ACCOUNT_STRUCT/
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_STRUCT/ IN_REC

INTEGER FAB_PTR,
1 RAB_PTR,
2 RET_STATUS,
3 LENGTH_FDLFILE,
4 KEY_IN,
5 INREC_PTR,
6 KEY_VALUE_PTR
7 CHARACTERS80 FDL_FILE
8 CHARACTERS7 KEY_VALUE
9 CHARACTERS1 DELETE_FLAG
INTEGER PROG_STAT /0/  ! VALUE 1 = no error
     VALUE 2 = some error
     VALUE 3 = wait-read-again

INTEGER FDL$PARSE,
    1 SYSSOPEN,
    2 SYSSCONNECT,
    3 SYSGET,
    4 SYSSDELETE,
    5 SYSCLOSE

INCLUDE '($RMSDEF)

CALL LIB$INIT_TIMER()

WRITE (6,1)
READ (5,2) length_FDLFILE, FDL_FILE

RET_STATUS = FDL$PARSE(
    1 FDL_FILE(1:length_FDLFILE),
    2 FAB_PTR,            ! Ptr to FAB returned by FDL
    3 RAB_PTR,...         ! Ptr to RAB returned by FDL

IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))

CALL SETFAB(%VAL(FAB_PTR))

RET_STATUS = SYSSOPEN(%VAL(FAB_PTR),)
IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))

INREC_PTR = %LOC(IN_REC)  ! internal addr input rec buff
KEY_VALUE_PTR = %LOC(KEY_VALUE)  ! internal addr key value

CALL SETRAB(
    1 %VAL(RAB_PTR),  ! Ptr to RAB
    2 INREC_PTR,     ! Internal addr record buffer
    3 INREC_LENGTH,  ! Length record buffer
    4 KEY_VALUE_PTR, ! Internal addr key value
    5 KEY_VALUE_LENGTH ! Length key value

RET_STATUS = SYSSCONNECT(%VAL(RAB_PTR),)
IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))

WRITE (6,6)  ! directions to stop run
WRITE (6,5)
WRITE (6,6)
WRITE (6,5)
WRITE (6,3)
READ (5,4) KEY_IN
DO WHILE (KEY_IN .GT. 0)
  Convert integer KEY_IN to right justified character string KEY_VALUE
  IF (PROG_STAT .NE. 3)
    WRITE (UNIT=KEY_VALUE,FMT='(I7)') KEY_IN
    RET_STATUS = SYS$GET(%VAL(RAB_PTR),,)
  IF (.NOT.RET_STATUS) THEN
    PROG_STAT = 2
    IF (RET_STATUS .EQ. RMS$ RNF) THEN
      WRITE (6,11) KEY_VALUE
    ELSE
      IF (RET_STATUS .EQ. RMS$ RLK) THEN
        WRITE (6,12)
        CALL LIB$WAIT(TIME_WAIT)
        PROG_STAT = 3
      ELSE
        CALL LIB$STOP(%VAL(RET_STATUS))
      END IF
    END IF
  END IF
  ELSE
    PROG_STAT = 1
  END IF
  IF (PROG_STAT .EQ. 1) THEN
    WRITE (6,*) IN_REC_SEQ_NO, ',IN_REC_SOC_SEC, ',
    IN_REC_LAST_NAME
    WRITE (6,14)
    READ (5,15) DELETE_FLAG
    IF (DELETE_FLAG .EQ. 'Y'
       .OR. DELETE_FLAG .EQ. 'Y') THEN
      RET_STATUS = SYS$DELETE(%VAL(RAB_PTR),,)
      IF (.NOT.RET_STATUS) CALL LIB$STOP(%VAL(RET_STATUS))
    END IF
    END IF  ! prog_stat = 1
  IF (PROG_STAT .NE. 3) THEN
    WRITE (6,3)
    READ (5,4) KEY_IN
  END IF  ! prog_stat .ne. 3
END DO
CALL SYS$CLOSE(VAL(FAB_PTR),)
CALL LIB$SHOW_TIMER()
CALL EXIT

1 FORMAT ('$Enter name of FDL file: '
2 FORMAT (0,A)
3 FORMAT ('$Enter SEQ_NO: '
4 FORMAT (I)
5 FORMAT (************
6 FORMAT (************
7 FORMAT ('Hit <CR> or enter zero to stop run')
8 FORMAT ('Record NOT FOUND with SEQ_NO = ',A)
9 FORMAT ('Record currently LOCKED ',
10 FORMAT (' - will try again shortly')
11 FORMAT ('$Do you wish to delete this record? (Y/<CR>): '
12 FORMAT (A)
13 END
EXTERNAL FORTRAN routines called by COBPARSE_INDEX.COB or
by FORPARSE_INDEX.FOR

SUBROUTINE SETFAB (FAB)
IMPLICIT NONE

INCLUDE '($FABDEF)'

RECORD /FABDEF/ FAB

! 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

! 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$M_SHNIL
FAB.FAB$B_SHR = FAB.FAB$B_SHR .OR. FAB$M_SHRPUT

1 .OR. FAB$M_SHRGET .OR. FAB$M_SHRDEL
2 .OR. FAB$M_SHRUPD

RETURN
ENDE

SUBROUTINE SETRAB (RAB,REC_PTR,REC_LENGTH,KEY_PTR,KEY_LENGTH)
IMPLICIT NONE

INTEGER*4 REC_PTR,KEY_PTR
INTEGER*2 REC_LENGTH,KEY_LENGTH

INCLUDE '($RABDEF)'

RECORD /RABDEF/ RAB

! 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
FORTRAN (Sheet 6 of 6)

! Specify key of reference
RAB.RABS_KRF = 0  ! primary key

! Specify number of local buffers you want RMS to allocate on CONNECT
RAB.RABS_MB = 003

! Specify type of mode to use for record access
RAB.RABS_RAC = RABSC_KEY  ! RABSC_SEQ or RABSC_RFA

! Provide address and length of output record buffer
RAB.RABSL_RBF = REC_PTR
RAB.RABSW_RSZ = REC_LENGTH

! Provide address and length of input record buffer
RAB.RABSL_UBF = REC_PTR
RAB.RABSW_USZ = REC_LENGTH

! Enable any record processing options to be used for entire run
RAB.RABSL_ROP = RAB.RABSL_ROP .OR. RABSM_FDL .OR. fast delete

RETURN
PASPARSE_INDEX.PAS

$LINK PASPARSE_INDEX,PAS_SETRMS

PAS_SETRMS is EXTERNAL module containing two routines:

SETFAB
SETRAB

PASCAL program using FDLPARSE to set RMS control blocks for
an indexed file. Thereafter, all access is done using
calls to RMS directly. User is prompted for SEQ_NO
of record to be retrieved randomly and given option
of deleting any records retrieved.

Program assumes FILE SHARING and RECORD LOCKING

[INHERIT ('SYS$LIBRARY: STARLET')]

PROGRAM PASPARSE_INDEX(INPUT,OUTPUT);

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;

WORD_INTEGER = [WORD] 0..64534;

PTR_TO_FAB = ^FABTYPE;
PTR_TO_RAB = ^RABTYPE;

VAR

FAB_PTR : PTR_TO_FAB;
RAB_PTR : PTR_TO_RAB;
IN_REC : ACCOUNT_STRUC;
INREC_LENGTH : WORD INTEGER := 80;
RET_STATUS : INTEGER;
FDL_FILENAME : [VOLATILE] PACKED ARRAY [1..80] OF CHAR;
KEY_IN : INTEGER;
PASCAL (Sheet 2 of 6)

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 PROG_STAT : INTEGER := 0; [ VALUE 1 = no error
59           VALUE 2 = some error
60           VALUE 3 = wait-read-again ]
61 PROCEDURE LIB$WAIT (num_secs : REAL); EXTERN;
62 PROCEDURE LIB$STOP (IMMED cond_value : INTEGER); EXTERN;
63 FUNCTION FDL$PARSE (FDLFILE : [CLASS_S] PACKED ARRAY
64               [.U INTEGER] OF CHAR;
65 VAR FAB_PTR : PTR_TO_FAB;
66 VAR RAB_PTR : PTR_TO_RAB); INTEGER; EXTERN;
67 PROCEDURE SETFAB (VAR FAB : FABTYPE); EXTERN;
68 PROCEDURE SETRAB (VAR RAB : RABTYPE;
69 INREC_PTR : INTEGER;
70 INREC_LENGTH : WORD_INTEGER;
71 KEY_VALUE_PTR : INTEGER;
72 KEY_LENGTH : WORD_INTEGER); EXTERN;
73 PROCEDURE LIB$INIT_TIMER (VAR HANDLER_ADR : INTEGER := $IMMED 0); EXTERN;
74 PROCEDURE LIB$SHOW_TIMER (HANDLER_ADR : INTEGER := $IMMED 0;
75 CODE : INTEGER := $IMMED 0;
76 [IMMEDIATE,UNBOUND] PROCEDURE ACTION_RTN (OUT_STR :
77 [CLASS_S] PACKED ARRAY [.U INTEGER]
78 OF CHAR); := $IMMED 0;
79 $IMMED USER_ARG : INTEGER := $IMMED 0); EXTERN;
80 BEGIN [ MAIN ]
81 LIB$INIT_TIMER;
82 WRITE ('Enter FDL filename: '); WRITEL (FDL_FILENAME);
83 READLN (FDL_FILENAME);
84 RET_STATUS := FDL$PARSE (FDL_FILENAME, FAB_PTR, RAB_PTR);
85 IF NOT ODD(RET_STATUS) THEN LIB$STOP (RET_STATUS);
86

13-72
PASCAL (Sheet 3 of 6)

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 WRITELN (KEY_VALUE,KEY_IN:7);
133
134 RET_STATUS := $GET(%IMMED RAB_PTR,);
135 IF NOT ODD (RET_STATUS) THEN
136 BEGIN
137 PROG_STAT := 2;
138 END;
139
140 CASE RET_STATUS OF
141
142 RMS$_RNF : BEGIN
143 WRITELN (' Record NOT FOUND with ',
144 ' SEQ_NO = ',KEY_VALUE);
145 END;
146
147 RMS$_RLK : BEGIN
148 WRITELN (' Record currently LOCKED ',
149 ' - will try again shortly');
150 LIB$WAIT (TIME_WAIT);
151 PROG_STAT := 3;
152 END
153
154 OTHERWISE
155
13-73
LIB$STOP (%IMMED RET_STATUS);

END [ CASE ]

END [ IF NOT ODD ]
ELSE PROG_STAT := 4;

IF (PROG_STAT = 1) THEN
BEGIN

WRITELN (IN_REC_SEQ_NO,' ',IN_REC_SOC_SEC,' ','IN_REC_LAST_NAME);

WRITE ('Do you wish to delete this record?','
' (Y/any char): ');

READLN (DELETE_FLAG);

IF ((DELETE_FLAG = 'Y') OR (DELETE_FLAG = 'y')) THEN
BEGIN

RET_STATUS := $DELETE(%IMMED RAB_PTR,);

IF NOT ODD(RET_STATUS) THEN

LIB$STOP(RET_STATUS);

END;

END; [ prog_stat = 1 ]

IF (PROG_STAT <> 3) THEN
BEGIN

WRITE ('Enter SEQ_NO: ');

READLN (KEY_IN);

END; [ prog_stat <> 3 ]

END; [ while loop ]

CLOSE(%IMMED FAB_PTR,);

LIB$SHOW_TIMER;

END. [ Main program ]
EXTERNAL PASCAL routines called by COBPARSE_INDEX.COB
and by PARSER_INDEX.PAS

(INHERIT ('SYS$LIBRARY: STARLET'))

MODULE RMS_SETTINGS;

TYPE
  WORD_INTEGER = [WORD] 0..64534;

PROCEDURE SETFAB (VAR FAB : FAB$TYPE);
BEGIN

  Extension # blocks if file is extended }
  FAB.FAB$W_DEQ := 10;

  File access desired for USER }
  FAB.FAB$B_FAC := FAB$M_DEL + FAB$M_GET + FAB$M_PUT + FAB$M_UPD;

  File options desired }
  FAB.FAB$L_FOP := FAB$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$M_SHNIL }
  FAB.FAB$B_SHR := FAB$M_SHPUT + FAB$M_SHRGET + FAB$M_SHRDDEL
                   + FAB$M_SHRUPD;

END;  [ SETFAB ]

PROCEDURE SETRAB (VAR RAB : RAB$TYPE;
  REC_PTR : INTEGER;
  REC_LENGTH : WORD_INTEGER;
  KEY_PTR : INTEGER;
  KEY_LENGTH : WORD_INTEGER);
BEGIN

  Provide address of key_value if keyed retrieval to be used
  and size of key }
  RAB.RAB$B_KEF := KEY_PTR;
  RAB.RAB$B_KSZ := KEY_LENGTH;
PASCAL (Sheet 6 of 6)

53 [ Specify key of reference ]
54 RAB.RAB$B_KRF := 0;  \{ primary key \}
55 [ Specify number of local buffers you want RMS to allocate on CONNECT ]
56 RAB.RAB$B_MBF := 3;
57
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$B_RBF := REC_PTR;
64 RAB.RAB$W_RSZ := REC_LENGTH;
65
66 [ Provide address and length of input record buffer ]
67 RAB.RAB$L_RBF := REC_PTR;
68 RAB.RAB$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 \}

13-76
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
- Source files and subroutines
- RMS procedures - PRAGUE and SHIELD

Source
CRAFTS KM from Applications - Chapter 4
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.

Example

SESSION  

TAG,DATE([TIMESTAMP])  

SESSION
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$N_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$N_FNB field. In cases where the program assumes only a single file, the results of this test may be used to explicitly disallow wildcards or search lists.
NAM$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.
<table>
<thead>
<tr>
<th>Field Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMS$V_SEARCH_LIST</td>
<td>A search list logical name is present in the file specification.</td>
</tr>
<tr>
<td>NAMS$V_WILD_CARD</td>
<td>File specification string includes a wildcard; returned whenever any of the other wildcard bits are set.</td>
</tr>
<tr>
<td>NAMS$V_WILD_DIR</td>
<td>Directory specification includes a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_GRP</td>
<td>Group number contains a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_MBR</td>
<td>Member number contains a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_NAME</td>
<td>File name contains a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_SFD1_to_SFD7</td>
<td>Subdirectory 1 through 7 specification includes a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_TYPE</td>
<td>File type contains a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_UPD</td>
<td>User file directory specification includes a wildcard character(s).</td>
</tr>
<tr>
<td>NAMS$V_WILD_VER</td>
<td>Version number contains a wildcard character(s).</td>
</tr>
</tbody>
</table>
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_STMT
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,
, ! default file spec.
3 RELATED_SPEC,
, ! stv_addr
4 USER_FLAGS)

IF (RET_STATUS .EQ. RMSS_NMF) GO TO 300

14-6
Example 1 (Sheet 2 of 2)

OPEN (UNIT=IUNIT, FILE=RESULT_SPEC, STATUS='OLD',
       FORM='UNFORMATTED', RECORDTYPE='FIXED',
       ORGANIZATION='INDEXED', ACCESS='SEQUENTIAL')

READ (IUNIT, END=200, IOSTAT=RET_STATUS) IN_REC

DO WHILE (RET_STATUS .EQ. 0)

   WRITE (6,*) IN_REC.LAST_NAME, ', IN_REC.SEQ_NO

   READ (IUNIT, IOSTAT=RET_STATUS) IN_REC

   IF (RET_STATUS .EQ. -1) GOTO 200

END DO

CALL ERRSNS(RMS_STS, RMS_STV, IUNIT,)
CALL LIB$STOP($VAL(RMS_STS), $VAL(RMS_STV))

200 CLOSE (IUNIT)

GO TO 100

300 CALL EXIT

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 finds 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 FILE/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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOWILD</td>
<td>If set, LIB$FIND_FILE returns an error if a wildcard is input.</td>
</tr>
<tr>
<td>1</td>
<td>MULTIPLE</td>
<td>If set, this performs temporary defaulting for multiple input files and the related-spec argument is ignored. See the description of context in LIB$FILE_SCAN. Each time LIB$FIND_FILE is called with a different file specification, the specification from the previous call is automatically used as a related file specification. This allows parsing of the elements of a search list logical name such as DISK2:[SMITH] FILL.TYP,FIL*2.TYP, and so on. Use of this feature is required to get the desired defaulting with search list logical name. LIB$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.</td>
</tr>
</tbody>
</table>

**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, LIB$GET_VM, LIB$FREE_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)

\[ \text{PASSEARCH.PAS} \]

\[ \text{NOTE: Test data for this program} \quad {\text{INTER*.DAT}} \]

In PASCAL you do not have to call FDL$PARSE to set up RMS structures. You are able to set them up and initialize them directly in PASCAL.

This PASCAL program sets up the RMS structures and does all access to a sequential file using RMS services rather than using any regular PASCAL I/O.

It also checks whether a wild card is used in the file spec. or a logical name search list by calling $PARSE.

If either is used then $SEARCH is called repeatedly until there are no further file name translations.

\[ \text{[INHERIT('SYS$LIBRARY:STARLET')]} \]

\[ \text{PROGRAM PASSEARCH(INPUT,OUTPUT);} \]

\[ \text{CONST} \]

\[ \text{STR_LEN = 255;} \]

\[ \text{NAM$V_WILD_VER = 3;} \]

\[ \text{NAM$V_WILD_TYPE = 4;} \]

\[ \text{NAM$V_WILD_NAME = 5;} \]

\[ \text{NAM$V_WILDCARD = 8;} \]

\[ \text{NAM$V_SEARCH_LIST = 11;} \]

\[ \text{NAM$V_WILD_DIR = 20;} \]

\[ \text{TYPE} \]

\[ \text{EMPLOYEE_STRUC = RECORD} \]

\[ \text{LASTNAME : PACKED ARRAY [1..10] OF CHAR;} \]

\[ \text{FILLER : PACKED ARRAY [1..100] OF CHAR;} \]

\[ \text{SEQNO : PACKED ARRAY [1..2] OF CHAR;} \]

\[ \text{END;} \]

\[ \text{BIT_VALUE = 0..1;} \]

\[ \text{BIT_TYPE = PACKED RECORD} \]

\[ \text{BIT_ARRAY : [BIT(32),POS(0)] PACKED ARRAY} \]

\[ \text{[0..31] OF BIT_VALUE;} \]

\[ \text{END;} \]

\[ \text{VAR} \]

\[ \text{IN_REC : EMPLOYEE_STRUC;} \]

\[ \text{FAB : FAB$TYPE;} \]

\[ \text{RAB : RAB$TYPE;} \]

\[ \text{NAM : NAM$TYPE;} \]

\[ \text{RET_STATUS : INTEGER;} \]

\[ \text{MOREFILES : INTEGER := 1;} \]

\[ \text{FMN_STR : PACKED ARRAY [1..255] OF CHAR;} \]

\[ \text{RES_STR : PACKED ARRAY [1..255] OF CHAR;} \]

\[ \text{EXP_STR : PACKED ARRAY [1..255] OF CHAR;} \]

\[ \text{BIT_SET : UNSIGNED := 0;} \]

\[ \text{INDEX : 0..255;} \]

\[ \text{PARSE_CHK : ARRAY [1..6] OF INTEGER :=} \]

\[ \text{(NAM$V_WILD_VER,NAM$V_WILD_TYPE,} \]

\[ \text{NAM$V_WILD_NAME,NAM$V_WILDCARD,} \]

\[ \text{NAM$V_SEARCH_LIST,NAM$V_WILD_DIR);} \]

\[ \text{BIT_SUM : UNSIGNED := 0;} \]

14-11
Example 2 (Sheet 2 of 4)

PROCEDURE LIBSSTOP(
  %IMMED cond_value : INTEGER); EXTERN;
BEGIN
PROCEDURE SEARCHFILE;
BEGIN
  RET_STATUS := $SEARCH(FAB);
  IF (RET_STATUS = RMS$_NMF) THEN MOREFILES := 0;
  FOR INDEX := 1 TO NAM$_NAM$B_RSL DO
    WRITE (RES_STR[INDEX]);
  WRITELN;
END { searchfile procedure }
PROCEDURE PARSEFILE;
BEGIN
  FAB := ZERO;
  FAB.FABS$B_BID := FABS$C_BID;
  FAB.FABS$B_BLN := FABS$C_BLN;
  FAB.FABS$L_FNA := IADDRESS(FNM$_STR);
  FAB.FABS$B_FNS := STR_LEN;
  FAB.FABS$B_ORG := FABS$C_SEQ;
  FAB.FABS$B_FAC := FABS$M_GET;
  FAB.FABS$L_NAM := IADDRESS(NAM);
  FAB.FABS$L_FOP := FABS$M_NAM;
  NAM := ZERO;
  NAM.NAM$B_BID := NAM$C_BID;
  NAM.NAM$B_BLN := NAM$C_BLN;
  NAM.NAM$L_RSA := IADDRESS(RES$_STR);
  NAM.NAM$B_RSS := STR_LEN;
  NAM.NAM$L_ESA := IADDRESS(EXP$_STR);
  NAM.NAM$B_ESS := STR_LEN;
  RAB := ZERO;
  RAB.RABS$B_BID := RABS$C_BID;
  RAB.RABS$B_BLN := RABS$C_BLN;
  RAB.RABS$L_FAB := IADDRESS(FAB);
  RAB.RABS$B_RAC := RABS$C_SEQ;
  RET_STATUS := $PARSE(FAB);
  FOR INDEX := 1 TO 6 DO
    BEGIN
      BIT_SET := NAM.NAM$L_FNB:BIT_TYPE.BIT_ARRAY
                  [PARSE_CHK[INDEX]];
      BIT_SUM := BIT_SUM + BIT_SET;
    END;
  IF (BIT_SUM > 0) THEN SEARCHFILE;
END { parsefile procedure }
PROCEDURE OPENFILE;
BEGIN

Example 2 (Sheet 3 of 4)

121  RET_STATUS := $OPEN(FAB);
122
123  IF ODD(RET_STATUS) THEN
124      RET_STATUS := $CONNECT(RAB)
125  ELSE
126      LIB$STOP(RET_STATUS);
127  END; [ useropen procedure ]
128
129  PROCEDURE GETFILE;
130
131  BEGIN
132
133      RAB.RAB$L_UBF := IADDRESS(IN_REC);
134      RAB.RAB$W_USZ := SIZE(IN_REC);
135
136      RET_STATUS := $GET(RAB);
137
138      WHILE ODD(RET_STATUS) DO
139      BEGIN
140
141         WRITELN (IN_REC.LASTNAME, ', ', IN_REC.SEQNO);
142
143         RET_STATUS := $GET(RAB);
144      END;
145
146      IF (RET_STATUS <> RMS$_EOF) THEN
147      BEGIN
148      LIB$STOP (RET_STATUS);
149      END;
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
175      END;
176
177      END;
178
179  END. [ MAIN ]
Example 2 (Sheet 4 of 4)

S RUN PASSEARCH
Enter filename: INTER6.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$INSTRUCTOR:[WOODS.RMS.DATA]INTER5.DAT;1
ASHE
JONES
RAKOS
TODD
VAIIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER6.DAT;1
ASHE
BUSH
JONES
RAKOS
TODD
VAIIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER7.DAT;1
ASHE
BUSH
EVANS
JONES
RAKOS
TODD
VAIIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER8.DAT;1
ASHE
BUSH
EVANS
JONES
RAKOS
SACK
TODD
VAIIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER9.DAT;1
ASHE
BUSH
EVANS
JONES
MAYO
RAKOS
SACK
TODD
VAIIL
DISK$INSTRUCTOR:[WOODS.RMS.DATA]INTER8.DAT;1

14-14
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 (WSEXENTENT) 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.
<table>
<thead>
<tr>
<th>Limit</th>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTLM</td>
<td>24</td>
<td>N</td>
<td>AST queue limit</td>
</tr>
<tr>
<td>BIOLM</td>
<td>18</td>
<td>N</td>
<td>Buffered I/O count limit</td>
</tr>
<tr>
<td>BYTLM</td>
<td>8192</td>
<td>P</td>
<td>I/O byte count limit</td>
</tr>
<tr>
<td>CPU</td>
<td>0</td>
<td>D</td>
<td>CPU time limit (0 = no limit)</td>
</tr>
<tr>
<td>DIOLM</td>
<td>18</td>
<td>N</td>
<td>Direct I/O count limit</td>
</tr>
<tr>
<td>ENQLM</td>
<td>30</td>
<td>P</td>
<td>Enqueue quota</td>
</tr>
<tr>
<td>FILLM</td>
<td>20</td>
<td>P</td>
<td>Open file limit</td>
</tr>
<tr>
<td>JTQUOTA</td>
<td>1024</td>
<td>P</td>
<td>Initial byte quota for job-wide logical name table</td>
</tr>
<tr>
<td>MAXACCTJOBS</td>
<td>0</td>
<td>S</td>
<td>Maximum active processes for a single account (0 = no limit)</td>
</tr>
<tr>
<td>MAXDETACH</td>
<td>0</td>
<td>S</td>
<td>Maximum detached processes for a single username (0 = no limit)</td>
</tr>
<tr>
<td>MAXJOBS</td>
<td>0</td>
<td>S</td>
<td>Maximum active processes for a single username (0 = no limit)</td>
</tr>
<tr>
<td>PGFLQUO</td>
<td>12800</td>
<td>P</td>
<td>Paging file limit</td>
</tr>
<tr>
<td>PRCLM</td>
<td>2</td>
<td>P</td>
<td>Subprocess creation limit</td>
</tr>
<tr>
<td>SHRFILEM</td>
<td>0</td>
<td>P</td>
<td>Maximum number open shared files (0 = no limit)</td>
</tr>
<tr>
<td>TQELM</td>
<td>10</td>
<td>P</td>
<td>Timer queue entry limit</td>
</tr>
</tbody>
</table>
Process Resource Limits, Suggested Values, Types, and Descriptions (Cont.)

<table>
<thead>
<tr>
<th>Limit</th>
<th>Value</th>
<th>Type*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSDEF</td>
<td>300</td>
<td>N</td>
<td>Default working set size</td>
</tr>
<tr>
<td>WSEXTENT</td>
<td>700</td>
<td>N</td>
<td>Working set extent</td>
</tr>
<tr>
<td>WSQUO</td>
<td>350</td>
<td>N</td>
<td>Working set quota</td>
</tr>
</tbody>
</table>

* 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 #19
# PART 6. ANALYZE/RMS/INTERACTIVE

## ANALYZE/RMS—FILE Interactive Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AGAIN</strong></td>
<td>Displays the current structure again.</td>
</tr>
<tr>
<td><strong>DOWN [branch]</strong></td>
<td>Move the structure pointer down to the next level. If the current node has more than one branch, the branch keyword must be specified.</td>
</tr>
<tr>
<td>DOWN AREA</td>
<td></td>
</tr>
<tr>
<td>DOWN KEY</td>
<td></td>
</tr>
<tr>
<td>DOWN DATA</td>
<td></td>
</tr>
<tr>
<td>DOWN BYTES</td>
<td>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 &quot;DOWN ?.&quot;</td>
</tr>
<tr>
<td>DOWN RRV</td>
<td></td>
</tr>
<tr>
<td>DOWN SIDRS</td>
<td></td>
</tr>
<tr>
<td><strong>DUMP n</strong></td>
<td>Displays a hexadecimal dump of the specified block.</td>
</tr>
<tr>
<td><strong>EXIT</strong></td>
<td>End the interactive session.</td>
</tr>
<tr>
<td><strong>FIRST</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>HELP [keyword...]</strong></td>
<td>Displays help messages about the interactive commands.</td>
</tr>
<tr>
<td><strong>NEXT</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>REST</strong></td>
<td>Moves the structure pointer along the rest of the structures on the current level, and each is displayed in turn.</td>
</tr>
<tr>
<td><strong>TOP</strong></td>
<td>Moves the structure pointer up to the file header. The file header is displayed.</td>
</tr>
<tr>
<td><strong>UP</strong></td>
<td>Moves the structure pointer up to the next level. The structure at that level is displayed.</td>
</tr>
</tbody>
</table>
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.
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Character</td>
<td>$X'00'$</td>
</tr>
<tr>
<td>Area Number</td>
<td>0</td>
</tr>
<tr>
<td>VBN Sample</td>
<td>4</td>
</tr>
<tr>
<td>Free Space Offset</td>
<td>$X'0104'$</td>
</tr>
<tr>
<td>Free Record ID Range</td>
<td>4-255</td>
</tr>
<tr>
<td>Next Bucket VBN</td>
<td>4</td>
</tr>
<tr>
<td>Level</td>
<td>0</td>
</tr>
<tr>
<td>Bucket Header Flags</td>
<td></td>
</tr>
<tr>
<td>(0) BKT$V_LASTBKT</td>
<td>1</td>
</tr>
<tr>
<td>(1) BKT$V_ROOTBKT</td>
<td>0</td>
</tr>
<tr>
<td>ANALYZE&gt; DOWN KEY</td>
<td></td>
</tr>
<tr>
<td>ANALYZE&gt; DOWN BYTES</td>
<td></td>
</tr>
<tr>
<td>01234567</td>
<td></td>
</tr>
</tbody>
</table>

```
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
31 30 30 30 30 30 00 49
```

```
31 30 30 30 30 30 00 49
```

```
0008 DIGITAL
```

```
0010 EQUIPMEN
```

```
0018 T CORPOR
```

```
0020 ATION 11
```

```
0028 0 SPIT B
```

```
0030 ROOK ROA
```

```
0038 D NASHUA
```

```
0040 NH03
```

```
0048 061
```
Sample Interactive Sessions

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-DOWNNPATH, AREAS Area descriptors
%ANLRMS-I-DOWNNPATH, KEYS Key descriptors

ANALYZE> DOWN KEYS
Example 1 (Sheet 2 of 15)

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) KEYSV_DUPKEYS 0
(3) KEYSV_IDX COMPR 0
(4) KEYSV_INITIDX 0
(6) KEYSV_KEY COMPR 0
(7) KEYSV_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:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>53</td>
<td>45</td>
<td>4E</td>
<td>FF</td>
<td>4A</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>0068</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ANALYZE> NEXT

INDEX RECORD (VBN 9, offset $X'007C'

2-Byte Bucket Pointer: 8
Key:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
</tbody>
</table>

ANALYZE> DOWN ?
%ANLRMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNSPATH, DEEPER
Index or data buckets

ANALYZE> DOWN DEEPER

BUCKET HEADER (VBN 8)
### Example 1 (Sheet 5 of 15)

**ANALYZE> DUMP 8**

**DUMP OF VIRTUAL BLOCK 8:**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>0000</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0008</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0010</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0018</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0020</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0028</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0030</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0038</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0040</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0048</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0050</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0058</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0060</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0068</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0070</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0078</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0080</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0088</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0090</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>0098</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00A0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00A8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00B0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00B8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00C0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00C8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00D0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00D8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00E0</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>00E8</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>00F0</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>00F8</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0100</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0108</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0110</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0118</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0120</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0128</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0130</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0138</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0140</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0148</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>0150</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0158</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0160</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0168</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0170</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0178</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0180</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0188</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0190</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0198</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01A0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01A8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01B0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01B8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01C0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01C8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01D0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01D8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01E0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01E8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01F0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>01</td>
<td>FF</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0F</td>
<td>01F8</td>
</tr>
</tbody>
</table>
Example 1 (Sheet 6 of 15)

ANALYZE> UP

KEY DESCRIPTOR $0 (VBN 1, offset $'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)

<table>
<thead>
<tr>
<th>Check Character: $'09'</th>
<th>Key of Reference: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBN Sample: 4</td>
<td></td>
</tr>
<tr>
<td>Free Space Offset: $'011B'</td>
<td></td>
</tr>
<tr>
<td>Free Record ID: 6</td>
<td></td>
</tr>
<tr>
<td>Next Bucket VBN: 6</td>
<td></td>
</tr>
<tr>
<td>Level: 0</td>
<td></td>
</tr>
</tbody>
</table>

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 (VBN 4, offset $'0000')

PRIMARY DATA RECORD (VBN 4, offset $'0000')

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

<table>
<thead>
<tr>
<th>Key:</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 20 20 20 45 48 53 41</td>
<td>0000</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0008</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0010</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0018</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0020</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0028</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0030</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0038</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0040</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0048</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0050</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0058</td>
</tr>
<tr>
<td>20 20 20 20 20 20 20 20</td>
<td>0060</td>
</tr>
</tbody>
</table>

16-9
### Example 1 (Sheet 7 of 15)

**PRIMARY DATA RECORD (VBN 4, offset \$X'0087')**

**Record Control Flags:**
- (2) IRC\$V_DELETED 0
- (3) IRC\$V_RRV 0
- (4) IRC\$V_NOPTR8Z 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:**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>48</td>
<td>53</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>0000</td>
<td>0008</td>
<td>0010</td>
<td>0018</td>
<td>0020</td>
<td>0028</td>
<td>0030</td>
<td>0038</td>
</tr>
<tr>
<td>0040</td>
<td>0048</td>
<td>0050</td>
<td>0058</td>
<td>0060</td>
<td>0068</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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_NOPTR8Z 0
- (5) IRC\$V_RU_DELETE 0
- (6) IRC\$V_RU_UPDATE 0

**Record ID:** 1  
**RRV ID:** 1, 4-Byte Bucket Pointer: 5

---

16-10
Example 1 (Sheet 8 of 15)

** 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:

- IRC$V_DELETED: 0
- IRC$V_RRV: 1
- IRC$V_NOPTRSZ: 0
- IRC$V_RU_DELETE: 0
- 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:

- IRC$V_DELETED: 0
- IRC$V_RRV: 1
- IRC$V_NOPTRSZ: 0
- IRC$V_RU_DELETE: 0
- 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:

- BKT$V_LASTBKT: 0

---

16-11
Example 1 (Sheet 9 of 15)

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.
### Dump of Virtual Block 4:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>00</td>
</tr>
<tr>
<td>00001</td>
<td>00</td>
</tr>
<tr>
<td>00002</td>
<td>00</td>
</tr>
<tr>
<td>00003</td>
<td>00</td>
</tr>
<tr>
<td>00004</td>
<td>00</td>
</tr>
<tr>
<td>00005</td>
<td>00</td>
</tr>
<tr>
<td>00006</td>
<td>00</td>
</tr>
<tr>
<td>00007</td>
<td>00</td>
</tr>
<tr>
<td>00008</td>
<td>00</td>
</tr>
<tr>
<td>00009</td>
<td>00</td>
</tr>
<tr>
<td>00010</td>
<td>00</td>
</tr>
<tr>
<td>00011</td>
<td>00</td>
</tr>
<tr>
<td>00012</td>
<td>00</td>
</tr>
<tr>
<td>00013</td>
<td>00</td>
</tr>
<tr>
<td>00014</td>
<td>00</td>
</tr>
<tr>
<td>00015</td>
<td>00</td>
</tr>
<tr>
<td>00016</td>
<td>00</td>
</tr>
<tr>
<td>00017</td>
<td>00</td>
</tr>
<tr>
<td>00018</td>
<td>00</td>
</tr>
<tr>
<td>00019</td>
<td>00</td>
</tr>
</tbody>
</table>

### Block Check:

- Address: 0013457
- Data: 00 00 00 00 00 00 00 00

### Next Block:

- Address: 0013457
- Data: 00 00 00 00 00 00 00 00

---

**Example I (Sheet 10 of 15)**

- Block Check: No
- Next Block: Yes
- Space Points:
  - 15 Free Space Points

---

**Check for:**

- Start of Key
- Dump of Virtual Block 4
- Block Check
- Next Block
- Space Points

---

**Sheet:**

- 10 of 15
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) $V_DUPKEYS 0
(1) $V_CHGKEYS 0
(2) $V_MULKEYS 0
(3) $V_IDXCOMPR 0
(4) $V_INITIDX 0
(6) $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
%ANLRMS-I-DOWNPATH, DATA

16-14
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 ?
%ANLREMS-I-DOWNHELP, The following is a list of paths down from this structure:
%ANLREMS-I-DOWPATH, RECORDS Index records

ANALYZE> DOWN RECORDS

INDEX RECORD (VBN 14, offset %X'00E')

- 2-Byte Bucket Pointer: 13
- Key: 7 6 5 4 3 2 1 0 01234567
- FF FF | 0000 | ...

16-15
Example 1 (Sheet 13 of 15)

ANALYZE> 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) BKTSV_LASTBKT 1

ANALYZE> DOWN?
%ANLRMS-I-DOWNSRC, The following is a list of paths down from this structure:
%ANLRMS-I-DOWNSRC, SIDRS SIDR record

ANALYZE> DOWN SIDRS

SIDR RECORD (VBN 13, offset %X'000E')

Key:

7 6 5 4 3 2 1 0 01234567

ANALYZE> REST

SIDR RECORD (VBN 13, offset %X'0017')

Key:

7 6 5 4 3 2 1 0 01234567

SIDR RECORD (VBN 13, offset %X'0020')

Key:

7 6 5 4 3 2 1 0 01234567

SIDR RECORD (VBN 13, offset %X'0029')

Key:

7 6 5 4 3 2 1 0 01234567

16-16
Example 1 (Sheet 14 of 15)

SIDR RECORD (VBN 13, offset \$X'0032'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 05</td>
<td>0000</td>
</tr>
</tbody>
</table>

SIDR RECORD (VBN 13, offset \$X'003B'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 06</td>
<td>0000</td>
</tr>
</tbody>
</table>

SIDR RECORD (VBN 13, offset \$X'0044'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 07</td>
<td>0000</td>
</tr>
</tbody>
</table>

SIDR RECORD (VBN 13, offset \$X'004D'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 08</td>
<td>0000</td>
</tr>
</tbody>
</table>

SIDR RECORD (VBN 13, offset \$X'0056'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 09</td>
<td>0000</td>
</tr>
</tbody>
</table>

SIDR RECORD (VBN 13, offset \$X'005F'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 0A</td>
<td>0000</td>
</tr>
</tbody>
</table>

SIDR RECORD (VBN 13, offset \$X'0068'

<table>
<thead>
<tr>
<th>Key:</th>
<th>7 6 5 4 3 2 1 0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00 0B</td>
<td>0000</td>
</tr>
</tbody>
</table>

%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:**

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>01234567</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>71</td>
<td>00</td>
<td>0D</td>
<td>01</td>
<td>0B</td>
<td>0000</td>
<td>...q...</td>
</tr>
<tr>
<td>00</td>
<td>07</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0D</td>
<td>0008</td>
</tr>
<tr>
<td>07</td>
<td>00</td>
<td>04</td>
<td>00</td>
<td>01</td>
<td>80</td>
<td>00</td>
<td>01</td>
<td>0010</td>
</tr>
<tr>
<td>00</td>
<td>04</td>
<td>00</td>
<td>02</td>
<td>80</td>
<td>00</td>
<td>02</td>
<td>00</td>
<td>0018</td>
</tr>
<tr>
<td>04</td>
<td>00</td>
<td>03</td>
<td>80</td>
<td>00</td>
<td>03</td>
<td>00</td>
<td>07</td>
<td>0020</td>
</tr>
<tr>
<td>00</td>
<td>04</td>
<td>00</td>
<td>04</td>
<td>00</td>
<td>04</td>
<td>07</td>
<td>00</td>
<td>0028</td>
</tr>
<tr>
<td>03</td>
<td>80</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>07</td>
<td>00</td>
<td>04</td>
<td>0030</td>
</tr>
<tr>
<td>80</td>
<td>00</td>
<td>06</td>
<td>00</td>
<td>07</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>0038</td>
</tr>
<tr>
<td>00</td>
<td>07</td>
<td>00</td>
<td>07</td>
<td>00</td>
<td>04</td>
<td>00</td>
<td>05</td>
<td>0040</td>
</tr>
<tr>
<td>08</td>
<td>00</td>
<td>07</td>
<td>00</td>
<td>06</td>
<td>00</td>
<td>01</td>
<td>80</td>
<td>0048</td>
</tr>
<tr>
<td>00</td>
<td>07</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>04</td>
<td>80</td>
<td>00</td>
<td>0050</td>
</tr>
<tr>
<td>07</td>
<td>00</td>
<td>05</td>
<td>00</td>
<td>05</td>
<td>80</td>
<td>00</td>
<td>09</td>
<td>0058</td>
</tr>
<tr>
<td>00</td>
<td>0F</td>
<td>00</td>
<td>04</td>
<td>80</td>
<td>00</td>
<td>0A</td>
<td>00</td>
<td>0060</td>
</tr>
<tr>
<td>0F</td>
<td>00</td>
<td>05</td>
<td>80</td>
<td>00</td>
<td>08</td>
<td>00</td>
<td>07</td>
<td>0068</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0070</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0078</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0080</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0088</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0090</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0098</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00A0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00A8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00B0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00B8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00C0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00C8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00D0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00D8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00E0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00F0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00F8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0100</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0108</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0110</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0118</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0120</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0128</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0130</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0138</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0140</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0148</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0150</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0158</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0160</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0168</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0170</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0178</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0180</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0188</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0190</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0198</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01A0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01A8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01B0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01B8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01C0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01C8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01D0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01D8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01E0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01E8</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01F0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01F8</td>
</tr>
</tbody>
</table>

**ANALYZE** EXIT

16-18
Example 2. Tracking a Record from Key Value to Data Record

(Sheet 1 of 4)

$ ANALYZE/RMS/INTER INTER11.DAT

FILE HEADER

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:

000 (0) KEYSV_DUPKEYS 0
020 (3) KEYSV_IDX_COMP 0
040 (4) KEYSV_INITIDX 0
060 (6) KEYSV_KEY_COMP 0
080 (7) KEYSV_REC_COMP 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'006A'
Free Record ID: 1
Next Bucket VBN: 9
Level: 2
Bucket Header Flags:
(0) BKTSV_LASTBKT 1
(1) BKTSV_ROOTBKT 1
Bucket Pointer Size: 2

VBN Free Space Offset: $X'01F7'

ANALYZE> DOWN RECORDS

INDEX RECORD (VBN 9, offset $X'006A')

2-Byte Bucket Pointer: 7
Key:

```
01 23 45 67 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

```
JONES
```

ANALYZE> DOWN DEEPER

BUCKET HEADER (VBN 7)

Check Character: $X'04,'
Key of Reference: 0
VBN Sample: 7
Free Space Offset: $X'006A'
Free Record ID: 1
Next Bucket VBN: 8
Level: 1
Bucket Header Flags:
(0) BKTSV_LASTBKT 0
(1) BKTSV_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:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>48</td>
<td>53</td>
<td>55</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

01234567

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

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

16-21
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:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Key = 'ASHE'

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 IV
DATA RECOVERY FOR CORRUPTED INDEX FILES
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)

6-DEC-1985 21:47:04.38

FILE HEADER

File Spec: DISK$INSTRUCTOR:[WOODS.RMS.COURSE]CORRUPT1.DAT;1
File ID: (15352,46,0)
Owner UIC: 010,007
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'00000')

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'00040')

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

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 Extent 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) KEYSV_DUPKEYS 0
(3) KEYSV_IDK_COMP 0
(4) KEYSV_INITIDX 0
(6) KEYSV_KEY_COMP 0
(7) KEYSV_REC_COMP 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
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 SYSSOUTPUT. 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_INRE11.LIS INTER11.DAT

14:52:40.35
File ID (33247,11,0)  End of file block 16 / Allocated 16

Virtual block number 1 (00000001), 512 (0200) bytes

14:52:40.35
File ID (33247,11,0)  End of file block 16 / Allocated 16

Virtual block number 2 (00000002), 512 (0200) bytes

17-5
### 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 / Aliquated 166

Record number 1 (00000001), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20454E49 444C4152 45472020 20202020</td>
<td>20202020 20204B41 5A312020 20202020</td>
<td>1IAK GERALDINE</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>59474479 48202020 46544C41 49450C50</td>
<td>410E0832 39135332 39202030 30313828 2010019935192 MAPLE BLVD HIGGY</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32303353 32405420 20202015F 49414D20</td>
<td>MANOR V124502.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 2 (00000002), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2020202E 43204252 41422020 20202020</td>
<td>20202020 20204B4E 4F544C55 46322020 20202020</td>
<td>2FULTON BARB C.</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>4E574F52 42202020 20594157 20524548</td>
<td>41422031 38303336 34383637 36383439 9466764663081 BAKER WAY BROWN</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30393373 39414320 20202045 4C494596</td>
<td>VILLE CA91790.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 3 (00000003), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2020202E 4C205952 41472020 20202020</td>
<td>20202020 20524153 45424549 54332020 20202020</td>
<td>2TIBBSER GARY L.</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>52455481 42205249 4220544E 41524843</td>
<td>4F432039 33303631 38303932 34313436 4641490816099 COCHRANE CIRCBAKER</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36303030 32494420 20202045 4C494596</td>
<td>VILLE D12006.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 4 (00000004), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2020202E 55204C4C 49422020 20202020</td>
<td>20202020 2048E140 52454083 41342020 20202020</td>
<td>4ACKERMAN BILL U.</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>4C414843 4D202044. 414F5220 454C4148</td>
<td>43420D34 35383137 35383336 35353437 7455638571854 MCHALE ROAD MCHAL</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36303030 32494420 20202045 4C494596</td>
<td>ETON D12006.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 5 (00000005), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20284A20 41485452 41420220 20202020</td>
<td>20202020 20204B4E 4F544C55 46352020 20202020</td>
<td>5FULTON MARTHA J.</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>45574843 5320454C 43524943 20594747</td>
<td>49462030 35353533 33313034 30383330 038301335555 HIGGY CIRCLE SCHNE</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30393373 3941434C 4C495654 52454B49</td>
<td>IKERTVILLCA91790.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 6 (00000006), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4D205245 4D495452 4F4D2020 20202020</td>
<td>20202020 20204B4E 4F544C55 46362020 20202020</td>
<td>6FULTON MORTIMER N</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>45574843 53204441 45522054 4E4144CC</td>
<td>41472031 33533834 30343032 37393735 5797304048531 GALLANT ROAD SCHNE</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31303139 3745544C 4C495654 52454B49</td>
<td>IKERTVILLTE7101.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 7 (00000007), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2020202E 5320414E 48412020 20202020</td>
<td>20202020 20454853 41372020 20202020</td>
<td>7ASHE ANNA S.</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>414C4C1 47202020 20202059 41572048</td>
<td>415A2030 30353635 33323431 37324373 3727142356500 ZAK WAY GALLA</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32303354 32494548 47554F4D 4F42544E</td>
<td>NTBOROUGHV124502.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>

Record number 8 (00000008), 80 (0050) bytes

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Value</th>
<th>Address</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>202E4E20 444C414E 4F442020 20202020</td>
<td>20202020 20205541 48382020 20202020</td>
<td>8HAW DONALD N.</td>
<td>000000</td>
</tr>
<tr>
<td></td>
<td>4C414843 4D435249 4320454E 41524843</td>
<td>4F432036 30323432 32333237 31303633 3601723224206 COCHRANE CIRCCHAL</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36313232 334C4620 2020454C 4C495645</td>
<td>EVILLE FL32216.</td>
<td>000040</td>
<td></td>
</tr>
</tbody>
</table>
```

17-7
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

    good_sequential_file_output

    corrupted_index_file

Step 4.

$CONVERT/FDL=new_index.FDL/STAT

    good_sequential new_index

If you do not have an alternate key or the CONVERSION utility using an alternate key is not successful, see Step 7.

6. If the CONVERSION 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 CONVERSION 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.
INTRODUCTION TO THE PATCH UTILITY

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 /NONEWVERSION 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.)
Patch Commands

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.

17-14
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 INTO11_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:[WODS.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 00000000 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 20202020
20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
20202020 20202020 20202020 20202020 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 20202020 20202020 20202020 20202020
20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
20202020 20202020 20202020 20202020 20202020 20202020 20202020 20202020
80000000 80000000 80000000 80000000 80000000 80000000 80000000 80000000
030001F7 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

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

5. $CONVERT/FDL=corrupt4.fdl/STAT seq4.dat GOOD4.DAT

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$INSTRUCT[woods.rms COURSE]CORRUPT4.DAT;1
  being overwritten
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

## Hexadecimal to Decimal Conversion Table

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>1</strong></td>
<td>256</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>512</td>
<td>32</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>768</td>
<td>48</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>1024</td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>1280</td>
<td>80</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>1536</td>
<td>96</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>1920</td>
<td>112</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>2048</td>
<td>128</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>9</strong></td>
<td>2304</td>
<td>144</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>3072</td>
<td>192</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>3276</td>
<td>256</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>3840</td>
<td>307</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>4096</td>
<td>512</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>4608</td>
<td>640</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>5408</td>
<td>819</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
# ASCII Table

<table>
<thead>
<tr>
<th>BITS</th>
<th>COLUMN</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>DLE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SP</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SOH</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DC1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>STX</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>ETX</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>EOT</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>DC3</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>ENQ</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>ACK</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>SYN</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>BEL</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>ETB</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>BS</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>CAN</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>HT</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>EM</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>LF</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>VT</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>CR</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>SO</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>SI</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
</tbody>
</table>

**KEY**

<table>
<thead>
<tr>
<th>ASCII CHARACTER</th>
<th>OCTAL</th>
<th>DECIMAL</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC</td>
<td>33</td>
<td>37</td>
<td>18</td>
</tr>
<tr>
<td>DEL</td>
<td>111</td>
<td>127</td>
<td>7F</td>
</tr>
</tbody>
</table>