OpenVMS Record Management Services Reference Manual

December 1995

This reference manual contains general information intended for use in any OpenVMS programming language, as well as specific information on writing programs that use OpenVMS Record Management Services (OpenVMS RMS).

Revision/Update Information: This manual supersedes the *OpenVMS Record Management Services Reference Manual*, OpenVMS AXP Version 1.5 and OpenVMS VAX Version 6.0.

Software Version: OpenVMS Alpha Version 7.0
OpenVMS VAX Version 7.0

Digital Equipment Corporation
Maynard, Massachusetts

December 1995

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

Digital conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

© Digital Equipment Corporation 1995. All rights reserved.

The following are trademarks of Digital Equipment Corporation: Bookreader, DECnet, DECwindows, Digital, MSCP, OpenVMS, PDP–11, RMS–11, RSX–11M, RSX–11M–PLUS, RSX–11S, RT–11, TOPS–10, TOPS–20, ULTRIX, VAX, VAXcluster, VAX DOCUMENT, VAX MACRO, VMS, VMScluster, and the DIGITAL logo.

The following are third-party trademarks:

MS–DOS is a registered trademark of Microsoft Corporation.

IBM is a registered trademark of International Business Machines Corporation.

All other trademarks and registered trademarks are the property of their respective holders.

ZK4523

This document is available on CD–ROM.

Contents

5 Name Block (NAM)

6 Record Access Block (RAB)

7 64-Bit Record Access Block (RAB64)

8 Allocation Control XAB (XABALL)

9 Date and Time XAB (XABDAT)

10 File Header Characteristic XAB (XABFHC)

11 Item List XAB (XABITM)

12 Journaling XAB (XABJNL)

13 Key Definition XAB (XABKEY)

14 Protection XAB (XABPRO)

15 Revision Date and Time XAB (XABRDT)

16 Recovery Unit XAB (XABRU)

17 Summary XAB (XABSUM)

18 Terminal XAB (XABTRM)

Part III OpenVMS RMS Services

A RMS Control Block Macros

B VAX MACRO Programming Information and Examples

Index

Examples

Figures

Tables

Preface

Intended Audience

This document describes OpenVMS Record Management Services (RMS) control blocks and services for programmers.

Document Structure

This document consists of three parts and two appendixes.

- Part I contains general information in three sections:
  - Chapter 1 introduces the reader to RMS functions and associated control blocks.
  - Chapter 2 discusses the RMS program interface that applies to using any OpenVMS programming language.
  - Chapter 3 describes to advanced high-level language programmers how to use RMS macros.
- Part II describes the RMS control blocks and their associated fields, in Chapter 4 through Chapter 18. This information is intended for a programmer using any programming language.
- Part III describes the record management services, including the control block fields accessed by each service. This information is intended for a programmer using any programming language.
- Appendix A contains the formats and associated usage notes for the RMS control block initialization and store macros used by VAX MACRO programmers.
- Appendix B describes the VAX MACRO programming interface, and how to use RMS macros. This appendix also provides additional descriptions implementing groups of record management services, together with appropriate VAX MACRO example programs.

Related Documents

The following documents contain information related to this reference manual:

- *OpenVMS Programming Interfaces: Calling a System Routine* contains information about calling routines on an OpenVMS system.
- The *Guide to OpenVMS File Applications* contains descriptions of file and record options available to users in a task-oriented format.
- The *OpenVMS Record Management Utilities Reference Manual* contains related information about RMS utilities and the File Definition Language (FDL).

- The *DECnet for OpenVMS Networking Manual* discusses the support of RMS options for remote file access to non-OpenVMS systems. For example, when the remote system is a PDP–11 system running RMS–11, Prolog 3 index files are not supported and some RMS XABKEY (key definition extended attribute block) fields, as well as other control block fields, are not fully supported.

For additional information on OpenVMS products and services, access the Digital OpenVMS World Wide Web site. Use the following URL:

`http://www.openvms.digital.com`

Reader's Comments

Digital welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@zko.mts.dec.com**
Fax	603 881-0120, Attention: OpenVMS Documentation, ZK03-4/U08
Mail	OpenVMS Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Location	Call	Fax	Write
U.S.A.	DECdirect 800–DIGITAL 800–344–4825	Fax: 800–234–2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	809–781–0505	Fax: 809–749–8300	Digital Equipment Caribbean, Inc. 3 Digital Plaza, 1st Street, Suite 200 P.O. Box 11038 Metro Office Park San Juan, Puerto Rico 00910–2138
Canada	800–267–6215	Fax: 613–592–1946	Digital Equipment of Canada, Ltd. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	—	Local Digital subsidiary or approved distributor
Internal Orders	DTN: 264–4446 603–884–4446	Fax: 603–884–3960	U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063–1260

ZK–7654A–GE

Conventions

The name of the OpenVMS AXP operating system has been changed to OpenVMS Alpha. Any references to OpenVMS AXP or AXP are synonymous with OpenVMS Alpha or Alpha.

The following conventions are used to identify information specific to OpenVMS Alpha or to OpenVMS VAX:

Alpha	The Alpha icon denotes the beginning of information specific to OpenVMS Alpha.
VAX	The VAX icon denotes the beginning of information specific to OpenVMS VAX.
♦	The diamond symbol denotes the end of a section of information specific to OpenVMS Alpha or to OpenVMS VAX.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 then press and release another key or a pointing device button.
GOLD *x*	A sequence such as GOLD *x* indicates that you must first press and release the key defined as GOLD then press and release another key. GOLD key sequences can also have a slash (/), dash (–), or underscore (_) as a delimiter in EVE commands. The GOLD key definition is often mapped to the PF1 key on the keypad.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	Horizontal ellipsis points in examples indicate one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.

[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In command format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason (user action that triggers a callback). Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text emphasizes important information and indicates complete titles of manuals and variables. Variables include information that varies in system messages (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Part I

OpenVMS RMS—General Information

Part I introduces the reader to general mechanisms and conventions associated with Record Management Services (RMS). It discusses the following topics:

- Argument passing
- Control blocks
- Symbols defined by RMS
- Invoking RMS services

1
Introduction to RMS

This section presents an overview of the general functions available through RMS. It also briefly describes the record management services and related control blocks.

1.1 RMS Functions

RMS is a set of generalized services that assist application programs in processing and managing files and their contents. OpenVMS languages may invoke these services using appropriate macros stored in system libraries using the OpenVMS calling standard. Record management services are system services identified by the entry point prefix SYS$ followed by three or more characters; but the SYS prefix is omitted in the corresponding VAX MACRO macro name. For example, the Create service has an entry point of SYS$CREATE and a VAX MACRO macro name of $CREATE. A complete description of each service is provided in Part III.

Table 1–1 describes the functions of each service, including the service entry point name and its corresponding VAX MACRO macro name.

Table 1–1 Record Management Services

Service Name	Macro Name	Description
File Processing and File Naming Macros		
SYS$CLOSE	$CLOSE	Terminates file processing and disconnects all record streams previously associated with the file
SYS$CREATE	$CREATE	Creates and opens a new file
SYS$DISPLAY	$DISPLAY	Returns the opened file's attributes to the application program
SYS$ENTER[1]	$ENTER	Enters a file name into a directory
SYS$ERASE	$ERASE	Deletes a file and removes its directory entry
SYS$EXTEND	$EXTEND	Allocates additional space to a file
SYS$OPEN	$OPEN	Opens an existing file and initiates file processing
SYS$PARSE	$PARSE	Parses a file specification
SYS$REMOVE[1]	$REMOVE	Removes a file name from a directory but does not actually delete the file data; compare this with the $ERASE macro

[1]This service is not supported for network operations involving file access between remote OpenVMS systems.

(continued on next page)

Table 1–1 (Cont.) Record Management Services

Service Name	Macro Name	Description
File Processing and File Naming Macros		
SYS$RENAME	$RENAME	Assigns a new name to (renames) a file
SYS$SEARCH	$SEARCH	Searches a directory, or possibly multiple directories, for a file name
Record Processing Macros		
SYS$CONNECT	$CONNECT	Establishes a record stream by associating a RAB with an open file
SYS$DELETE	$DELETE	Deletes a record from a relative or indexed file
SYS$DISCONNECT	$DISCONNECT	Terminates a record stream by disconnecting a RAB from an open file
SYS$FIND	$FIND	Locates the specified record, establishes it as the current record and returns the record's RFA to the application program
SYS$FLUSH	$FLUSH	Writes (flushes) modified I/O buffers and file attributes to the disk before closing a file
SYS$FREE	$FREE	Unlocks all records previously locked by the record stream
SYS$GET	$GET	Retrieves a record from a file
SYS$NXTVOL[1]	$NXTVOL	Causes processing of a magnetic tape file to continue to the next volume of a volume set
SYS$PUT	$PUT	Writes a new record to a file
SYS$RELEASE	$RELEASE	Unlocks a record pointed to by the contents of the RAB$W_RFA field
SYS$REWIND	$REWIND	Establishes the first file record as the current record
SYS$TRUNCATE	$TRUNCATE	Truncates a sequential file
SYS$UPDATE	$UPDATE	Deletes and rewrites (updates) an existing file record
SYS$WAIT	$WAIT	Awaits the completion of an asynchronous record operation
Block I/O Processing Macros		
SYS$READ	$READ	Retrieves a specified number of bytes from a file, beginning on block boundaries
SYS$SPACE	$SPACE	Positions forward or backward in a file to a block boundary
SYS$WRITE	$WRITE	Writes a specified number of bytes to a file, beginning on block boundaries

[1]This service is not supported for network operations involving file access between remote OpenVMS systems.

Although RMS supports unit-record devices, such as terminals and printers, it primarily provides a comprehensive software interface to mass storage devices, such as disk and magnetic tape drives.

RMS provides a variety of disk file organizations, record formats, and record access modes from which you can select the appropriate processing techniques for your application. RMS supports sequential, relative, and indexed file organizations, and fixed-length and variable-length record formats are supported for each file organization. (Relative and sequential files also support other record formats.) The RMS record access modes permit you to access records sequentially, directly by key value, directly by relative record number, or directly by record file address (RFA). RMS also provides a means of performing block I/O operations for users with certain performance-critical applications (such applications may require user-defined file organizations or record formats, or both).

RMS ensures safe and efficient file sharing by providing:

- Multiple file access modes to match file sharing with file operations
- Automatic record locking in applicable file access modes that ensures data integrity during record updates
- Optional buffer sharing to minimize I/O operations when multiple processes access the same file

RMS also enforces the security requirements of a multiuser system with potential multinode access by restricting file access to one or more user types and a list of user names.

For systems that support network capabilities, RMS provides a subset of file and record management services through the data access protocol (DAP) to remote network nodes. Network DAP remote file operations are generally transparent to application programs.

1.2 Passing Arguments to RMS

RMS flexibility requires application programs to pass a multitude of arguments to RMS services for doing common operations such as file creation and file access. To minimize the problems associated with passing numerous arguments for each service call, the application program places the arguments in one or more data control blocks before it invokes a record management service. The only argument required for most services is the symbolic address of the appropriate data control block.

1.2.1 Record Management Services and Control Blocks

Because RMS operates on files and records, its services generally belong to one of two groups:

- File services that create and access a new file, access (open) an existing file, extend the disk space allocated to a file, close a file, obtain file characteristics, and perform other functions related to files
- Record services that get, find (locate), put (insert), update, and delete records, and perform other operations not directly related to record I/O, such as associating one or more record streams (methods of accessing records) with an open file

To support service operations, RMS provides two types of control blocks:

- Control blocks that provide file-related arguments to file services
- Control blocks that provide record-related arguments to record services

1.2.2 Control Blocks for File Services

File services use a control block called the file access block (FAB). When creating a file, the user must store arguments in the FAB that define the file characteristics, the file specification, and certain run-time access options. When your process opens an existing file, the FAB specifies only the file specification and the run-time access options.

There are three categories of FAB arguments; the following list briefly introduces each category.

- **File specification** arguments identify primary and default file specifications used at run time to locate the file.
- **File characteristics** arguments specify the file organization, record type, space allocation information, and other information.
- **Run-time access options** specify the operations that can be done by the initiating process and the operations that can be done by sharing processes, a variety of file-processing options, and the address (or addresses) of one or more control blocks whose fields supplement or supersede the information in the FAB.

The two types of optional control blocks that can supplement or supersede the information in the FAB are the extended attribute block (XAB, pronounced ''zab'') and the NAM block.

A XAB usually supersedes and supplements the file characteristics specified in the associated FAB, and multiple XABs may support a single file. There are several types of XABs, each of which is used for a different purpose. Each type of XAB has a 6-letter mnemonic name consisting of the prefix ''XAB'' followed by a 3-letter mnemonic that relates to the XAB function. For instance, the XAB that supplements and supersedes the file allocation information in the FAB is called an allocation control XAB, or XABALL.

The XABs used for file operations are briefly described in the following list:

- Allocation control XAB (XABALL)—allows greater control over disk file allocation and positioning during file allocation.
- Date and time XAB (XABDAT)—specifies date and time values for when the files was backed up, created, and expired. It also provides the time and date for file revisions and the revision number.

  The XABDAT also provides date and time values for when the file data is effective and when the data was recorded.
- File header characteristic XAB (XABFHC)—receives the file characteristics information contained in the file header block.
- Item list XAB (XABITM)—provides a convenient means for using item lists to pass information between RMS and the application program.
- VAX Journaling XAB (XABJNL)—on VAX systems, supports file journaling operations.♦
- Key definition XAB (XABKEY)—defines the key characteristics to be associated with an indexed file.
- File protection XAB (XABPRO)—defines file protection characteristics that specify what class of users or list of users can have certain specified access rights. For ANSI magnetic tape files using HDR1 labels, this XAB specifies the accessibility field character.

- Revision date and time XAB (XABRDT)—specifies the revision date and time value and the revision number associated with closing a file.

VAX

- Recovery unit XAB (XABRU)—on VAX systems, supports the use of recovery units to assure data file integrity.♦
- Summary XAB (XABSUM)—stores additional file characteristics associated with an indexed file.

1.2.3 Control Blocks for Record Services

Record services use a control block known as the record access block, or RAB. Some of the arguments the user must store in the RAB include the address of the related FAB, the address of input and output record buffers, the type and size of general I/O buffers, whether a file's records will be accessed directly or sequentially, certain tuning options, and other information.

An extended attribute block (XAB) can both supersede and supplement the record characteristics specified in the RAB. As with a XAB that supersedes and supplements a FAB, a XAB that supersedes and supplements a RAB has a 6-letter mnemonic name consisting of the prefix ''XAB'' followed by three letters. Note that there are only two XAB types used for record operations, the terminal XAB (XABTRM) and, on VAX systems only, the recovery unit XAB (XABRU).

The XABTRM defines the symbolic address and length of a user-supplied argument list that defines the terminal operation and provides more flexibility than using RAB fields.

VAX

See the *RMS Journaling for OpenVMS Manual* for details relating to the use of the XABRU on VAX systems.♦

1.2.4 The Dual Purpose of Control Blocks

Control blocks provide input to and output from record management services, including the following run-time information:

- Detailed file characteristics, including file organization, record format, and record size
- Device characteristics
- File, directory, and device identifiers
- The address (location) and length of a requested record
- Returned condition values

For this reason, certain programs specifically allocate a NAM block or one or more XABs dedicated to receiving information returned by RMS. Typically, such information can be examined to determine how the file should be processed.

In most cases, however, control blocks are used both to transmit and to receive information between the application program and RMS, and should not be located in a read-only program section.

Be sure that control block fields not currently used by a particular service have valid default values, because future versions of RMS may use them. This applies also to control block fields that are currently described as ''ignored for DECnet for OpenVMS operations'' because future versions of RMS or DECnet for OpenVMS may support those fields.

A NAM block supplements the file specification information stored in the related FAB. It is especially useful for locating and opening files when the file specification is entered by an interactive user or when a file specification includes a wildcard character or a search list logical name representing multiple files.

There is only one type of NAM block, and you usually associate only one NAM block with a file. For an extra level of file specification defaults, RMS applies defaults using additional NAM blocks that contain the file specifications of related files.

2
The RMS Program Interface

This section introduces the application program interface with RMS that is applicable to all OpenVMS languages in terms of the following:

- The run-time processing environment
- RMS symbol-naming conventions
- The calling sequence for record management services
- Allowable program execution modes
- Condition values returned by record management services

2.1 The RMS Run-Time Environment

The RMS run-time processing environment consists of a set of blocks and the run-time services. The control block fields accessed by each service specify the appropriate file and record operations. Depending on the operation, RMS uses one or more control blocks by referring to one or more fields as input to, or output from, the operation.

To use RMS, you must do the following:

1. Allocate the appropriate control block, usually at assembly time or compilation time. Control blocks must not reside in read-only storage and should be aligned on a longword boundary to maximize efficiency.
2. Insert the appropriate values into the control block fields before you invoke the related service.
3. Invoke the appropriate service. As part of this step, a condition value should always be examined.

To perform advanced RMS functions, you may need to set various control block field values at run time between the time the file is opened and when the appropriate service is invoked.

Note that OpenVMS languages perform some of these steps transparently when a particular language statement or macro is present in a source program.

Two fields in each control block—the block length (BLN) field and the block identifier (BID) (or block code [COD] field in a XAB)—define the length of the control block (in bytes) and identify the control block type, respectively. These internal fields are always used as input arguments by the service that accesses the control block, and must be set before the control block can be used. After the block length and block identifier fields are established, you must treat them as read-only fields until the control block is no longer needed.

Part II describes each control block field in detail, including its length and its symbolic name.

Appendix A lists the VAX MACRO calling format for each VAX MACRO control block macro.

Part III lists the calling format for each service together with the input control block fields and the output control block fields for each service.

2.2 Conventions for Naming Fields

RMS uses mnemonics to identify control block fields. For example, the mnemonic name for the FAB allocation quantity field is ALQ.

The mnemonic name (usually consisting of three characters) serves as a suffix to a symbolic name that identifies the location of each control block field. You should use the symbolic names to be sure you 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 the control block to the beginning of the field. These field names are thus called symbolic offsets.

Symbolic offset names are defined when the appropriate VAX MACRO control block initialization macro is used, when the appropriate VAX MACRO control block symbol definition macro is used, and when some languages invoke RMS. Alternatively, all control block symbolic offset names are available when you use the VAX MACRO $FABDEF, $RABDEF, $NAMDEF, and $XAB...DEF macros in a VAX MACRO program or procedure.

The format of the symbolic offsets consists of a 3-letter control block identifier (FAB, NAM, XAB, or RAB), a dollar sign ($), a 1-letter indicator of the length of the field (B, W, L, Q, or T), an underscore (_), and the field mnemonic, which is usually three letters.

The general format of the symbolic offset is shown in the following example:

```
ccc$x_fff
```

The components of the symbolic offset format are summarized in the following table.

Component	Length	Description
ccc	3 letters	Identifies the type of control block: FAB, NAM, XAB (for all XABs), or RAB.
$	1 character	Separates the control block identifier from the field length identifier; a dollar sign ($).
x	1 letter	Identifies the length of the field: B for byte, W for word, L for longword, Q for quadword, T for text buffer address. Symbolic length fields are identified by the letter S in this position. For example, the value field XAB$S_CACHE_TMO specifies the number of bytes allocated for defining the value of the cache timeout. See text for exceptions.
_	1 character	Separates the field length identifier from the field name; always an underscore (_).
fff	3 or more letters	Identifies the mnemonic name of the field, which is used in the VAX MACRO control block macro. Some mnemonics contain more than three letters; for example, symbolic offset XAB$B_PROLOG (from XABKEY).

For example, the FAB field whose mnemonic is ALQ has a length of one longword and is identified by the symbolic offset FAB$L_ALQ. The field NAM$L_RLF is a NAM longword field whose mnemonic RLF reflects its name, the related file field.

Exceptions to the length designation are NAMW_DID, NAMW_FID, XAB$W_RFI, and RAB$W_RFA, each of which is three words in length rather than one word.

The length of a T field is specified by the corresponding S field; for example, the length of the NAM$T_DVI field is specified by the symbolic value field named NAM$S_DVI.

When a control block field contains options identified by bits, each valid bit location has a symbolic offset name. Certain control block fields are binary options fields consisting of bit values. For these bits in a binary options field, the format of symbolic names resembles the format of the field names, except for the length indicator. Instead of identifying the field length, which is always one bit, the length field indicates whether a mask value (M) or bit offset (V) is defined by the symbolic name, as described in the following table.

Format	Description
xxx$M_fff	Indicates a mask value in a binary options field, typically where multiple bit options can be chosen. Used to set or clear bit values.
xxx$V_fff	Indicates the symbolic bit offset (number of bits from the beginning of the binary options field). Used to test bit values or to set bit values.

The *xxx* identifies the control block (FAB, NAM, XAB, or RAB); the *$* and _ are separator characters, and the *fff* defines the mnemonic for the bit option. For example, the option CTG in the FAB file-processing options (FOP) field has a symbolic bit offset of FAB$V_CTG and a mask value of FAB$M_CTG.

Constant (or keyword) fields can contain only a limited set of values, thus there are no mask values or symbolic bit offsets. In some instances, the letter K is used to denote a constant (keyword) value field in place of the letter C; otherwise, the naming convention is the same.

Unlike a binary options field, each possible value is identified by a symbolic constant value, in the following form:

```
xxx$C_fff
```

Note that the letter C replaces the letter M, denoting that this field is a constant (keyword) value field, not a mask value field. For example, the file organization (ORG) field of the FAB (FAB$B_ORG) can contain only the values FAB$C_IDX (indexed), FAB$C_REL (relative), or FAB$C_SEQ (sequential).

When specifying control block field locations, avoid using actual byte displacement values to identify control block field locations; instead, use the supplied symbolic offsets. RMS control block field locations may not always be the same from release to release; however, the symbolic offset names that identify the field locations always identify the same fields.

2.3 The RMS Calling Sequence

RMS uses the appropriate OpenVMS standard calling sequence and conventions, and preserves all general registers across a call, except for register 0 (R0) and register 1 (R1). When the service completes execution, it returns control to the calling program, passing a condition value in R0. You should analyze the completion value to determine the success or failure of the service and to alter the flow of execution of your program, if necessary. Where applicable, you should use the STS field and the STV field of the appropriate control block for signaling errors, instead of R0. For additional information about RMS completion values, see Section 2.4.

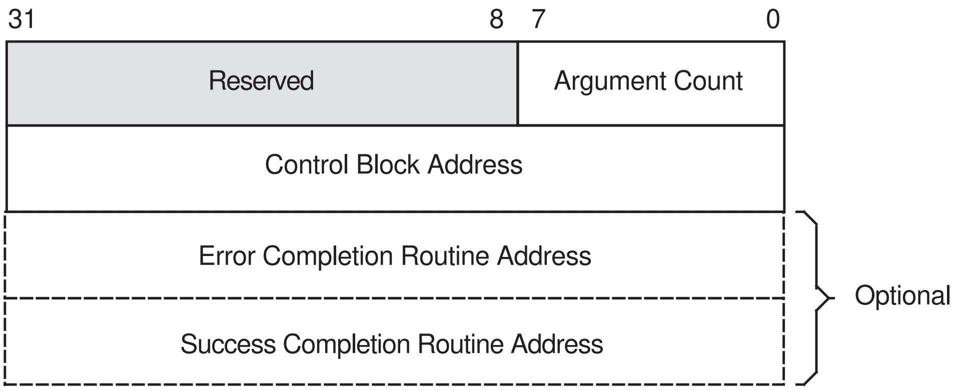
When calling a service, you must provide an argument list to specify the associated control block (FAB or RAB) and, optionally, any completion routines.

Note

When a service invokes an AST-level completion routine, it passes the address of the associated control block (FAB or RAB) as the AST argument value in the AST argument list.

The argument list sent to the service is from two to four longwords in length, as shown in Figure 2–1. (The Rename service, however, uses a 5-longword argument list.)

Figure 2–1 Argument List Format



ZK–0875–GE

RMS interprets the fields in the argument list as follows:

- The argument count field contains a binary value, from 1 through 3, representing the number of arguments in the argument list. For the Rename service only, set this value to 4.
- The control block address field contains the address of either the FAB (for file operations) or the RAB (for record operations).
- The error completion routine address field optionally contains the address of the entry mask of a user-written completion routine to be called if the requested operation fails. If used, the completion routine executes as an asynchronous system trap (AST).

- The success completion routine address field optionally contains the address of the entry mask of a user-written completion routine to be called if the requested operation completes successfully. If used, the completion routine executes as an asynchronous system trap (AST).
- The new FAB address field (not shown in Figure 2–1) contains the address of the FAB that contains the new file name for the Rename service. This field applies only to the Rename service.

2.4 Service Completion

This section describes various service completion scenarios. The events associated with completing an RMS service call depend to some extent on how the user calls the service. The user may specify either the synchronous or asynchronous option in the control block (FAB or RAB) passed to the service, and the user may or may not specify an AST. RMS returns the success status RMS$_SYNCH if the requested service completes its task immediately.

If the user specifies the asynchronous option, RMS returns control to the application program as soon as an I/O operation begins. If the user specifies synchronous I/O, RMS does not return control to the application program until the I/O operation terminates. If the user specifies an AST routine to the service, RMS does not return control to the application program until the AST routine terminates.

In all cases, the called service clears the completion status fields (STS and STSV where applicable). If the user specifies the synchronous option, the service returns the completion status value in R0 as well as in STS. If the user specifies asynchronous I/O, the service returns an I/O pending code in R0 and waits until the I/O operation terminates before returning completion status in STS.

Note

The user may also specify an option that switches from asynchronous I/O to synchronous I/O when conditions permit immediate I/O operations.

In general, you may receive one of many error or success codes from an operation. The discussion of each service in Part III includes a list of the possible condition values that you can receive when you invoke the service.

Note

$RMSDEF gives you the names of the condition values returned by RMS. See the OpenVMS system messages documentation for a list of all RMS status codes.

You should test for success by checking only the low-order bit of the status code for a true condition (bit set). The three low-order bits returned in the status code indicate the severity of the code. The severity codes are as follows:

001 (1)	Success (low-order bit set).
011 (3)	Information (low-order bit set).

000 (0)	Warning; indicates a nonstandard condition. The operation may have performed some, but not all, of the requested function.
010 (2)	Error; you must recognize that a problem exists and provide a contingency plan in your program for such a condition.
100 (4)	Severe error; normally caused by program logic errors or other unrecoverable conditions.

The usual method of testing the completion status is to examine register 0 for success, failure, or specific completion values. For certain completion values, RMS returns additional information in the status value field (STV) of the control block. The description of the codes presented in the OpenVMS system messages documentation indicates the instances when the STV contains such information.

The STS and STV fields should be used to signal RMS errors to ensure that the error message includes all relevant information. 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. (Consult Table 1–1 if you are not sure of the group to which a particular service belongs.)

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 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 *OpenVMS RTL Library (LIB$) Manual*.

RMS services are considered system services for the purpose of generating system service exceptions on errors. You can choose whether to test and handle errors in your program or set the system service failure exception mode (using the Set System Failure Exception Mode system service, SYS$SETSFM) to have failures automatically signaled. For most applications, especially if a high-level language is used, testing and handling errors in your program is the preferred method. If you test for error conditions in your program, you should be sure to disable any unwanted system service exception generation.

Note that if the FAB or RAB is invalid or inaccessible, 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 R0 (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$_FAB	FAB not writable or invalid block ID field
RMS$_RAB	RAB not writable 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 can only be detected and signaled using R0, but are rare and, if they occur at all, would most likely occur during initial program debugging and testing. Therefore, you must examine the completion value in R0 (instead of the STS field of the FAB or RAB) for the completion codes. If necessary, your program could test for these errors and, if encountered, indicate the completion values using R0 instead of the STS and STV fields.

RMS completion status error codes generally fall into one of four groups:

- Programming errors
- Program design errors
- System environment errors
- Operator/user errors

Programming Errors

These errors are caused by incorrect programming and are usually detected during the early stages of developing and debugging a program that uses RMS. Typical examples are missing values for required fields, referring to a RAB instead of a FAB, invalid address for a buffer, and so forth. This type of error is generally self-explanatory and usually requires only a minor change to the program.

Program Design Errors

These errors are caused by more subtle errors that may rarely occur, particularly if asynchronous record I/O, multistreamed sharing, or shared files are involved. These errors may occur long after a program has been in use and could require either a major program revision or the addition of substantial error recovery code to handle the error conditions. Record-lock errors, resource-exhaustion errors, and record-stream-currently-active errors are typical program design errors.

System Environment Errors

These errors include hardware errors and RMS or other system software errors that are not caused by your program. You may need to add substantial defensive error-handling code or you may be able to run the program again without error.

Operator/User Errors

These errors include errors by the user of the program, such as not mounting a device before running the program, or typing an invalid file specification. As with system environment errors, you may need to add substantial defensive error-handling code or simply reprompt the user for the correct information or user action.

There are conditions in which completion status codes may not be returned as expected:

- The completion status codes that apply to the Close service do not include errors introduced by the FAB$V_SCF option and the FAB$V_SPL option. If the request is serviced successfully, then a success completion code is returned, even if the request is found to be in error by the job controller process.
- The Wait service has unique errors. This service can return any status code of the awaited operation.
- Errors associated with output operations may not necessarily be reported as the status of a particular operation because modified I/O buffers are not always written out immediately. Such errors are reported as the status of a subsequent operation, which may be an input, output, or control operation.

When you submit an SPR, you should also provide a magnetic tape copy of the file causing the error.

When using the debugger, use the debug command EXAMINE/CONDITION to view the message corresponding to the value in R0 or the STS field or the STV field. For example, you can view the error codes in the STS and STV fields of the FAB at symbolic address (label) MYFAB when debugging a VAX MACRO program by entering the following commands:

```
DBG> EXAMINE/CONDITION MYFAB+FAB$L_STS
DBG> EXAMINE/CONDITION MYFAB+FAB$L_STV
```

For additional information about the debugger, see the *OpenVMS Debugger Manual*.

2.5 Allowable Program Execution Modes

Generally, RMS executes in either executive mode or executive AST mode. When an operation is initiated, processing begins in executive mode. If device I/O is necessary to process the request, the $QIO system service is called. RMS specifies an executive-mode AST to signal completion. At this point, RMS exits from executive mode. If the operation is being performed asynchronously, control is returned to the caller; if the operation is synchronous, RMS waits for an event flag in the access mode of the caller. When the I/O is complete, RMS continues processing in executive AST mode. Thus, user-mode ASTs can be serviced while a synchronous operation called from user mode is awaiting I/O completion. However, processing in user mode during an asynchronous operation is interrupted by RMS processing in executive AST mode when I/O completes.

RMS should not be called from kernel mode, from executive AST mode, or from executive mode when executive-mode ASTs are disabled.

2.6 Access-Mode Protected Memory

RMS protects the following data structures and their associated I/O buffers at EW (execute read/write):

- RMS-controlled data structures
- Process-permanent data structures
- Image-activated data structures

Previously, the data structures were protected at UREW (user read, executive write).

The following memory protection exceptions apply to USER-mode accessors of RMS and are protected at UREW:

- Internal RMS I/O buffers to facilitate RAB$V_LOC mode
- RMS buffers containing collated tables used for indexed files

2.7 Reserved Event Flags

RMS uses system-reserved event flags to synchronize its internal operations. RMS reserves event flags 27, 28, 29, and 30 for possible use; in addition, event flag 31 is used to specify a ‘‘do not care’’ event flag for asynchronous processing.

2.8 DEC Multinational Character Set

You can use any character in the DEC Multinational character set in RMS records, including the key value of an indexed file. Keys are collated according to their corresponding character code value.

For a list of characters in the DEC Multinational character set, see the *OpenVMS User's Manual*.

3

Implementing RMS from C Programs

This section includes C programming examples illustrating the implementation of RMS from a high-level programming interface. Each of the sample programs illustrates the implementation of a particular programming task in the OpenVMS environment.

From a high-level language program, you can create new files, process existing files, extend and delete files, and read, write, update, and delete records within files in an RMS environment.

To create and process RMS files, your program must contain calls to appropriate record management services from your language interface with RMS. Generally, you make these calls by using the service macros for run-time processing. When encountered at run time, the expanded code of these macros generates a call to the corresponding service. Each call represents a program request for a file service, a record service, or a block I/O transfer operation.

3.1 Creating, Accessing, and Deaccessing a File

You can create, access, and deaccess a file using either the Create service or the Open service. The Create service constructs a new file structured to the attributes you specify in the FAB for the file, whereas the Open service makes an existing file available for processing by your program. Both of these services allocate the system resources needed to establish an access path to a file.
You must open or create a file to perform most file operations and any record operations on that file. Where applicable, you must declare the type of shared access when you create or open a file. You do this through your program interface with RMS by selecting file access control options.

RMS provides several file-processing options for the Create service. The *create-if* option requests that the file be created only if it does not exist in the specified directory. If the file does exist in the specified directory, the existing file is opened. The Open and Create services both establish access to the desired file, but the Create service also allocates disk space and performs the functions related to allocation.

When you are finished processing a file, you invoke the Close service to close the file, disconnect all record streams associated with the file, and free all resources allocated to the file. If you do not explicitly invoke the Close service when the program image exits, RMS attempts an implicit close. All resources associated with open files are returned when the files are deaccessed at image rundown time. However, process permanent files are not implicitly closed when an image exits.

3.1.1 Example of Copying Records from One File to Another File

Example 3–1 illustrates the use of various services to access and copy records from one file to another.

Example 3–1 Use of the Create, Open, and Close Services

```
/*
** COPYFILE.C     This program copies the input file to the output file.
** It is made to resemble the MACRO example in the RMS Reference Manual.
*/
#define REC_SIZE 132
#define INPUT_NAME "INFILE"
#define OUTPUT_NAME "OUTFILE"
#define DEFAULT_NAME ".DAT"

#include <rms>  /* defines for rabs and fabs */
#include <stdio>  /* defins printf...  */
#include <starlet>  /* defines sys$open et al  */

COPYFILE ()
{
struct FAB infab, outfab, *fab; /* Allocate fabs and a pointer to fab */
struct RAB inrab, outrab, *rab; /* Allocate fabs and a pointer to fab */
int  lib$signal();
int  stat;
char  rec_buff[REC_SIZE]; /* maximum record size        */

infab = cc$rms_fab;   /* Make this a real FAB (bid and bln) */
infab.fab$l_fna = (char *) &INPUT_NAME; /* Primary file name: (logical) name..*/
infab.fab$b_fns = sizeof INPUT_NAME -1; /* .. and its size        */
infab.fab$l_dna = (char *) &DEFAULT_NAME; /* Default name: here file type..     */
infab.fab$b_dns = sizeof DEFAULT_NAME -1; /* .. and its size        */

inrab = cc$rms_rab;   /* Make this a real RAB (bid and bln) */
inrab.rab$l_fab = &infab;  /* Point to FAB for $CONNECT       */
inrab.rab$v_rah = 1;   /* Set bitVield to request read-ahead */
inrab.rab$l_ubf = rec_buff;  /* Point to buffer area..       */
inrab.rab$w_usz = REC_SIZE;  /* and indicate its size       */

outfab = cc$rms_fab;   /* Make this a real FAB (bid and bln) */
outfab.fab$v_ctg = 1;   /* Allocate contigeously       */
outfab.fab$v_put = 1;   /* Write access (default on create)   */
outfab.fab$v_nil = 1;   /* No sharing (default on create)     */
outfab.fab$b_rat = FAB$M_CR;  /* Set option using bitMask       */
outfab.fab$w_mrs = REC_SIZE;
outfab.fab$l_fna = (char *) &OUTPUT_NAME;
outfab.fab$b_fns = sizeof OUTPUT_NAME -1;
outfab.fab$l_dna = (char *) &DEFAULT_NAME;
outfab.fab$b_dns = sizeof DEFAULT_NAME -1;

outrab = cc$rms_rab;
outrab.rab$l_fab = &outfab;
outrab.rab$v_wbh = 1;   /* Write Ahead         */
outrab.rab$l_rbf = rec_buff;  /* Same buffer address as before      */

fab = &infab;    /* for error handling      */
stat = sys$open ( fab );  /* Actual open (could use &infab)   */
if (stat & 1)    /* $OPEN Success ? */
{
    outfab.fab$l_alq = infab.fab$l_alq; /* Set proper size for output     */
    fab = &outfab;   /* for error handling      */
    stat = sys$create ( fab );  /* Try to create the file     */
}
```

(continued on next page)

Example 3–1 (Cont.) Use of the Create, Open, and Close Services

```
if (stat & 1)    /* Both open & create success ?     */
{
    rab = &outrab;   /* for error handling      */
    stat = sys$connect ( rab );  /* get some rms internal buffers    */
    if (stat & 1)   /* output $CONNECT Success ?     */
    {
 rab = &inrab;   /* for error handling      */
 stat = sys$connect ( rab ); /* input $CONNECT Success ?     */
    }
    if (stat & 1)   /* CONNECTs succes? then prime loop */
 stat = sys$get ( rab );  /* setting stat for while     */

    while (stat & 1)   /* success on record operation ? */
    {
 /*
 ** Main Code. Opened and connected files and buffer
 ** First $GET done and inrab is current. Copy records.
 */
 outrab.rab$w_rsz = inrab.rab$w_rsz; /* set correct size */
 rab = &outrab;    /* error handler */
 stat = sys$put ( rab );   /* Actual copy  */
 if (stat & 1)    /* $PUT success? */
 {
     rab = &inrab;   /* for error handling   */
     stat = sys$get ( rab );  /* $GET next , set stat */
 }
    } /* while */

    /*
    ** Fallen through while. stat must be EOF if copy was succesful.
    ** if not, signal error from connect, get or put. Using stat instead
    ** of rab->rab$l_sts to handle (programming) error providing RAB.
    */

    if (stat != RMS$_EOF)
 stat = lib$signal( stat, rab->rab$l_stv );

    stat = sys$close ( &infab );
    stat = sys$close ( &outfab );
    }
else
{
    /* Failed to open input or output file */
    stat = lib$signal( stat, fab->fab$l_stv );
}

return stat;   /* Using output close stat to return */
}
```

This example illustrates how you can use the sequential file organization to create a new file by copying records from an existing file. The newly created file and the source file have variable-length records.

This example assumes that an external program has identified the input file as a search list logical name using the equivalent of the following DCL command:

```
$ ASSIGN [INV]30JUN93,[INV.OLD]30JUN93 INFILE
```

This command directs RMS to look for the input file in directory [INV] first, and, if it does not find the file, to look in directory [INV.OLD].

The program also specifies the default file type .INV for the input file using this statement:

```
infab.fab$l_dna = &DEFAULT_NAME; /* Default name: here file type..     */
infab.fab$b_dns = sizeof DEFAULT_NAME; /* .. and its size        */
```

Next the program configures the RAB used for the input file. The first argument links the RAB to the associated FAB. This is the only required argument to a RAB. The rest of the arguments specify the read-ahead option (described in later text) and the record buffer for the input file. The Get service uses the user record buffer address (UBF) field and the user record buffer size (USZ) field as inputs to specify the record buffer and the record size, respectively.

Note

When you invoke the GET service, RMS takes control of the record buffer and may modify it. RMS returns the record size and only guarantees the contents from where it accessed the record to the completion of the record.

The program then configures the FAB for the output file. The first argument equates the file name to the externally defined logical name. After the program specifies the default file type for the output file, it specifies three additional FAB fields.

First, it specifies that RMS should allocate contiguous space for the output file by setting the CTG bit in the FAB$L_FOP field of the FAB.

Next, the program uses a program-defined variable to store the value 132 in the MRS field:

```
#define REC_SIZE 132
outfab.fab$w_mrs = REC_SIZE;
```

The program then specifies that each record is to be preceded by a line feed and followed by a carriage return whenever the record is output to a line printer or terminal:

```
outfab.fab$b_rat = FAB$M_CR;  /* Set option using bitMask       */
```

Because the program alternately reads and then writes each record, the input file and the output file may share the same buffer. However, because the Put service does not have access to the UBF and UBZ fields, the output RAB defines the buffer using the RBF and the RSZ fields.

Note that the UBF, USZ, and RBF values are set prior to run time, but that the RSZ value is set at run time, just prior to invocation of the Put service. This is done because the input file contains variable-length records and the Put service relies on the Get service to supply each record's size by way of the RSZ field, an INRAB output field.

The following statement from the sample program illustrates this feature:

```
outrab.rab$w_rsz = inrab.rab$w_rsz; /* set correct size */
```

The run-time processing macros for the input file consist of a $OPEN, a $CONNECT, a $GET, and a $CLOSE macro. Because the input file already exists, the program accesses it with a $OPEN macro. The sole argument to this macro identifies the FAB to the Open service:

```
stat = SYS$OPEN ( fab );  /* Actual open (could use &infab)   */
```

Next, the program connects a record stream to the input file by calling the Connect service and specifying INRAB as the appropriate RAB:

```
stat = SYS$CONNECT ( rab );  /* get some rms internal buffers    */
```

Note that upon completion of each service call, the program tests the condition value in STAT returned by the service before proceeding to the next call. If the call fails, the program exits with the appropriate control block address in *rab* or *fab*.

After creating the output file and establishing its record stream, the program begins a processing loop in which the Get service reads a record from the input file and the Put service writes the record to the output file. When all file records are copied, as indicated by the detection of the end of the file, the main *while* loop terminates.

The Close service disconnects the record stream for all RABs connected to the specified FAB. In a multistream environment (more than one RAB can be connected to a single FAB), a program may disconnect individual record streams using the Disconnect service.

3.2 Program to Illustrate Record Operations

The record-processing services provide the following record operations:

- Get
- Find
- Put
- Update
- Delete

This section illustrates the use of RMS record operations from a C program as shown in Example 3–2.

Example 3–2 Record Operations

```
/*
  RECORD OPERATIONS

 This program looks up records by key and then

    *  copies the record to an output file,
    *  deletes the record, or
    *  updates the record
*/

#include <rms>  /* defines for rabs and fabs */
#include <stdio>  /* defins printf...  */
#include <string> /* defines strlen  */
#include <stdlib> /* defines exit   */
#include <starlet>  /* defines sys$open et al  */

#define INPUT_NAME "INFILE:"
#define OUTPUT_NAME "OUTFILE:"
#define RECORD_SIZE 132
#define TYPING_SIZE 100

struct FAB infab, outfab;
struct RAB inrab, outrab;
```

(continued on next page)

Example 3–2 (Cont.) Record Operations

```
error_exit (code, value)
long code;
long value;
{
 void lib$signal();
 lib$signal (code, value);
 exit (0);
}
main ()
{
 char record [RECORD_SIZE];  /* record buffer */
 char key [RECORD_SIZE];  /* key buffer */
 char choice [TYPING_SIZE];  /* typing buffer */
 long status;

 /* set up input fab */
 infab = cc$rms_fab;
 infab.fab$b_fac = FAB$M_GET | FAB$M_PUT | FAB$M_UPD | FAB$M_DEL;
 infab.fab$b_shr = FAB$M_SHRGET | FAB$M_SHRPUT | FAB$M_SHRUPD
      | FAB$M_SHRDEL; /* read/write sharing */
 infab.fab$l_fna = (char *) &INPUT_NAME; /* logical name INFILE */
 infab.fab$b_fns = sizeof INPUT_NAME - 1;

 /* set up output fab */
 outfab = cc$rms_fab;
 outfab.fab$b_fac = FAB$M_PUT;
 outfab.fab$l_fna = (char *) &OUTPUT_NAME; /* logical name OUTFILE */
 outfab.fab$b_fns = sizeof OUTPUT_NAME - 1;
 outfab.fab$w_mrs = RECORD_SIZE;  /* record size */
 outfab.fab$b_org = FAB$C_REL;  /* relative file */
 outfab.fab$b_rat = FAB$M_CR;  /* implied carriage return */

 /* set up input rab */
 inrab = cc$rms_rab;
 inrab.rab$l_fab = &infab;
 inrab.rab$b_rac = RAB$C_KEY;  /* key access */
 inrab.rab$b_krf = 0;   /* access by primary key */
 inrab.rab$l_kbf = key;   /* key buffer */
 inrab.rab$l_ubf = record;  /* record buffer */
 inrab.rab$w_usz = RECORD_SIZE;  /* maximum record size */

 /* set up output rab */
 outrab = cc$rms_rab;
 outrab.rab$l_fab = &outfab;
 outrab.rab$l_rbf = record;  /* record buffer */

 /* open files and connect streams */
 status = sys$open (&infab);
 if (! (status & 1))
  error_exit (status, infab.fab$l_stv);
 status = sys$connect (&inrab);
 if (! (status & 1))
  error_exit (status, inrab.rab$l_stv);
 status = sys$create (&outfab);
 if (! (status & 1))
  error_exit (status, outfab.fab$l_stv);
 status = sys$connect (&outrab);
 if (! (status & 1))
  error_exit (status, outrab.rab$l_stv);
```

(continued on next page)

Example 3–2 (Cont.) Record Operations

```
while (1)
{
 /* get a key and a record */
 printf ("Please input key value: ");
 gets (key);   /* get key from user */
 if (feof (stdin))  /* stop on ctrl-Z */
  break;
 inrab.rab$b_ksz = strlen (key); /* set key length */
 status = sys$get (&inrab);
 if (! (status & 1))
  error_exit (status, inrab.rab$l_stv);

 /* display the record */
 record[inrab.rab$w_rsz] = '\0';
 printf ("Record: {%s}\n", record);

 /* choose what to do */
 printf ("Please choose C(opy), D(elete), or U(pdate):");
 gets (choice);   /* get choice from user */
 if (feof (stdin))  /* stop on ctrl-Z */
  break;
 switch (choice[0])
 {
  case 'c':
  case 'C':
   /* copy the record */
   outrab.rab$w_rsz = inrab.rab$w_rsz;
     /* out length = in length */
   status = sys$put (&outrab);
   if (! (status & 1))
    error_exit (status, outrab.rab$l_stv);
   break;
  case 'd':
  case 'D':
   /* delete */
   status = sys$delete (&inrab);
   if (! (status & 1))
    error_exit (status, inrab.rab$l_stv);
   break;
  case 'u':
  case 'U':
   /* get a new record */
   printf ("Please input record value: ");
   gets (record); /* get record from user */
   inrab.rab$w_rsz = strlen (record);
     /* set record length */
   status = sys$update (&inrab);
   if (! (status & 1))
    error_exit (status, inrab.rab$l_stv);
   break;
  default:
   /* do nothing */
   break;
 }
}
```

(continued on next page)

Example 3–2 (Cont.) Record Operations

```
 /* close files */
 status = sys$close (&infab);
 if (! (status & 1))
  error_exit (status, infab.fab$l_stv);
 status = sys$close (&outfab);
 if (! (status & 1))
  error_exit (status, outfab.fab$l_stv);
}
```

The program requires access to RMS.H and STDIO.H in SYS$LIBRARY to provide RMS structure definitions and standard C input/output. The program sets up four FABs: a FAB and a RAB for the input file and a FAB and a RAB for the output file. The program then opens and connects the input file and creates and connects the output file.

The main loop prompts the user for a key and then retrieves the record with that key. It then prompts the user for a decision of what to do with the record:

- Copy the record to the output file
- Delete the record
- Update the record using new user data

The program handles all errors by signaling the error and then exiting.

3.3 Program to Show Index Root Levels

Example 3–3 shows the index root level(s) for a specified file. You can modify the program to display more parameters, add LIB$FIND_FILE, and so forth.

To use the program, define an external DCL command and pass the filespec to the program as a parameter.

Example 3–3 Displaying the Index Root for a File

```
** Show_roots.c
**
*/

#include <rms>
#include <stdio>
#define MAXKEY 10
main (int argc, char *argv[])
{
struct FAB fab;
struct XABSUM sum;
struct XABKEY xab[MAXKEY];
int  i, stat, lvl, keys;
```

(continued on next page)

Example 3–3 (Cont.) Displaying the Index Root for a File

```
fab = cc$rms_fab;
sum = cc$rms_xabsum;
fab.fab$b_shr = FAB$M_SHRPUT;
fab.fab$b_fac = FAB$M_GET;
fab.fab$l_fna = argv[1];
fab.fab$b_fns = STRLEN( argv[1] );
fab.fab$l_xab = &sum;
stat = SYS$OPEN ( &fab );
if (!(stat&1)) return stat;
if (fab.fab$b_org!=FAB$C_IDX) return RMS$_ORG;
keys = sum.xab$b_nok;
fab.fab$l_xab = &xab[0];
for (i=0; i<keys; i++)
    {
    /*
    ** Init Xab Key for each defined key
    ** Point previous to current except first.
    */
    xab[i] = cc$rms_xabkey;
    xab[i].xab$b_ref = i;
    if (i) xab[i-1].xab$l_nxt = &xab[i];
    }
/*
** Ask RMS to fill in the XABs hooked off the FAB.
*/
stat = SYS$DISPLAY ( &fab );
if (!(stat&1)) return stat;
printf ("File %s, Root levels: %d", argv[1], xab[0].xab$b_lvl);
for (i=1; i<keys; i++) printf (", %d", xab[i].xab$b_lvl);
printf (".\n");
return stat;
}
```

Part II

RMS Control Blocks

Part II describes each RMS control block, including a complete listing and description of each field.

4
File Access Block (FAB)

The file access block (FAB) defines file characteristics, file access, and certain run-time options. It also indicates whether other control blocks are associated with the file.

4.1 Summary of Fields

Many FAB fields are directly equivalent to certain File Definition Language (FDL) attributes. For information about FDL, refer to the *OpenVMS Record Management Utilities Reference Manual*.

The symbolic offset, the size, the FDL equivalent, and a brief description of each FAB field are presented in Table 4–1.

Table 4–1 FAB Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
FAB$B_ACMODES	1	None	File access modes
FAB$L_ALQ	4	FILE ALLOCATION	Allocation quantity (blocks)
FAB$B_BID[1]	1	None	Block identifier
FAB$B_BKS	1	FILE BUCKET_SIZE	Bucket size
FAB$B_BLN[1]	1	None	Block length
FAB$W_BLS	2	FILE MT_BLOCK_SIZE	Magnetic tape block size
FAB$V_CHAN_MODE[2]	–	None	Channel access mode protection
FAB$L_CTX	4	FILE CONTEXT	Context
FAB$W_DEQ	2	FILE EXTENSION	Default file extension quantity
FAB$L_DEV[3]	4	None	Device characteristics
FAB$L_DNA	4	FILE DEFAULT_NAME	Default file specification string address
FAB$B_DNS	1	FILE DEFAULT_NAME	Default file specification string size
FAB$B_FAC	1	ACCESS[3]	File access

[1]This field is statically initialized by the $FAB macro to identify this control block as a FAB.

[2]This is a 2-bit subfield.

[3]This field contains options; corresponding FDL equivalents are listed in the description of the field.

(continued on next page)

Table 4–1 (Cont.) FAB Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
FAB$L_FNA	4	FILE NAME	File specification string address
FAB$B_FNS	1	FILE NAME	File specification string size
FAB$L_FOP	4	FILE[3]	File-processing options
FAB$B_FSZ	1	RECORD CONTROL_FIELD_SIZE	Fixed-length control area size
FAB$W_GBC	2	FILE GLOBAL_BUFFER_COUNT	Global buffer count
FAB$W_IFI[4]	2	None	Internal file identifier
FAB$B_JOURNAL	1	None	Journal flags status
FAB$V_LNM_MODE	-[2]	None	Logical name translation access mode
FAB$L_MRN	4	FILE MAX_RECORD_NUMBER	Maximum record number
FAB$W_MRS	2	RECORD SIZE	Maximum record size
FAB$L_NAM	4	None	Name block address
FAB$B_ORG	1	FILE ORGANIZATION	File organization
FAB$B_RAT	1	RECORD[3]	Record attributes
FAB$B_RFM	1	RECORD FORMAT	Record format
FAB$B_RTV	1	FILE WINDOW_SIZE	Retrieval window size
FAB$L_SDC[4]	4	None	Secondary device characteristics
FAB$B_SHR	1	SHARING[3]	File sharing
FAB$L_STS[4]	4	None	Completion status code
FAB$L_STV[4]	4	None	Status values
FAB$L_XAB	4	None	Extended attribute block address

[2]This is a 2-bit subfield.

[3]This field contains options; corresponding FDL equivalents are listed in the description of the field.

[4]This field cannot be initialized by the $FAB macro.

Each FAB field is described in this section. Unless indicated otherwise, each field is supported for DECnet for OpenVMS operations on files at the remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

To use a FAB, you must allocate process storage and specify the character string for the primary file specification and, optionally, the default file specification. The FAB$L_FNA and FAB$B_FNS fields define the primary file specification to RMS; the FAB$L_DNA and FAB$B_DNS fields define the default file specification to RMS.

The format and arguments of the $FAB macro and the $FAB_STORE macro are defined in Appendix A.

4.2 FAB$B_ACMODES Field

This field comprises four 2-bit subfields, two of which are unsupported. The supported subfields are the FAB$V_CHAN_MODE subfield and the FAB$V_LNM_MODE subfield (see Section 4.22).

4.3 FAB$L_ALQ Field

The allocation quantity (ALQ) field defines the number of blocks to be initially allocated to a disk file when it is created (using the Create service) or to be added to the file when it is explicitly extended (using the Extend service). This field corresponds to the FDL attribute FILE ALLOCATION.

The field takes numeric values in the range of 0 through 4,294,967,295, although the maximum value depends on the number of blocks available on the disk. There are several rules concerning the use of the value 0:

- If you specify 0 blocks when you create a sequential file, RMS allocates no initial space for the file.
- If you specify 0 blocks for relative or indexed files, RMS allocates the minimum number of blocks needed to support the file organization. For example, if you specify 0 blocks when you create an indexed file, RMS allocates the number of blocks necessary to contain specified key and area definitions as the initial extent.
- A value of 0 blocks is meaningless when you intend to extend a file.

When RMS opens an existing file, it puts the highest virtual block number currently allocated to the file in the FAB$L_ALQ field.

For the Extend service, this field specifies the number of blocks to be added to the file. Note that RMS uses this as the extension value when a process extends a file using the Extend service, unless the process changes the value before it invokes the Extend service.

When you use the Create and Extend services, the allocation quantity value is rounded up to the next disk cluster boundary; the number of blocks actually allocated is returned in the FAB$L_ALQ field.

Note that the function of the FAB$L_ALQ field changes if allocation control XABs are used when you create or extend a file. The description of the XABALL control block (see Chapter 8) discusses how allocation control XABs affect the FAB$L_ALQ field.

4.4 FAB$B_BID Field

The block identifier (BID) field is a static field that identifies a control block as a FAB. Once set, this field should not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value FAB$C_BID (this is done by the $FAB macro).

4.5 FAB$B_BKS Field

The bucket size (BKS) field is used only for relative or indexed files to specify the number of blocks in each bucket of the file.

This field contains a numeric value in the range of 0 to 63. If you do not specify a value or specify a value of 0, RMS uses a default of the minimum number of blocks needed to contain a single record, or a minimum of two records for indexed files. If the file will be processed by RMS–11, the bucket size must be less than or equal to 32 blocks.

When calculating the bucket size, you must consider the control information (overhead) associated with each bucket. Also, certain record types contain control bytes; thus, the number of records per bucket multiplied by the number of control bytes per record equals the number of record overhead bytes per bucket. See the *Guide to OpenVMS File Applications* for more information.

Before specifying a bucket size, you must be aware of the relationship between bucket size and record size. You must also consider any record control bytes (overhead) required for the type of record chosen. Because RMS does not allow records to cross bucket boundaries, you must ensure that the number of blocks per bucket conforms to formulas designed to handle one of the following:

- Relative files with fixed-length records
- Relative files with variable-length records
- Relative files with VFC (variable with fixed control) records
- Indexed files with fixed-length records
- Indexed files with variable-length records

You can use the Edit/FDL utility to determine the optimum bucket size. Note that if an allocation control XAB is specified, the value specified in the XAB$B_BKZ field supersedes the value specified in the FAB$B_BKS field. When multiple allocation control XABs are specified, the largest value in any XAB$B_BKZ field supersedes the value in the FAB$B_BKS field. Refer to Section 8.6 for information about the XAB$B_BKZ field.

When you open an existing relative or indexed file, RMS sets the FAB$B_BKS field to the defined size of the largest bucket size in the file. However, when you create a new relative or indexed file, set the FAB$B_BKS field before you invoke the Create service rather than use the default.

With indexed files, note that if the FAB$B_BKS field is not specified and a maximum record size (FAB$W_MRS field) is specified, then RMS selects a bucket size that allows at least one maximum size record to fit. Generally, performance for record insertion and sequential retrieval on the primary key improves if at least six or seven data records fit into a primary data bucket. If either the bucket size or the disk cluster size is other than one block, use a default extension quantity (FAB$W_DEQ) that is the least common multiple of the bucket size and cluster size to avoid allocated but unused blocks within the file.

This field corresponds to the FDL attribute FILE BUCKET_SIZE.

4.6 FAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the FAB. Once set, this field should not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value FAB$C_BLN (this is done by the $FAB macro).

4.7 FAB$W_BLS Field

RMS uses the block size (BLS) field as input for nondisk files. The BLS field usually defines the size, in bytes, of the blocks on a magnetic tape. Note that for some devices, this value must be an even number. This field corresponds to the FDL attribute FILE MT_BLOCK_SIZE.

The FAB$W_BLS field contains a numeric value in the range of 20 through 65,535 for ANSI-formatted tapes and 14 through 65,532 for foreign tapes. (Foreign tapes are those that are not in the standard ANSI format used by OpenVMS operating systems, and must be mounted using the DCL command MOUNT/FOREIGN.) If no value or a value of 0 is specified, the default selected when the volume was mounted is used.

When you create a magnetic tape file, you can set the FAB$W_BLS field before you invoke the Create service. In all other cases, RMS ignores this field. When you open an existing sequential file with an Open service, RMS returns the device buffer size. For terminals, this field defines the WIDTH setting; for mailboxes, this field defines the maximum message size.

For compatibility with RMS–11, RMS rounds off the block size for an ANSI-formatted tape to the next highest multiple of 4. For example, if you set the block length to 38, RMS sets it to 40. The block size of a foreign tape is not rounded off by RMS.

To create a magnetic tape for interchange with Digital operating systems other than OpenVMS systems, consult the documentation for the recipient system to identify possible limitations on block size. ANSI standards require that the block size be less than or equal to 2048 bytes.

4.8 FAB$V_CHAN_MODE Subfield

The channel access mode protection (FAB$V_CHAN_MODE) subfield is part of the FAB$B_ACMODES field. The 2-bit FAB$V_CHAN_MODE subfield provides one of two functions:

- For standard processing, where neither the FAB$V_NFS nor FAB$V_UFO option is selected in the FAB$L_FOP field, the user program can use the FAB$V_CHAN_MODE subfield to override the access-mode protected files function.
- When the user program specifies a nonstandard processing option by selecting either non-file-structured operations (FAB$V_NFS) or a user-file-open operation (FAB$V_UFO), RMS uses the FAB$V_CHAN_MODE subfield to assign the calling program's access mode to the I/O channel.

4.8.1 Override Value

The access-mode protected files function ensures that accessors who are operating in an outer access mode cannot access files opened or created by inner-mode accessors. Access-mode protection includes memory and data structures that are interrelated to access-mode protected files.

When the user program seeks to override access-mode protection, it must specify the value PSL$C_USER in the FAB$V_FAB$V_CHAN_MODE subfield.

4.8.2 Channel Access Mode Function

When used to specify the channel access mode, the FAB$V_CHAN_MODE subfield can contain one of the following values, with the related constant value for each shown in parentheses:

0	None
1	Executive mode (PSL$C_EXEC)
2	Supervisor mode (PSL$C_SUPER)
3	User mode (PSL$C_USER)

The default value is 0 (none), which is interpreted by RMS as executive mode.

If the access mode requested is more privileged than the access mode of the calling process, RMS uses the access mode of the caller and does not return an error.

There is no corresponding FDL equivalent for this field. The FAB$V_CHAN_MODE subfield is used locally for channel to DECnet for OpenVMS communications but is ignored on the remote system.

To set this field from MACRO level, you include the appropriate expression as an argument to the $FAB macro. For example, to specify supervisor channel access mode, you might include a statement in this format:

```
$FAB    ...,CHAN_MODE = PSL$C_SUPER,...
```

If you are using a high-level language, refer to your documentation as to how (and whether) you can directly access RMS fields and then incorporate the appropriate channel access mode expression into the appropriate language statement.

4.9 FAB$L_CTX Field

The user context (CTX) field allows you to convey user information to a completion routine in your program. This field contains a user-specified value, up to four bytes long, and is intended solely for your use. RMS never uses it for record management activities.

This field corresponds to the FDL attribute FILE CONTEXT.

4.10 FAB$W_DEQ Field

The default file extension quantity (DEQ) field specifies the number of blocks to be added when RMS automatically extends the file. Automatic extension only applies to files that reside on disk devices and occurs whenever your process invokes a Put or Write service and the currently allocated file space is exhausted. When you invoke a Put service, automatic file extension occurs when needed, regardless of the file organization. When you invoke the Write service, automatic extension occurs only for sequential files (indexed and relative files require the Extend service to extend file allocation).

This field corresponds to the FDL attribute FILE EXTENSION.

This field contains a numeric value in the range 0 through 65,535, which is rounded up to the next cluster boundary. A large value results in fewer file extensions over the life of a file; a small value results in numerous file extensions over the life of a file. When a file has numerous file extensions that may be noncontiguous, this slows record access.

If you do not specify a value or specify the value 0 when you create a file, RMS uses the default specified by the DCL command SET RMS_DEFAULT/EXTEND_QUANTITY. If this value is 0, RMS uses the system default extension quantity specified by the system parameter RMS_EXTEND_SIZE. If this value is 0, RMS computes the default value.

If the value in the FAB$W_DEQ field, the value set by the SET RMS_DEFAULT/EXTEND_QUANTITY command, and the value of the system parameter RMS_EXTEND_SIZE are *all* 0, RMS may provide an overly large value to minimize the number of extend operations that it must perform. At times, this value can exceed the available disk quota even though there is actually enough space for the file if only the required amount is used. You can use the DCL command SET RMS_DEFAULT/EXTEND_QUANTITY to limit (explicitly specify) the extension size to the recommended number of blocks. An appropriate size is the number of blocks specified as the cluster size for the device (set by the DCL command INITIALIZE/CLUSTER_SIZE). For large files on a volume where there is sufficient disk space, consider using a multiple of the cluster size to improve subsequent performance.

When creating a new file, you can specify the extension quantity for the file by setting the desired value in the FAB$W_DEQ field before or after invoking the Create service. This value becomes a permanent attribute for the file.

When processing an existing file, you can temporarily override the default extension quantity specified when the file was created. To do this, set the desired value before or after invoking the Open service. When you subsequently close the file, the default extension quantity reverts to the value set when the file was created.

See the discussion under FAB$B_BKS for indexed files.

Note that the use of an allocation control XAB overrides the value in this field. See Chapter 8 for a detailed description of allocation control XABs.

4.11 FAB$L_DEV Field

The device characteristics (DEV) field allows your program to obtain the generic characteristics of the device containing the file. You can locate and test the various bits in the field using symbolic offsets. RMS returns a value in this binary options field when you invoke an Open, Create, or Display Service. RMS also returns a value in this field for the Parse service unless you take the syntax check option (NAM$V_SYNCHK in the NAM$B_NOP field is clear).

Table 4–2 describes the bits in the device characteristics field. Each bit has its own symbolic bit offset and mask value. These definitions are made available to your program by referring to the $DEVDEF macro in SYS$LIBRARY:STARLET.MLB. The symbolic bit offset is formed by prefixing the characteristic name with DEV$V_. The mask value is formed by prefixing the characteristic name with DEV$M_. For example, the DEV$V_REC bit has a mask value of DEV$M_REC.

Table 4–2 Device Characteristics

Bit Offset	Description
DEV$V_ALL	Device is allocated.
DEV$V_AVL	Device is available for use.
DEV$V_CCL	Carriage control device.
DEV$V_CDP	Device has dual access paths, one of which is a remote node using an MSCP server.
DEV$V_CLU	Device is available on a VMScluster.
DEV$V_DET	Terminal device is detached.
DEV$V_DIR	Directory-structured device.
DEV$V_DMT	Device is marked for dismount.
DEV$V_DUA	Device has dual access paths, both of which use a disk class driver.
DEV$V_ELG	Device is error log enabled.
DEV$V_FOD	File-oriented device (disk and magnetic tape).
DEV$V_FOR	Device is mounted foreign (non-file-structured).
DEV$V_GEN	Device is a generic device.
DEV$V_IDV	Device can provide input.
DEV$V_MBX	Device is a mailbox.
DEV$V_MNT	Device is currently mounted.
DEV$V_NET	Network device.
DEV$V_ODV	Device can accept output.
DEV$V_OPR	Device has been enabled as an operator console terminal.
DEV$V_RCK	Device has read-check enabled.
DEV$V_RCT	Device includes a Replacement and Caching Table. See the *OpenVMS I/O User's Reference Manual* for more information about RCTs.
DEV$V_REC	Record-oriented device (terminal, mailbox, line printer, for example). If field is 0, device is assumed to be block-oriented (disk device or magnetic tape device). All record-oriented devices are considered sequential in nature.
DEV$V_RND	Device is random access in nature (disk).
DEV$V_RTM	Device is real-time in nature; not suitable for use by RMS.
DEV$V_RTT	Terminal device is a remote terminal (DCL command SET HOST).
DEV$V_SDI	Single directory device (master file directory only).
DEV$V_SHR	Shareable device.
DEV$V_SPL	Device is being spooled.
DEV$V_SQD	Sequential block-oriented device (magnetic tape).
DEV$V_SWL	Device is currently software write-locked.
DEV$V_TRM	Terminal device.
DEV$V_WCK	Device has write-check enabled.

For DECnet for OpenVMS operations, this field represents the actual characteristics of the target device when a Create, Open, or Display service is invoked. It is not filled in when a Parse service is invoked using a file specification that contains a node name.

4.12 FAB$L_DNA Field

The default file specification string address (DNA) field provides the address of a file specification string RMS uses to apply defaults for any missing components of the file specification. This field works with the FAB$B_DNS field, which initializes the default file specification string size, to provide a default file specification string. Defaults are applied after RMS examines the primary file specification string to which the FAB$L_FNA field (described in Section 4.15) points.

This field and the FAB$B_DNS field correspond to the FDL attribute FILE DEFAULT_NAME.

The FAB$L_DNA field contains the symbolic address of a default file specification string, which is an ASCII string containing one or more components of a file specification. The components in the string must be in the order in which they would occur in a complete file specification.

The default file specification string is used primarily when a process accepts file specifications interactively; normally, file specifications known to a user program are specified completely in the FAB$L_FNA and FAB$B_FNS fields. You can specify defaults for one or more of the following file specification components:

- Node
- Device
- Directory
- File name
- File type
- File version number

The default file specification string is used only if components are missing from the string whose address is stored in the FAB$L_FNA field and those components are present in the default file specification string.

4.13 FAB$B_DNS Field

The default file specification string size (DNS) field indicates the size, in bytes, of the string whose address is contained in the FAB$L_DNA field. This field contains a numeric value in the range 0 to 255.

This field and the FAB$L_DNA field correspond to the FDL attribute FILE DEFAULT_NAME.

4.14 FAB$B_FAC Field

The file access (FAC) field specifies the operations and services a process is seeking to use in accessing a file. RMS uses this field, together with the share field (SHR) in each potential accessor's FAB, to determine whether to permit a process to access a file. The FAC field corresponds to the FDL primary attribute ACCESS.

Within the FAC field, each bit position corresponds to an operation or service option that the process intends to use when accessing the file. In this manner, a process may specify several options, assuming they are compatible, by setting the appropriate bits. Each option has its own symbolic bit offset and mask value. For example, the GET service option has a symbolic bit offset of FAB$V_GET and a mask value of FAB$M_GET.

When RMS attempts to open a file for a process, it examines the process's FAB$B_FAC field to determine what operations or services the process is seeking to use in conjunction with the file access.

RMS determines whether or not the process seeking access to the file intends to use operations and services that are compatible with the sharing options permitted by processes currently accessing the file. It checks the FAC field of the requesting process to determine whether it requires read or write access to the file. It then checks the SHR field of the requesting process to determine whether it will share read or write access with other processes that are accessing the file. A read (GET) implies read access. Delete (DEL), write (PUT), truncate (TRN), and update (UPD) all imply write access.

For example, assume that Process A opens the file for GET access (FAC=GET) and is willing to share the file with processes that are doing GET and PUT accesses (SHR=GET,PUT). Since this is the only process accessing the file, RMS permits it to read access the file.

Assume that a second process, Process B, wants to access the same file doing PUT accesses (FAC=PUT) and is willing to share the file with processes doing GET accesses and PUT accesses (SHR=GET,PUT). Because Process B is compatible with Process A (they both agree to share the file with any process that is doing either GET accesses or PUT accesses), RMS permits the second process to access the file.

Now assume that a third process, Process C, wants GET access (FAC=GET) to the same file but will share the file only with processes doing GET accesses (SHR=GET). Although Process C is compatible with Process A (FAC=GET), it is not compatible with Process B (FAC=PUT), so RMS denies Process C access to the file. Conversely, if C tries to access the file before B, C gets access and B is denied access.

RMS always grants file access to the first process accessing a file, assuming no security access restrictions exist. When a process acquires access to a file, RMS rejects any attempt to use a service not included in the initial access request.

Options

FAB$V_BIO

Requests file access for doing block I/O operations that use Read (FAB$V_GET), Write (FAB$V_PUT), or the Space services. Specifying block I/O prohibits the use of record I/O operations (such as the Get, Put, Update, Delete, or Truncate services).

This option corresponds to the FDL attribute ACCESS BLOCK_IO.

FAB$V_BRO

Requests file access for doing either block I/O or record I/O as determined by the state of the RAB$V_BIO bit in the RAB at connect time. For additional information, see Section 6.19 and Section B.3.10.

This option corresponds to the FDL attribute ACCESS RECORD_IO.

FAB$V_DEL

Requests file access for invoking the Delete service (or the equivalent language statement that deletes a record). This option applies only to relative and indexed files.

This option corresponds to the FDL attribute ACCESS DELETE.

FAB$V_GET
Requests file access for invoking either the Get or Find service (or equivalent language statement that reads a record). This is the default if a process requests access to a file without including FAB$B_FAC field information. If the process takes the FAB$V_GET option together with either the FAB$V_BIO option or the FAB$V_BRO option, it can invoke the Read service.

This option corresponds to the FDL attribute ACCESS GET.

FAB$V_PUT
Requests file access for invoking either the Put or Extend service (or the equivalent language statement that writes a record or extends a file). This is the default when a process creates a file. If the process takes the FAB$V_PUT option together with either the FAB$V_BIO option or the FAB$V_BRO option, it can invoke the Write service.

This option corresponds to the FDL attribute ACCESS PUT.

FAB$V_TRN
Requests file access for invoking the Truncate service (or the equivalent language statement that truncates a file). Also allows use of the RAB$L_ROP truncate-on-put (RAB$V_TPT) option with the Put and Write service. This option applies only to sequential files.

This option corresponds to the FDL attribute ACCESS TRUNCATE.

FAB$V_UPD
Requests file access for invoking either an Update or Extend service (or the equivalent language statement that rewrites a record or extends a file). Also allows use of the RAB$L_ROP update-if (RAB$V_UIF) option for the Put service.

This option corresponds to the FDL attribute ACCESS UPDATE.

4.15 FAB$L_FNA Field

The file specification string address (FNA) field works with the FAB$B_FNS field to specify the primary file specification of the file to be processed. If this string does not contain all the components of a full file specification, RMS uses the defaults supplied in the default file specification string (see FAB$L_DNA and FAB$B_DNS). If no default string is present, or if the file specification is still incomplete, RMS provides additional defaults.

This field contains the symbolic address of a file specification string, which is an ASCII string containing one or more components of a file specification. This field is used as input by many file-processing services. To obtain the file specification returned by RMS after it translates any logical names and applies defaults, a NAM block must be present (see FAB$L_NAM).

This field and the FAB$B_FNS field correspond to the FDL attribute FILE NAME.

4.16 FAB$B_FNS Field

The file specification string size (FNS) field specifies the size, in bytes, of the ASCII file specification string, whose address is contained in the FAB$L_FNA field. This field contains a numeric value in the range of 0 through 255.

This field and the FAB$L_FNA field correspond to the FDL attribute FILE NAME.

4.17 FAB$L_FOP Field

FAB$L_FOP is the symbolic offset value for the FAB's file-processing options (FOP) field. This field specifies which of the various optional file operations are to be implemented for the process.

The FOP is a 32-bit field in which each file-processing option has a corresponding bit assignment to let you specify multiple options (multiple bits can be set), when applicable. Each option has a unique symbolic offset value and mask value, but you need only specify the appropriate 3-letter mnemonic when coding a function. For example, the spool-file-on-close option has a symbolic offset value of FAB$V_SPL, but to specify the option, you use the following MACRO statement:

```
FOP=SPL
```

As detailed in the appropriate descriptions, the only file-processing option bits that may be affected by record management services are the FABV_CBT, FABV_CTG, FAB$V_RCK, and FAB$V_WCK bits.

This section presents the seven types of file-processing options alphabetically by functional category:

- Allocation and extension options
- File name parsing modifiers
- File disposition options
- Magnetic tape processing options
- Nonstandard processing options
- Performance options
- Reliability options

Table 4–3 lists each of the options alphabetically by category.

Table 4–3 File Processing Options

Option	Symbolic Offset
Allocation and Extension Options	
Contiguous best try	FAB$V_CBT
Contiguous allocation	FAB$V_CTG
Truncate at end of file	FAB$V_TEF

(continued on next page)

Table 4–3 (Cont.) File Processing Options

Option	Symbolic Offset
File Name Parsing Modifier Options	
Create-if	FAB$V_CIF
Maximum version number	FAB$V_MXV
Use NAM block inputs	FAB$V_NAM
Output file parse	FAB$V_OFP
Supersede existing file	FAB$V_SUP
File Disposition Options	
Delete on close	FAB$V_DLT
Submit command file on close	FAB$V_SCF
Spool file on close	FAB$V_SPL
Temporary marked for delete	FAB$V_TMD
Temporary file	FAB$V_TMP
Magnetic Tape Processing Options	
Do not set to EOF	FAB$V_NEF
Current position	FAB$V_POS
Rewind file on close	FAB$V_RWC
Rewind file on open	FAB$V_RWO
Nonstandard Processing Options	
Non-file-structured	FAB$V_NFS
User file open	FAB$V_UFO
Performance Options	
Asynchronous operation	FAB$V_ASY
Deferred write	FAB$V_DFW
Sequential only	FAB$V_SQO
Synchronous status	FAB$V_SYNCSTS
Reliability Options	
Read-check	FAB$V_RCK
Write-check	FAB$V_WCK

This field corresponds to the FDL primary attribute FILE.

Allocation and Extension Options

FAB$V_CBT

Contiguous best try; indicates that the file is to be allocated contiguously on a ‘‘best effort’’ basis. It is input to the Create service and output from the Open service to indicate the file status. The FAB$V_CBT option overrides the FAB$V_

CTG option. Note that this option is ignored if multiple areas are defined for an indexed file.

This option corresponds to the FDL attribute FILE BEST_TRY_CONTIGUOUS.

FAB$V_CTG
Contiguous; indicates that the space for the file is to be allocated contiguously. If this cannot be done, the operation fails. It is input to the Create service and is output by the Open service to indicate the status of the file. Note that this option is ignored if multiple areas are defined for an indexed file. The FAB$V_CBT option overrides the FAB$V_CTG option.

This option corresponds to the FDL attribute FILE CONTIGUOUS.

FAB$V_TEF
Truncate at end of file; indicates that unused space allocated to a file is to be deallocated on a Close service. This option is tested only at $CLOSE time. The FAB$V_TEF option applies only to sequential files.

This option corresponds to the FDL attribute FILE TRUNCATE_ON_CLOSE.

File Name Parsing Modifiers

FAB$V_CIF
Create if nonexistent; creates and opens a file and returns the alternate success status RMS$_CREATED, assuming the file does not exist. If you specify an existing file, RMS opens it. Note that if you specify the CIF option for an indexed file, you need to provide a key XAB. If you do not provide a key XAB, RMS returns an RMS$_NPK error.

The FAB$V_CIF option is input only to the Create service and overrides the FAB$V_SUP option. When the create-if option is used with a search list logical name and the file is not found in any of the file specifications supplied using the search list, the file is created using the file specification from the first element of the search list.

This option corresponds to the FDL attribute FILE CREATE_IF.

FAB$V_MXV
Maximize version number; indicates that the version number of the file should be the maximum of the explicit version number given in the file specification, or one greater than the highest version number for an existing file in the same directory with the same file name and file type. This option enables you to create a file with a specific version number (if the requested version number is greater than that of the existing file) or a file with a version number that is one higher than the existing file's version number.

This option is used as input to the Create service only and it corresponds to the FDL attribute FILE MAXIMIZE_VERSION (default is ''YES'').

FAB$V_NAM
Use NAM block inputs; indicates that the NAM block whose address is contained in the FAB$L_NAM (name block address) field provides the device, file, and/or the directory identification when a file is being opened, closed, or erased (deleted). If a file is being created, the field specifies the device and directory identification.

This option has no corresponding FDL attribute and it is not supported for DECnet for OpenVMS operations.

FAB$V_OFP
Output file parse; specifies that related file resultant file specification strings, if used, are to provide directory, file name, and file type defaults only (requires NAM block).

This option corresponds to the FDL attribute FILE OUTPUT_FILE_PARSE.

FAB$V_SUP
Supersede existing file; allows an existing file to be superseded on a Create service by a new file of the same name, type, and version. The FAB$V_CIF and the FAB$V_MXV option take precedence over the FAB$V_SUP option.

This option corresponds to the FDL attribute FILE SUPERSEDE.

File Disposition Options

FAB$V_DLT
Delete file on Close; indicates that the file is to be deleted when closed. This option may be specified for the Create, Open, or Close services. However, if you set the bit when you create or open a file, RMS deletes the file when you close it, regardless of the state of the bit when you invoke the Close service. You can specify the FAB$V_DLT option with the FAB$V_SCF or FAB$V_SPL option.

This option corresponds to the FDL attribute FILE DELETE_ON_CLOSE.

FAB$V_SCF
Submit command file on Close; indicates that the file is to be submitted as a batch-command file to the process-default batch queue (SYS$BATCH) when the file is closed. This option can be specified for the Create, Open, and Close services. However, if you set the bit when you create or open a file, RMS submits the file to SYS$BATCH when you close it, regardless of the state of the bit when you invoke the Close service.

The FAB$V_SCF option applies to sequential files only and it corresponds to the FDL attribute FILE SUBMIT_ON_CLOSE.

FAB$V_SPL
Spool file on Close; indicates that the file is to be spooled to the process-default print queue (SYS$PRINT) when the file is closed. This option can be specified for the Create, Open, or Close services. However, if you set the bit when you create or open a file, RMS spools the file to SYS$PRINT when you close it, regardless of the state of the bit when you invoke the Close service.

The FAB$V_SPL option applies to sequential files only and it corresponds to the FDL attribute FILE PRINT_ON_CLOSE.

FAB$V_TMD
Temporary file marked for delete; indicates that a temporary file is to be created but is to be deleted when the file is closed. This option is input only to the Create service. The FAB$V_TMD option takes precedence over the FAB$V_TMP option.

This option corresponds to the FDL attribute FILE TEMPORARY.

FAB$V_TMP
Temporary file; indicates that a temporary file is to be created and retained, but that no directory entry is to be made for it. This option is used solely as input to the Create service. If you have a NAM block, you are given the file identification (FID) of the file, which you can use to reopen the file. If you do not have a NAM block or if you do not save the FID, the file becomes inaccessible once it is closed. The FAB$V_TMD option overrides the FAB$V_TMP option.

This option corresponds to the FDL attribute FILE DIRECTORY_ENTRY (''NO'' means this bit is set).

Magnetic Tape Processing Options

FAB$V_NEF
Do not position to end of file; inhibits positioning to the end of a file when a tape file is opened and the FAB$B_FAC (file access) field indicates a Put service.

This option corresponds to the FDL attribute FILE MT_NOT_EOF.

FAB$V_POS
Current position; directs RMS to position the magnetic tape volume set immediately after the most recently closed file (the current position) when it opens the next file. If you use this option when you invoke the Create service, RMS begins overwriting data beginning with the current tape position.

The FAB$V_POS option corresponds to the FDL attribute FILE MT_CURRENT_POSITION and is superseded by the FAB$V_RWO option, where applicable.

FAB$V_RWC
Rewind file on Close; specifies that the magnetic tape volume is to be rewound when the file is closed. This option can be specified for the Close, Create, or Open services.

This option corresponds to the FDL FILE attribute MT_CLOSE_REWIND.

FAB$V_RWO
Rewind on Open; specifies that the magnetic tape volume is to be rewound before the file is opened or created. If you use this option when you invoke the Create service, RMS overwrites the tape beginning with the first file. The FAB$V_RWO option takes precedence over the FAB$V_POS option.

This option corresponds to the FDL FILE attribute MT_OPEN_REWIND and takes precedence over the FAB$V_POS option, where applicable.

Nonstandard Processing Options

FAB$V_NFS
Non-file-structured; indicates (on an Open or Create service) that the volume is to be processed in a non-file-structured manner. This option allows the use of volumes created on non-Digital systems.

The FAB$V_NFS option corresponds to the FDL attribute FILE NON_FILE_STRUCTURED and it is not supported for DECnet for OpenVMS operations.

FAB$V_UFO
User file open; indicates that RMS operations for this file are limited to opening it or creating it. To perform additional processing of the file, invoke the $QIO system service using the channel number returned by RMS in the status value field (FAB$L_STV). This channel is assigned the access mode of the caller unless otherwise specified by the FAB$V_CHAN_MODE bits.

If you specify this option, you *must* set the FAB$B_SHR field FAB$V_UPI bit option unless the file is not shared (FAB$B_SHR field FAB$V_NIL option is set). For the Create service, the end-of-file mark is set to the end of the block specified in the FAB$L_ALQ field on input. For either the Open or Create services, the FAB$W_IFI field is set to 0 on return to indicate that RMS cannot perform any more operations on the file. If you set the FAB$V_UFO option with the Open or Create service, the channel needs only to be deassigned when you finish with the file. A Close service is not required.

This option corresponds to the FDL attribute FILE USER_FILE_OPEN and it is not supported for DECnet for OpenVMS operations.

Performance Options

FAB$V_ASY

Asynchronous; indicates that the specified task is to be done asynchronously. The FAB$V_ASY option is relevant only to file tasks that involve I/O operations. The asynchronous I/O option is typically used with success/error ASTs, or in conjunction with the $WAIT service, to synchronize the program with task completion. When you specify FAB$V_ASY, you pass the address of the FAB as an argument to the AST routine and RMS returns control to your program immediately.

This option corresponds to the FDL attribute FILE ASYNCHRONOUS.

FAB$V_DFW

Deferred write; indicates that writing back to the file of modified I/O buffers is to be deferred until the buffer must be used for other purposes. This option applies to relative files, indexed files, and sequential files opened for shared access.

This option corresponds to the FDL attribute FILE DEFERRED_WRITE and is not supported for DECnet for OpenVMS operations.

FAB$V_SQO

Sequential only; indicates that the file can be processed only in a sequential manner, permitting certain processing optimizations. Any attempt to perform random access results in an error. The FAB$V_SQO option is input to the Create and Open services.

This option corresponds to the FDL attribute FILE SEQUENTIAL_ONLY.

Note

For DECnet for OpenVMS operations, this option enables file transfer mode for Get, Put, Read, and Write services. File transfer mode is a Data Access Protocol (DAP) feature that allows several records to be transferred in a single-network I/O operation to maximize throughput for single-direction, sequential access file transfer.

FAB$V_SYNCSTS

Synchronous status; returns the success status RMS$_SYNCH if the requested service completes its task immediately. The most common reason for not completing a task immediately is that the task involves I/O operations. If the service completes synchronously (that is, it has not returned to caller’s execution mode prior to completion), RMS returns RMS$_SYNCH as the completion status in R0, stores the true completion status (success or failure) in FAB$L_STS, and does not deliver an AST.

The FAB$V_SYNCSTS option is best used in conjunction with the FAB$V_ASY option.

The system returns RMS$_SYNCH status in R0. Refer to the FAB$L_STS field for the actual success status or failure status of the task.

Reliability Options

FAB$V_RCK

Read-check; specifies that transfers from disk volumes are to be checked by a read-compare operation, which effectively doubles the amount of disk I/O at some increase in reliability. This option is an input to the Open and Create services. If FAB$V_RCK is set, then checking is performed for the duration of the access. The FAB$V_RCK option is also an output of the Open service, which indicates the default for the file. This option is not available for RX01 and RX02 devices, or for any device that has been mounted using the DCL command MOUNT/FOREIGN.

This option corresponds to the FDL attribute FILE READ_CHECK.

FAB$V_WCK

Write-check; indicates that transfers to disk volumes are to be checked by a read-compare operation. The FAB$V_WCK option is similar to the FAB$V_RCK option. This option is not available for RX01 and RX02 devices, or for any device that has been mounted using the DCL command MOUNT/FOREIGN.

This option corresponds to the FDL attribute FILE WRITE_CHECK.

4.18 FAB$B_FSZ Field

The fixed-length control area size (FSZ) field is used only for variable with fixed-length control (VFC) records. When you create a file with this record type, you must set the value for the fixed-control area before you issue the Create service. When you open an existing file that contains variable with fixed control records, RMS sets this field equal to the value specified when the file was created. The FAB$B_FSZ field is not applicable to indexed files.

This field corresponds to the FDL attribute RECORD CONTROL_FIELD_SIZE.

This field contains a numeric value in the range of 1 to 255 that indicates, in bytes, the size of the fixed control area; the default size is 2 bytes. If you do not specify a value or specify 0, then the default size is used.

4.19 FAB$W_GBC Field

The global buffer count (GBC) field indicates the requested number of global buffers for a file. This field contains a numeric value in the range of 0 to 32,767; the default is 0.

Global buffers support sharing of I/O buffers by more than one process. The use of global buffers can minimize I/O operations for a shared file, thus reducing record access time at the cost of using additional system resources. RMS is able to locate requested records (or blocks) in the global buffers associated with this file, which it can read directly from memory, eliminating much I/O. However, since global buffers use global sections, the value contained in FAB$W_GBC is limited by systemwide restrictions on resources determined by the system parameters GBLSECTIONS (number of global sections), GBLPAGES (number of global page table entries), GBLPAGFIL (number of systemwide pages allowed for global page-file sections, or scratch global sections), and RMS_GBLBUFQUO (total number of simultaneously active global buffers allowed for the system). For a complete description of these parameters, see the *OpenVMS System Management Utilities Reference Manual*.

If global buffers are specified for a file, global buffers are used instead of local (process) buffers, with the exception of deferred write operations (FAB$L_FOP field FAB$V_DFW option).

The value that is specified when the file is created is returned in the FAB$W_GBC field as output from the Open service. This value is then used as input to the Connect service.

If you want to override the default value specified when the file was created, you can set a different value in the FAB$W_GBC field after opening the file but before invoking the Connect service. If you do not want to use global buffers, you can clear the field before issuing the Connect service if the default value is not 0.

If you modify the value in the FAB$W_GBC field that is returned from the Open service prior to the Connect service, this action determines whether or not global buffers are assigned to your process.

If you want to permanently change the default global buffer count value for the file, use the following command syntax:

```
SET FILE file-spec /GLOBAL_BUFFERS=buffer-count
```

If you want to permanently clear the default global buffer count for a file, use the following command syntax:

```
SET FILE file-spec /GLOBAL_BUFFERS=0
```

You can also vary the number of global buffers used each time you process the file. If you choose this method, you change (or clear) the FAB$W_GBC field after you open the file, but before you invoke the Connect service. In this case, the specified value is assigned to the FAB$W_GBC field, or the FAB$W_GBC field remains clear only for the current processing of the file; that is, you do not permanently alter the FAB$W_GBC field in the FAB. If no value is specified in the FAB$W_GBC field when the file is created, the default value is 0.

The number of global buffers for a file is determined by the first record stream to connect to the file (systemwide). If the file is already open and connected, then the number of global buffers is already set and modifications made before the Connect service are useful only to request that this process use (or not use) global buffers.

To specify a read-only global buffer cache, the initial accessor must set the FAB$B_SHR field FAB$V_SHRGET and FAB$V_MSE bits on. Selecting the FAB$V_MSE option turns on locking to coordinate access to the global buffer cache.

You can use global buffers for all file organizations opened for shared record access. If the global buffer count is nonzero for the first process that connects to the file, then a temporary global section that is large enough to contain the specified number of buffers (as well as internal RMS data structures) is created and mapped. This section is mapped by processes that subsequently connect to the file, thus allowing multiple processes to reference a single set of one or more buffers without performing additional I/O operations. Thus, the first user to open the file requesting global buffers determines the number of the global buffers. For shared sequential file operations, the value stored in the RAB$B_MBC field establishes the global buffer size. See Section 6.11 for more information.

The FAB$W_GBC field corresponds to the FDL attribute FILE GLOBAL_BUFFER_COUNT and it is not supported for DECnet for OpenVMS operations.

4.20 FAB$W_IFI Field

The internal file identifier (IFI) field associates the FAB with the corresponding internal file access block. RMS sets this field on successful Create or Open services. It is then an input for subsequent Close, Connect, Display, and Extend services. The Close service deallocates the internal control structures and clears the FAB$W_IFI field. When the user file open (FAB$V_UFO) option in the FAB$L_FOP field is specified, no internal structures are allocated on Create or Open services. Therefore, the FAB$W_IFI field remains cleared.

There is no FDL equivalent for this field.

4.21 FAB$B_JOURNAL Field (VAX Only)

VAX

On VAX systems, the journal field provides flags to identify whether a file is marked for journaling and to identify, where applicable, the type of journaling:

- After-image (AI) journaling
- Before-image (BI) journaling
- Recovery unit (RU) journaling

These flags are used for output *only*, from either the Open service or the Display service. Although they are included as inputs to the Create service, the input value must be 0. If the input value is nonzero, RMS returns an RMS$_JNS error. You can only store the associated attributes in the file header through the DCL interface using the SET FILE command with an appropriate qualifier. (See the *OpenVMS DCL Dictionary* for information about using the SET FILE command to store these attributes in a file header.)

You can obtain additional information about a file marked for journaling using the journaling XAB. For example, you can obtain the name of the after-image journal file, and so forth.

Here is a code example showing how you might use the Open service to determine if a file is marked for RU journaling. In the example, the program opens a file and branches to a label FILE_MARKED_FOR_RU if the file is marked for RU journaling:

```
$OPEN   FAB = MY_FAB
BLBC    R0,ERROR
BBS     #FAB$V_RU,FAB$B_JOURNAL+MY_FAB,-
        FILE_MARKED_FOR_RU
```

Each journaling flag in the field has a unique symbolic bit offset and mask value. For example, the after-image journaling flag has the symbolic bit offset FAB$V_AI and the constant value FAB$M_AI. If a flag is set, the logical value is 1; if it is reset, the logical value is 0.

There are no corresponding FDL attributes for the journaling flags because they cannot be set through the FDL interface.

A listing of symbolic offsets for each of the journaling flags follows:

Flags

FAB$V_AI
The file is marked for after-image journaling.

FAB$V_BI
The file is marked for before-image journaling.

FAB$V_RU
The file is marked for recovery unit journaling.♦

4.22 FAB$V_LNM_MODE Subfield

The logical name translation access mode (LNM_MODE) subfield is the part of the FAB$B_ACMODES field that specifies the RMS access mode used to translate logical names during parsing.

The FAB$V_LNM_MODE subfield may contain one of these four values, with the related constant value for each shown in parentheses:

0	None
1	Executive mode (PSL$C_EXEC)
2	Supervisor mode (PSL$C_SUPER)
3	User mode (PSL$C_USER)

The default value is 0 (none), which RMS interprets as user mode.

The FAB$V_LNM_MODE field is not supported for DECnet for OpenVMS operations, and it is ignored during remote file access.

There is no corresponding FDL equivalent for this field. For more information about logical name concepts, see the *OpenVMS Programming Interfaces: Calling a System Routine*.

4.23 FAB$L_MRN Field

The maximum record number (MRN) field applies only to relative files and indicates the highest record number that can be written to a file.

This field contains a numeric value of the highest numbered record allowed in the file, in the range of 0 to 2,147,483,647, although the maximum value depends on the number of blocks on the device to be used. The default for this field is 0.

If you attempt to write (put) or retrieve (get) a record with a relative record number higher than the specified limit, an error occurs and RMS returns a message indicating an invalid record number. Checking is suppressed if you specify 0 for the FAB$L_MRN field.

Note that RMS does not maintain the relative record number of the highest existing record in the file.

This field corresponds to the FDL attribute FILE MAX_RECORD_NUMBER.

4.24 FAB$W_MRS Field

The maximum record size (MRS) field defines the size of all records in a file with fixed-length records, the maximum size of variable-length records, the maximum size of the data area for variable with fixed-length control records, and the cell size (minus overhead) for relative files.

This field contains a numeric value in the range applicable to the file type and record format (see Table 4–4) that indicates the size of the records in the file, in bytes. This value specifies the number of bytes of data and does *not* include any control bytes associated with each record.

For fixed-length records, the value represents the actual size of each record in the file. You must specify a size when you create a file with fixed-length records.

For variable-length records, the value represents the size of the largest record that can be written into the file. If the file is not a relative file, a value of 0 is used to suppress record size checking, thus indicating that there is no user limit on record size, except for the limitations listed in Table 4–4 and certain physical limitations. For magnetic tape files, a value of 0 sets an effective maximum record size that is equal to the block size minus 4.

The size of variable-length records must conform to physical limitations. With indexed and relative files, for example, records may not cross bucket boundaries. If both the FAB$B_BKS and FAB$W_MRS fields are 0 (not specified) for an indexed file, RMS attempts to calculate a reasonable bucket size, usually 2. Thus, if any record requires more than two buckets, you must explicitly specify the required value for the FAB$B_BKS or the FAB$W_MRS field. If the FAB$B_BKS field is specified, the value should specify a bucket size large enough to exceed the longest possible record.

For variable with fixed-length control records, the value includes only the data portion; it does not include the size of the fixed control area.

For all relative files, the size is used in conjunction with the FAB$B_BKS field to determine the size of the record cell. You must specify the FAB$W_MRS field when you create a relative file.

You specify a value when you invoke a Create service. RMS returns the maximum record size when you invoke an Open service.

Table 4–4 summarizes the maximum record size allowed for the various file and record formats.

Table 4–4 Maximum Record Size for File Organizations and Record Formats

File Organization	Record Format	Maximum Record Size
Sequential	Fixed length	32,767
Sequential (disk)	Variable length	32,767
Sequential (disk)	VFC	32,767-FSZ[1]
Sequential (disk)	Stream	32,767
Sequential (disk)	Stream-CR	32,767
Sequential (disk)	Stream-LF	32,767
Sequential (ANSI tape)	Variable length	9,995
Sequential (ANSI tape)	VFC	9,995-FSZ[1]
Relative	Fixed length	32,255
Relative	Variable length	32,253
Relative	VFC	32,253-FSZ[1]
Indexed, Prolog 1 or 2	Fixed length	32,234

[1]The FSZ represents the size, in bytes, of the fixed control area in a record having VFC record format. On a disk device, the length of the largest record in a sequential file using variable or VFC format is also maintained by RMS and is available through the longest record length field (XAB$W_LRL) in the file header characteristics XAB (XABFHC). See Chapter 10.

(continued on next page)

Table 4–4 (Cont.) Maximum Record Size for File Organizations and Record Formats

File Organization	Record Format	Maximum Record Size
Indexed, Prolog 1 or 2	Variable length	32,232
Indexed, Prolog 3	Fixed length	32,224
Indexed, Prolog 3	Variable length	32,224

For DECnet for OpenVMS remote file access, the maximum record size may be set by the /NETWORK_BLOCK_COUNT=n qualifier to the SET RMS_DEFAULT command or by a $XABITM parameter. DECnet for OpenVMS remote file access can support record sizes as large as the record sizes that RMS supports. The default number of blocks is equal to the system parameter RMS_DFNBC, the default for which is 8 blocks (4096 bytes). For more information about the SET RMS_DEFAULT command, see the *OpenVMS DCL Dictionary*. The system parameters are detailed in the *OpenVMS System Management Utilities Reference Manual*.

This field corresponds to the FDL attribute RECORD SIZE.

4.25 FAB$L_NAM Field

The name block address (NAM) field specifies the address of the NAM block used to invoke a file service, such as an Open or Create. The NAM block, described in Chapter 5, is required only in conjunction with the file specification processing services. But it can also be used with other services, typically to obtain a file specification string after all logical name translation is completed and all defaults applied. Note that the FAB$L_FOP option FAB$V_NAM must be specified for the NAM block to be used as input (see Section 4.17).

4.26 FAB$B_ORG Field

The file organization (ORG) field assigns the organization of the file.

The FAB$B_ORG field is a keyword value field in which each file organization has a symbolic value. Options are identified using 3-letter mnemonics. Each option in the FAB$B_ORG field has its own symbolic constant value. For example, the relative (REL) file organization has a constant value of FAB$C_REL.

You must set this field before you invoke a Create service. RMS returns the contents of this field when you invoke an Open service. The options are described in the following list:

- FAB$C_IDX — Indexed file organization
- FAB$C_REL — Relative file organization
- FAB$C_SEQ — Sequential file organization (default)

This field corresponds to the FDL attribute FILE ORGANIZATION.

4.27 FAB$B_RAT Field

The record attributes (RAT) field specifies control information associated with each record in a file, including carriage control information, block boundary control, and count byte formatting for variable-length records. Within the FAB$B_RAT field, each control bit has a unique symbolic offset and constant value. For example, the CR (carriage return) control bit has a symbolic offset of FAB$V_CR and a mask value of FAB$M_CR.

For most programs, the default value for the carriage control is FAB$V_CR (carriage return). When you create your own file, however, the default value is 0. When you want to create a stream format file or a file containing ASCII text, specify the FAB$V_CR option for the Create service. RMS sets this field when you invoke an Open service.

When a process-permanent file is accessed indirectly for output, the value in this field is always an input value. Therefore, RMS automatically uses the process-permanent file's record attributes.

This field corresponds to the FDL primary attribute RECORD.

Options

FAB$V_CR

Indicates that each record is to be preceded by a line feed and followed by a carriage return when the record is written to a carriage control device such as a line printer or terminal.

This option corresponds to the FDL attribute RECORD CARRIAGE_CONTROL CARRIAGE_RETURN. It cannot be used with either the FAB$V_FTN option or the FAB$V_PRN option.

FAB$V_FTN

Indicates that the first byte of each record contains a FORTRAN (ASA) carriage control character.

This option corresponds to the FDL attribute RECORD CARRIAGE_CONTROL FORTRAN. Records are defined as follows:

Byte 0 Value (Hex)	ASCII Character	Meaning
0	(null)	Null carriage control (sequence: print buffer contents).
20	(space)	Single-space carriage control (sequence: line feed, print buffer contents, carriage return).
30	0	Double-space carriage control (sequence: line feed, line feed, print buffer contents, carriage return).
31	1	Page eject carriage control (sequence: form feed, print buffer contents, carriage return).
2B	+	Overprint carriage control (sequence: print buffer contents, carriage return). Allows double printing for emphasis.
24	$	Prompt carriage control (sequence: line feed, print buffer contents).

Byte 0 Value (Hex)	ASCII Character	Meaning
Other values		Same as ASCII space character: single-space carriage control.

The FAB$V_FTN option cannot be used with either the FAB$V_CR option or the FAB$V_PRN option.

FAB$V_PRN

Indicates print file format for variable-length records having 2-byte fixed-length control fields, where the fixed-length control area contains the carriage control specification. The first byte of the control area constitutes a ‘‘prefix’’ area, that is, action to be taken before printing the record. The second byte constitutes a ‘‘suffix’’ area, that is, action to be taken after printing the record.

The following table shows the coding scheme for both control bytes (even though they are interpreted separately):

Bit 7	Bit 6	Bit 5	Bit 4	Meaning
0	0	0	0	To specify no carriage control (NULL), set bits 3 through 0 at 0.
0	x	x	x	Use bits 6 through 0 to specify a count of new lines (line feed followed by carriage return).
1	0	0	x	Output the ASCII C0 control character specified by the configuration of bits 4 through 0.
1	0	1	x	Reserved.
1	1	0	0	Skip to the vertical format unit (VFU) channel (1–16) specified by bits 3 through 0. Devices that do not have hardware VFUs translate these codes as a 1-line advance.
1	1	0	1	Reserved.
1	1	1	0	Reserved.

This option corresponds to the FDL attribute RECORD CARRIAGE_CONTROL PRINT. It cannot be used with either the FAB$V_CR option or the FAB$V_FTN option.

FAB$V_BLK

Applicable to sequential files only; indicates that records are not permitted to cross block boundaries. The FAB$V_BLK record attribute option may be used with any one of the other record attribute options (FABV_CR, FABV_FTN, or FAB$V_PRN), but it cannot be used with files that use the STREAM record format.

This option corresponds to the FDL attribute RECORD BLOCK_SPAN.

FAB$V_MSB

The state of this control bit determines whether the format for the 2-byte count field that is prefixed to each variable-length record is formatted as LSB or MSB. If the bit is set, RMS reads the contents of the 2-byte count field using the most significant byte as the high-order byte. The 2-byte count field contains the number of bytes in the associated variable-length record.

This option corresponds to the FDL attribute RECORD MSB_RECORD_LENGTH.

4.28 FAB$B_RFM Field

The record format (RFM) field specifies the format for all the records in a file.

The FAB$B_RFM field is a keyword value field where each record format has a symbolic value. Options are identified by mnemonics. Each option has its own symbolic constant value. For example, the FIX (fixed) record format has a symbolic constant value of FAB$C_FIX; the STMCR (stream with carriage return) record format has a symbolic constant value of FAB$C_STMCR.

When you create the file, you must set this field before you invoke the Create service. RMS returns the record format when you invoke an Open service. The record format options are described under **Options**.

This field corresponds to the FDL attribute RECORD FORMAT.

Options

FAB$C_FIX
Indicates fixed-length record format.

This option corresponds to the FDL attribute RECORD FORMAT FIXED.

FAB$C_STM
Indicates stream record format. Records are delimited by FF, VT, LF, or CR LF, and all leading zeros are ignored. This format applies to sequential files only and cannot be used with the block spanning option.

This option corresponds to the FDL attribute RECORD FORMAT STREAM.

FAB$C_STMCR
Indicates stream record format. Records are delimited by CR. This format is supported for sequential files only.

This option corresponds to the FDL attribute RECORD FORMAT STREAM_CR.

FAB$C_STMLF
Indicates stream record format. Records are delimited by LF. This format is supported for sequential files only.

This option corresponds to the FDL attribute RECORD FORMAT STREAM_LF.

FAB$C_UDF
Indicates undefined record format. The undefined record format is valid for sequential files only. This is the default value if the FAB is not initialized with a $FAB macro.

This option corresponds to the FDL attribute RECORD FORMAT UNDEFINED.

FAB$C_VAR
Indicates variable-length record format. For the $FAB macro, this is the default value.

This option corresponds to the FDL attribute RECORD FORMAT VARIABLE.

FAB$C_VFC
Indicates variable-length with fixed-length control record format. This format is not supported for indexed files.

This option corresponds to the FDL attribute RECORD FORMAT VFC.

If you intend to use stream record format, then specify the FAB$V_CR record attribute (see FAB$B_RAT).

4.29 FAB$B_RTV Field

The retrieval window size (RTV) field specifies the number of retrieval pointers RMS is to maintain in memory for the file. Retrieval pointers are stored in the file header and indicate the beginning of each extent associated with the file. If a file has been extended repeatedly, the extents may be scattered noncontiguously on the disk, requiring numerous retrieval pointers. When RMS needs to access a new extent, it must obtain the retrieval pointer for that extent. RMS first looks for the retrieval pointer in the retrieval window, which contains the number of retrieval pointers specified by this field. If the retrieval pointer is not in the retrieval window, RMS reads the file header, thereby requiring an additional I/O operation.

This field contains a numeric value in the range of 0 through 127, or 255. A value of 0 directs RMS to use the system default number of retrieval pointers. A value of 255 means to map the entire file, if possible. If you specify a value of 255 when creating a file, the initial number of retrieval pointers is minimal; as records are added, however, the number of retrieval pointers increases as the number of extents increases. The system resources required for retrieval windows are subtracted from the buffered I/O quota of the process. Values from 128 to 254 (inclusive) are reserved for future use.

This field corresponds to the FDL attribute FILE WINDOW_SIZE and it is not supported for DECnet for OpenVMS operations.

4.30 FAB$L_SDC Field

The secondary device characteristics (SDC) field is equivalent to the FAB$L_DEV field, except that secondary device characteristics refer to the intermediate device used for spooling or the logical link for DECnet for OpenVMS operations. Within the FAB$L_SDC field, the bit definitions are the same as those defined for the FAB$L_DEV field (see Table 4–2). Like the FAB$L_DEV field, the bit definitions must first be made available to your process referring to the $DEVDEF system macro definition; the values are set by certain record management services (see FAB$L_DEV for additional information).

4.31 FAB$B_SHR Field

The file sharing (SHR) field defines the record operations that the opening process allows sharing processes to perform. RMS supports file sharing for all file organizations.

Within the FAB$B_SHR field, each record operation that sharing processes are permitted to do has a corresponding bit assignment. You can specify multiple record operations (multiple bits may be set).

Options are identified by symbolic bit offsets. Note that conflicts between the names of symbolic offsets in the FAB$B_SHR field and the names of symbolic offsets in the FAB$B_FAC field are resolved by prefixing the letters SHR to the symbolic offset in the FAB$B_SHR field. For example, both the FAB$B_FAC and FAB$B_SHR fields have a bit that specifies the *get* record option. In the FAB$B_FAC field, this bit offset is assigned the symbol FAB$V_GET; in the FAB$B_SHR field, this bit is assigned the symbol FAB$V_SHRGET.

Note that the letters SHR in the mnemonic part of the bit offset symbol may be omitted by VAX MACRO programs. Thus, the GET option, which is common to the FAB$B_FAC and FAB$B_SHR fields, has a symbolic bit offset of FAB$V_SHRGET and a mask value of FAB$M_SHRGET, but VAX MACRO programs may use the synonyms FAB$V_GET and FAB$M_GET. This rule applies to the FABV_SHRPUT, FABV_SHRGET, FAB$V_SHRDEL, and FAB$V_SHRUPD options.

The way in which RMS uses the file access (FAB$B_FAC) field and file sharing (FAB$B_SHR) field is described in greater detail in the FAB$B_FAC field discussion.

Note that if you do not specify a value in the FAB$B_SHR field, the following defaults apply:

- If the FAB$B_FAC field is set or defaulted to FAB$V_GET, the FAB$B_SHR field defaults to FAB$V_SHRGET.
- If the FAB$B_FAC field is set or defaulted to either FABV_PUT, FABV_DEL, FAB$V_UPD, or FAB$V_TRN, the FAB$B_SHR field defaults to FAB$V_NIL. Thus, write sharing must be explicitly requested using the FAB$B_SHR field (because it is not the default).

This field corresponds to the FDL primary attribute SHARING. See the *Guide to OpenVMS File Applications* for additional details on file sharing.

The following list includes descriptions of the sharing options.

Options

FAB$V_MSE

Allows multistream access and is relevant for record operations only. You must specify FAB$V_MSE whenever you want to call Connect services for multiple RABs for this FAB.

Note that if you specify the FAB$V_MSE and FAB$V_BIO options, you must set the FAB$V_UPI bit regardless of the other sharing bits. To specify a read-only global buffer cache, the initial accessor must set the FAB$B_SHR field FAB$V_SHRGET and FAB$V_MSE bits. Selecting the FAB$V_MSE option turns on locking to coordinate access to buffers.

The FAB$V_MSE option is not supported for DECnet for OpenVMS operations. RMS returns an error when the application program attempts to connect a second stream. Although RMS cannot perform multistreaming for DECnet for OpenVMS operations, you can obtain similar functionality by using multiple FABs to access the file in a shared manner.

This option is available for all file organizations and corresponds to the FDL attribute SHARING MULTISTREAM.

FAB$V_NIL

Prohibits any file sharing by other users. If FAB$V_NIL is specified with other options, it takes precedence.

This option corresponds to the FDL attribute SHARING PROHIBIT.

FAB$V_SHRPUT

Allows other users to write records to the file or to extend the file.

This option corresponds to the FDL attribute SHARING PUT.

FAB$V_SHRGET
Allows other users to read the file.

This option corresponds to the FDL attribute SHARING GET.

FAB$V_SHRDEL
Allows other users to delete records from the file.

This option corresponds to the FDL attribute SHARING DELETE.

FAB$V_SHRUPD
Allows other users to update records that currently exist in the file or to extend the file.

This option corresponds to the FDL attribute SHARING UPDATE.

FAB$V_UPI
This option is used when the user wants to assume responsibility for interlocking of multiple, simultaneous accessors of a file. This option disables all RMS locking for the current access of the file. Except for block I/O, the FAB$V_MSE option overrides the FAB$V_UPI option. Usually, the FAB$V_UPI option is used for a file that is open for block I/O (FAB$V_BIO or FAB$V_BRO).

When you select the FAB$V_UFO option, you must also select the FAB$V_UPI option if the file is write shared. A file is specified as being write shared when you select either the FAB$V_PUT option, the FAB$V_DEL option, the FAB$V_TRN option, or the FAB$V_UPD option in the FAB$B_SHR field.

This option corresponds to the FDL attribute SHARING USER_INTERLOCK.

4.32 FAB$L_STS Field

RMS sets the completion status code (STS) field with success or failure codes before it returns control to your program (except for a subset of errors, as detailed in Section 2.4). Register 0 contains the same status as the STS field. Potential error codes for specific services are listed under their descriptions.

4.33 FAB$L_STV Field

RMS sets the status value (STV) field on the basis of the operation performed and the contents of the completion status code (FAB$L_STS) field, communicates additional completion information to your program.

4.34 FAB$L_XAB Field

The extended attribute block address (XAB) field specifies the XAB, or first of a series of XABs, that you want to use for file operations. This field contains the symbolic address of a XAB control block. A value of 0 (the default) indicates no XABs for the file.

For some operations, you must associate extended attribute blocks with a FAB to convey additional attributes about a file. (See Section 1.2.2 for a description of a XAB.) The FAB$L_XAB field can contain the symbolic address of the first associated block (of a potential chained list of such blocks) for the file.

RMS uses XAB values as follows:

- If you specify a XAB for either an Open or Display service, RMS returns the file attributes to the XAB.
- If you specify a XAB for a Create, Close, or Extend service, RMS uses the XAB as input to those functions.

5
Name Block (NAM)

The name (NAM) block provides additional fields for extended file specification use, including parsing and obtaining the actual file specification used for a file operation.

5.1 Summary of Fields

The symbolic offset, the size, and a brief description of each NAM block field are presented in Table 5–1. Additional details are given in the remaining sections of this chapter.

Table 5–1 NAM Block Fields

Field Offset	Size (Bytes)	Description
NAM$B_BID[1]	1	Block identifier
NAM$B_BLN[1]	1	Block length
NAM$B_DEV[2]	1	Device string length
NAM$L_DEV[2]	4	Device string address
NAM$W_DID[2]	6	Directory identification
NAM$B_DIR[2]	1	Directory string length
NAM$L_DIR[2]	4	Directory string address
NAM$T_DVI[2]	16	Device identification
NAM$L_ESA	4	Expanded string area address
NAM$B_ESL[2]	1	Expanded string length
NAM$B_ESS	1	Expanded string area size
NAM$W_FID[2]	6	File identification
NAM$L_FNB[2]	4	File name status bits
NAM$B_NAME[2]	1	File name string length
NAM$L_NAME[2]	4	File name string address
NAM$B_NODE[2]	1	Node name string length
NAM$L_NODE[2]	4	Node name string address
NAM$B_NOP	1	Name block options
NAM$L_RLF	4	Related file NAM block address
NAM$L_RSA	4	Resultant string area address

[1]This field is statically initialized by the $NAM macro to identify this control block as a NAM.

[2]This field cannot be initialized by the $NAM macro.

(continued on next page)

Table 5–1 (Cont.) NAM Block Fields

Field Offset	Size (Bytes)	Description
NAM$B_RSL[2]	1	Resultant string length
NAM$B_RSS	1	Resultant string area size
NAM$B_TYPE[2]	1	File type string length
NAM$L_TYPE[2]	4	File type string address
NAM$B_VER[2]	1	File version string length
NAM$L_VER[2]	4	File version string address
NAM$L_WCC[2]	4	Wildcard context

[2]This field cannot be initialized by the $NAM macro.

The NAM block fields have no corresponding FDL equivalents. However, if your application requires the presence of a NAM block, consider using the $NAM macro (or equivalent) in a USEROPEN or a USERACTION routine.

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

Depending on the services to be used, the user may need to allocate program storage for the expanded string and the resultant string. The Parse service uses the expanded string to pass information related to wildcards (or search lists) to the Search service. When it creates a resultant string for other file services, RMS uses the expanded string as a work area to apply defaults. You can use the resultant string with file services to provide the file specification that results from the translation of logical names and the application of defaults. Typical uses of the resultant string include showing the resulting file specification after a partial file specification is entered by a terminal user, reporting errors, and logging the progress of a program.

To request use of the expanded or resultant strings, you must indicate the address and size of the user-allocated buffer to receive the string. The expanded string is indicated by the NAM$L_ESA and NAM$B_ESS fields; the resultant string is indicated by the NAM$L_RSA and NAM$B_RSS fields. When it fills in the expanded or resultant strings, RMS returns the actual length of the returned string in the NAM$B_ESL or NAM$B_RSL fields.

The format and arguments of the $NAM and $NAM_STORE macros are defined in Appendix A.

5.2 File Specification Component Descriptors

For each element of the fully qualified file specification returned in the expanded-string field or the resultant-string field in the NAM block, RMS returns a descriptor in the NAM block made up of a 1-byte size field and a 4-byte (longword) address field. The fields of these descriptors are described as one of the following:

NAM$B_xxx	(size field of xxx)
NAM$L_xxx	(address field of xxx)

Descriptors

NAMB_NODE, NAML_NODE
Node name descriptor, including access control string and double colon (::) delimiter.

NAMB_DEV, NAML_DEV
Device name descriptor, including colon (:) delimiter.

NAMB_DIR, NAML_DIR
Directory name descriptor, including brackets ([ ] or <>).

NAMB_NAME, NAML_NAME
File name descriptor or, if the file specification following a node name is within quotation marks (''file''), a quoted string descriptor.

NAMB_TYPE, NAML_TYPE
File type descriptor, including period (.) delimiter.

NAMB_VER, NAML_VER
File version number descriptor, including semicolon (;) or period (.) delimiter.

These descriptors are returned, enabling the program to extract a particular component from the resultant string without having to parse the resultant or expanded string. The entire resultant or expanded string, including delimiters, is described by the various component descriptors. If the value in the NAM$B_RSL field is nonzero, then the descriptors point to the NAM$L_RSA field. If the value in the NAM$B_RSL field is 0 and the value in the NAM$B_ESL field is nonzero, then the descriptors point to the NAM$L_ESA field. In all other cases, they are undefined.

This is an example of a resultant file specification and its file specification component descriptors:

```
NODE"TEST password"::WORK_DISK:[TEST.TEMP]FILE.DAT;3

   NODE   NODE"TEST password"::
   DEV    WORK_DISK:
   DIR    [TEST.TEMP]
   NAME   FILE
   TYPE   .DAT
   VER    ;3
```

You can use the file component descriptors individually or collectively to describe sections of the resultant or expanded string. For example, if you want to use the file name and file type fields, use NAM$L_NAME for the starting address and NAM$B_NAME+NAM$B_TYPE for the total length.

5.3 NAM$B_BID Field

The block identifier (BID) field is a static field that identifies this control block as a NAM block. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value NAM$C_BID (this is done by the $NAM macro).

5.4 NAM$B_BLN Field

The block length (BLN) field is a static field that defines the length of the NAM block, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value NAM$C_BLN (this is done by the $NAM macro).

5.5 NAM$B_DEV and NAM$L_DEV Fields

The device name length and address (DEV) field contains a pointer to either the NAM$L_ESA or NAM$L_RSA fields, depending on whether a Parse service was invoked or an Open, Create, or Search service was invoked. You can tell which name string is used by referring to the NAM$B_ESL or NAM$B_RSL fields, which contain the length of the returned file specification string.

5.6 NAM$W_DID Field

The directory identification (DID) field identifies the directory for the file. RMS outputs this 3-word field as part of the Open, Create and Display services. RMS also outputs this field as part of the Parse service unless you specify the syntax check option (NAM$V_SYNCHK). If, once you open the file, you want to refer to this directory again, you can do so more quickly by specifying that the NAM block has a valid directory identifier.

This field is not supported for DECnet for OpenVMS operations; it is ignored on input and zero-filled on output.

5.7 NAM$B_DIR and NAM$L_DIR Fields

The directory name length and address (DIR) field contains a pointer to either the NAM$L_ESA or NAM$L_RSA fields, depending on whether a Parse service was invoked or an Open, Create, or Search service was invoked. You can tell which name string is used by referring to the NAM$B_ESL or NAM$B_RSL fields, which contain the length of the returned file specification string.

5.8 NAM$T_DVI Field

The device identification (DVI) field defines the device for the file. RMS outputs this field as part of the Open, Create, and Display services. RMS also outputs this field as part of the Parse service unless you specify the syntax check option (NAM$V_SYNCHK). You can use this field with the file identification field to reopen the file by referring to the NAM block. The symbolic value NAM$S_DVI gives the length of this field in bytes. The form of this field is a counted string. The first byte is a count of the number of characters following it.

This field is not supported for DECnet for OpenVMS operations; it is ignored on input and zero-filled on output.

5.9 NAM$L_ESA Field

The expanded string area address (ESA) field contains the symbolic address of a user buffer in the application program to receive the file specification string resulting from the translation of logical names and the application of default file specification information.

You must specify this field for processing wildcard characters.

5.10 NAM$B_ESL Field

RMS sets the expanded string length (ESL) field as part of the Open, Create, and Parse services. This field contains the length, in bytes, of the file specification string returned in the buffer whose address is in the NAM$L_ESA field.

5.11 NAM$B_ESS Field

The expanded string area size (ESS) field contains the size of the user-allocated buffer whose address is contained in the NAM$L_ESA field.

This field contains a numeric value representing the size, in bytes, of the user buffer that will receive the file specification string, in the range of 0 through 255.

The symbolic value NAM$C_MAXRSS defines the maximum possible length of an expanded file specification string.

5.12 NAM$W_FID Field

The file identification (FID) field is a 3-word field that RMS uses to identify the file when it invokes an Open, Create, or Display service. When you want to open a file by using the file identifier, set this field before you open the file.

This field is not supported for DECnet for OpenVMS operations; it is ignored on input and zero-filled on output.

5.13 NAM$L_FNB Field

The file name status (FNB) field is a binary options field that RMS uses to convey status information obtained from the file specification parsing routine. Each bit within this field denotes a specific status relative to the various components of the file specification.

Each status bit has its own offset and mask value. For instance, the number of directory levels (DIR_LVLS) field has a symbolic bit offset of NAM$V_DIR_LVLS and a mask value of NAM$M_DIR_LVLS. The bits and the conditions they express for the NAM$L_FNB field are described in Table 5–2.

Table 5–2 NAM$L_FNB Status Bits

Field Offset	Description
NAM$V_CNCL_DEV	Device name is a concealed device.
NAM$V_DIR_LVLS	Indicates the 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.

(continued on next page)

Table 5–2 (Cont.) NAM$L_FNB Status Bits

Field Offset	Description
NAM$V_HIGHVER	A higher-numbered version of the file exists (output from Create and Enter services). For DECnet for OpenVMS operations, this bit is returned as a binary zero.
NAM$V_LOWVER	A lower-numbered version of the file exists (output from Create and Enter services). For DECnet for OpenVMS operations, this bit is returned as a binary zero.
NAM$V_NODE	File specification includes a node name.
NAM$V_PPF	File is indirectly accessed process-permanent file.
NAM$V_QUOTED	File specification includes a quoted string; indicates that the file name length and address field contains a quoted string file specification. Applies to network operations or magnetic tape devices only.[1]
NAM$V_ROOT_DIR	Device name incorporates a root directory.
NAM$V_SEARCH_LIST	A search list logical name is present in the file specification.
NAM$V_WILDCARD	File specification string includes a wildcard; returned whenever any of the other wildcard bits is set.
NAM$V_WILD_DIR	Directory specification includes a wildcard character.
NAM$V_WILD_GRP	Group number contains a wildcard character.
NAM$V_WILD_MBR	Member number contains a wildcard character.
NAM$V_WILD_NAME	File name contains a wildcard character.
NAM$V_WILD_SFD1–NAM$V_WILD_SFD7	Subdirectory 1 through 7 specification includes a wildcard character.
NAM$V_WILD_TYPE	File type contains a wildcard character.
NAM$V_WILD_UFD	User file directory specification includes a wildcard character.
NAM$V_WILD_VER	Version number contains a wildcard character.

[1]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 are set.

5.14 NAM$B_NAME and NAM$L_NAME Fields

The file name length and address (NAME) field contains a pointer to either the NAM$L_ESA or NAM$L_RSA fields, depending on whether a Parse service was invoked or an Open, Create, or Search service was invoked. You can tell which name string is used by referring to the NAM$B_ESL or NAM$B_RSL fields, which contain the length of the returned file specification string.

5.15 NAM$B_NODE and NAM$L_NODE Fields

The node name length and address (NODE) field contains a pointer to either the NAM$L_ESA or NAM$L_RSA fields, depending on whether Parse service was invoked or an Open, Create, or Search service was invoked. You can tell which name string is used by referring to the NAM$B_ESL or NAM$B_RSL fields, which contain the length of the returned file specification string.

5.16 NAM$B_NOP Field

The name block options (NOP) field indicates the options applicable to the file name parsing services.

The NAM$B_NOP field is a binary options field in which each option has a corresponding bit assignment. Multiple options can be specified (multiple bits can be set) but the default state for each bit is clear (not set). Each option has its own symbolic bit offset and mask value. For example, the SYNCHK option has a symbolic bit offset of NAM$V_SYNCHK and a mask value of NAM$M_SYNCHK.

Options

NAM$V_NOCONCEAL
When used with the Open, Create, Parse, Search, or Display services, a concealed device logical name is translated into its physical device name (and directory, if so defined) in the resultant string field, whose address is supplied by the NAM$L_RSA field. If this option is not set and a concealed device name is used, the concealed device name appears in the resultant string field, instead of the physical device name (and directory, if so defined).

NAM$V_PWD
When used with the Create, Open, Parse, Search, or Display services, the password in the access control string of a node specification, if present, is returned unaltered in the expanded or resultant file specification fields. If you do not select this option, the actual password used is replaced by the word ‘‘password’’ in the resultant or expanded file specification string fields for security reasons.

NAM$V_SRCHXABS
When used with the Search service for remote file access, this option directs RMS to fill in the FAB and any chained XABs as if a Display service had been invoked. This allows you to obtain file attribute information using the Search service without the need to open the file.

NAM$V_SYNCHK
This gives you the option of using the Parse service to verify the syntax validity of the file specification without invoking I/O processing that verifies the actual existence of the specified device and directory.

If you invoke the Parse service without setting the NAMV_SYNCHK bit, an accompanying I/O process verifies that the device and directory exist. Note that this processing verifies the existence of the device and directory, only; it does not verify the existence of the file. You can only verify the existence of the file using a $SEARCH or $OPEN.

If you opt to set the NAM$V_SYNCHK bit when you invoke the Parse service, RMS does not return the device characteristics (FAB$L_DEV and FAB$L_SDC fields) and does not fill in the NAM$W_DID and NAM$T_DVI fields, rendering the results of the $PARSE unusable for subsequent Search services.

5.17 NAM$L_RLF Field

The related file NAM block address (RLF) field contains the symbolic address of the NAM block for the related file. This field supports the secondary file concept of the command language (DCL), giving an extra default level in processing file specifications.

To provide an extra level of file specification defaults, the related NAM block must have been used previously by an Open, Create, Search, or Display service to create a resultant file specification string. Moving the address of the related NAM block into the NAM$L_RLF field of the current NAM block specifies that the previously parsed NAM block's resultant file specification string should be used as a default when the current NAM block is parsed. Note that the previously parsed NAM block must contain a resultant file specification (see NAM$L_RSA and NAM$B_RSS for additional details).

Refer to the *Guide to OpenVMS File Applications* for additional details on file specification parsing concepts.

5.18 NAM$L_RSA Field

The resultant string area address (RSA) field contains the symbolic address of a buffer in your program to receive the resultant file specification string. The NAM$B_RSS field must also be specified to obtain a resultant file specification string.

This string is the fully specified name of the file that results from the resolution of all system defaults, including version numbers and wildcard character substitution in the expanded file name string. You must specify this field for wildcard processing.

5.19 NAM$B_RSL Field

The resultant string length (RSL) contains the length, in bytes, of the resultant file specification string returned in the buffer whose address is in the NAM$L_RSA field.

5.20 NAM$B_RSS Field

The resultant string area size (RSS) field defines the size of the user-allocated buffer whose address is contained in the NAM$L_RSA field.

This field contains a numeric value representing the size, in bytes, of the user buffer that will receive the resultant file specification string, in the range of 0 through 255.

The symbolic value NAM$C_MAXRSS defines the maximum possible length of a resultant file specification string.

5.21 NAM$B_TYPE and NAM$L_TYPE Fields

The file type length and address (TYPE) field contains a pointer to either the NAM$L_ESA or NAM$L_RSA fields, depending on whether a Parse service was invoked or an Open, Create, or Search service was invoked. You can tell which name string is used by referring to the NAM$B_ESL or NAM$B_RSL fields, which contain the length of the returned file specification string.

5.22 NAM$B_VER and NAM$L_VER Fields

The file version length and address (VER) field contains a pointer to either the NAM$L_ESA or NAM$L_RSA fields, depending on whether a Parse service was invoked or an Open, Create, or Search service was invoked. You can determine the name string that is used by referring to the NAM$B_ESL or NAM$B_RSL fields, which contain the length of the returned file specification string.

5.23 NAM$L_WCC Field

The wildcard context (WCC) field contains the information needed to use wildcard characters in place of the various file specification components. In particular, this field restarts a directory search to find the next matching file name, type, and/or version number. You can also use it to identify various RMS extended contexts; for instance, during remote file processing.

6

Record Access Block (RAB)

The record access block (RAB) defines run-time options for a record stream and for individual operations within a predefined record stream context. After you connect the file to a record stream and associate the record stream with a FAB, you use the RAB fields to specify the next record you want to access and to identify appropriate record characteristics.

Alpha

Note

If you are using 64-bit addressing on an OpenVMS Alpha system, refer to Chapter 7 for information about using RAB64 instead of RAB. ♦

6.1 Summary of Fields

Table 6–1 gives the symbolic offset, the size, the FDL equivalent, and a brief description of each RAB field.

Table 6–1 RAB Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
RAB$B_BID[1]	1	None	Block identifier
RAB$L_BKT	4	CONNECT BUCKET_CODE	Bucket code
RAB$B_BLN[1]	1	None	Block length
RAB$L_CTX	4	CONNECT CONTEXT	User context
RAB$L_FAB	4	None	File access block address
RAB$W_ISI[2]	2	None	Internal stream identifier
RAB$L_KBF	4	None	Key buffer address
RAB$B_KRF	1	CONNECT KEY_OF_REFERENCE	Key of reference
RAB$B_KSZ	1	None	Key size
RAB$B_MBC	1	CONNECT MULTIBLOCK_COUNT	Multiblock count
RAB$B_MBF	1	CONNECT MULTIBUFFER_COUNT	Multibuffer count
RAB$L_PBF	4	None	Prompt buffer address
RAB$B_PSZ	1	None	Prompt buffer size
RAB$B_RAC	1	CONNECT[3]	Record access mode

[1]This field is statically initialized by the $RAB macro to identify this control block as a RAB.

[2]This field cannot be initialized by the $RAB macro.

[3]This field contains options; corresponding FDL equivalents are listed in the description of the field.

(continued on next page)

Table 6–1 (Cont.) RAB Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
RAB$L_RBF	4	None	Record buffer address
RAB$W_RFA	6	None	Record file address
RAB$L_RHB	4	None	Record header buffer address
RAB$L_ROP	4	CONNECT[3]	Record-processing options
RAB$W_RSZ	2	None	Record size
RAB$L_STS[2]	4	None	Completion status code
RAB$L_STV[2]	4	None	Status value
RAB$W_STV0[4]	2	None	Low-order word status value
RAB$W_STV2[4]	2	None	High-order word status value
RAB$B_TMO	1	CONNECT TIMEOUT_PERIOD	Timeout period
RAB$L_UBF	4	None	User record buffer address
RAB$W_USZ	2	None	User record buffer size
RAB$L_XAB	4	None	Next XAB address

[2]This field cannot be initialized by the $RAB macro.

[3]This field contains options; corresponding FDL equivalents are listed in the description of the field.

[4]Alternate definition of RAB$L_STV field.

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations using files at remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

The format and arguments of the $RAB macro and the $RAB_STORE macro are described in Appendix A.

6.2 RAB$B_BID Field

The block identifier (BID) field is a static field that identifies the block as a RAB. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic offset value RAB$C_BID (this is done by the $RAB macro).

6.3 RAB$L_BKT Field

The bucket code (BKT) field is used with records in a relative file and when performing block I/O.

This field contains a relative record number or a numeric value representing the virtual block number to be accessed.

For relative files, the relative record number of the record acted upon (or which produced an error) is returned to the RAB$L_BKT field only after the completion of a sequential operation. That is, RMS returns the relative record number when you set the record access mode for sequential access (RAB$B_RAC is RAB$C_SEQ) on the execution of a Get, Put, or Find service.

Before performing block I/O on disk devices, this field must contain the virtual block number (VBN) of the first block you want to read or write. For all other devices, this field is not used. If you specify a VBN of 0, RMS begins the block transfer at the block pointed to by the next block pointer (NBP). (The NBP is an internal pointer maintained by RMS; it is described in Section B.3.10.)

This field is also input to the Space service to specify the number of blocks to be spaced forward or backward.

This field corresponds to the FDL attribute CONNECT BUCKET_CODE.

6.4 RAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the RAB, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value RAB$C_BLN (this is done by the $RAB macro).

6.5 RAB$L_CTX Field

The user context (CTX) field contains any user-selected value, up to four bytes long. This field is devoted exclusively to your use. RMS makes no use of the contents of this field; therefore, you can set any value you want in this field. For example, you could use this field to communicate with a completion routine in your program.

This field corresponds to the FDL attribute CONNECT CONTEXT.

6.6 RAB$L_FAB Field

The file access block address (FAB) field contains the symbolic address of the FAB for the file. Before you invoke the Connect service, you must set this field to indicate the address of the FAB associated with the open file.

6.7 RAB$W_ISI Field

The internal stream identifier (ISI) field associates the RAB with a corresponding FAB. RMS sets this field after the execution of a Connect service. A Disconnect service clears this field. This field should not be altered.

6.8 RAB$L_KBF Field

The key buffer address (KBF) field contains the symbolic address of the buffer containing the key value for random access. Note that the RAB$B_KBF field has the same offset as the RAB$L_PBF (prompt buffer address) field, but no conflict is presented because the fields are used in mutually exclusive operations.

When the RAB$B_RAC (record access mode) field specifies random access by key value, this field provides the address of a buffer that contains the key of the desired record. The key is the relative record number in files that are organized for relative access or in files organized for sequential access containing fixed-length records. In indexed files, the key value in RAB$L_KBF is used with the index specified in the key of reference field (RAB$B_KRF) to randomly access the desired record. Note that although a collating key affects the stored order for records, the collating value does not govern record lookups. For example, a collating sequence may assign the same ordering for the keys ‘‘dog’’ and ‘‘DOG’’. However, both keys do not have the same access (lookup) value. Therefore, when doing lookups, a program should specify either the specific key value or a range of values that includes the uppercase and lowercase combinations of the key. See the *Guide to OpenVMS File Applications* for more information about accessing indexed records.

Before you invoke the Get or Find service for doing random access in an indexed file, you place the address of the location containing the specified key value in the RAB$L_KBF field. The size of this key value must be specified in the RAB$B_KSZ field. During execution of the Get or Find service, RMS uses the specified key value for searching an index (which you specify using the RAB$B_KRF field) to locate the desired record. The type of match, that is, exact, generic, approximate, or approximate generic, is determined by examining the RAB$B_KSZ field in combination with selected search bits in the RAB$L_ROP field.

6.9 RAB$B_KRF Field

The key of reference (KRF) field specifies the key or index (primary, first alternate, and so on) to which the operation applies. The RAB$B_KRF field is applicable to indexed files only.

This field contains a numeric value representing the key path to records in a file. The value 0, the default, indicates the primary key. The values 1 through 254 indicate alternate keys.

When your program invokes a Get or Find service in random access mode, the key of reference specifies the index to search for a match on the key value that is described by the RAB$L_KBF and RAB$B_KSZ fields. When your program invokes a Connect or Rewind service, the key of reference identifies the index in the file of the next record in the stream. The next record is important when records are retrieved sequentially.

Note that although a collating key affects the stored order for records, the collating value does not govern record lookups. For example, a collating sequence may assign the same ordering for the keys ‘‘dog’’ and ‘‘DOG’’. However, both keys do not have the same access (lookup) value. Therefore, when doing lookups, a program should specify either the specific key value or a range of values that includes the uppercase and lowercase combinations of the key. See the *Guide to OpenVMS File Applications* for more information about accessing indexed records. This field corresponds to the FDL attribute CONNECT KEY_OF_REFERENCE.

6.10 RAB$B_KSZ Field

The key size (KSZ) field contains a numeric value equal to the size, in bytes, of the record key pointed to by the RAB$L_KBF field.

Note that the RAB$B_KSZ field has the same offset as the RAB$B_PSZ (prompt buffer size) field but no conflict is presented because the fields are used in mutually exclusive operations.

For indexed files, the size of the key depends on the key data type:

- For string data-type keys, a value from 1 through the size of the key field can be used.

Note

The string data-type keys include STRING, DSTRING, COLLATED, and DCOLLATED keys.

If the specified size is less than the size of the key field, then only the leftmost characters of each key are used for comparison.

- For numeric key data types, a value of 0 causes RMS to use the key data type defined at file creation to determine the key size. A nonzero value is checked against the defined size, and an error is returned if they are not equal.

Note that for DECnet for OpenVMS operations, the RAB$B_KSZ field must be explicitly specified as a nonzero value because the key data type information might not be available to RMS at the local node.

The size of the relative record number of a record in a relative file or a sequential file with fixed-length records is a longword, positive, integer value; therefore, the key size is 4. For relative record numbers, the default value of 0 causes a key size of 4 to be used. For DECnet for OpenVMS operations, however, the RAB$B_KSZ field must be explicitly specified as 4 for relative files.

With indexed files, the size of key values in bytes of an indexed file can be from 1 to 255 bytes.

A program can access indexed file records directly in one of four ways:

- By an exact match
- By an approximate match
- By a generic match
- By a combination of approximate and generic matches

The program specifies the type of match using the RAB$B_KSZ field together with the search options from the RAB$L_ROP field (see Indexed File Options).

- To specify an exact match, do the following:
  - Set a value in the RAB$B_KSZ field equal to the number of bytes in the key.
  - Reset the appropriate search bit, RAB$V_EQNXT or RAB$V_NXT.
  - Specify the search direction by doing the following:
    * For a forward search (toward the end of the file), be sure that the RAB$V_REV bit is not set.
    * For a reverse search (toward the beginning of the file), set the RAB$V_REV bit.

Note

Digital recommends the use of the mask designator (M) in specifying the *equal or next* and the *next* search options because it is more universally applicable, especially for high-level languages. Using mask designators, you can specify multiple options in a single instruction. See the description of the reverse search option in RAB$V_REV.

- To specify an approximate match, do the following:
  - Set a value in the RAB$B_KSZ field equal to the number of bytes in the key.

- Specify the type of match by doing the following:
  * If you want to match on a record having either an equal key value or the next key value (greater for ascending sort order, less for descending sort order) set the RAB$V_EQNXT bit.

Note

Sort order is established in the data type (XAB$B_DTP) field of the associated XABKEY when the file is created.

  * If you want to match on a record having the next value and to ignore equal key values, set the RAB$V_NXT bit.
- Specify the search direction by doing the following:
  * For a forward search (toward the end of the file), be sure that the RAB$V_REV bit is not set.
  * For a reverse search (toward the beginning of the file), set the RAB$V_REV bit.

• To specify a generic match, do the following:
  - Set a value in the RAB$B_KSZ field equal to the number of leading bytes in the key you want to match on.
  - Reset the appropriate search bit, RAB$V_EQNXT or RAB$V_NXT
  - Specify the search direction by doing the following:
    * For a forward search (toward the end of the file), be sure that the RAB$V_REV bit is not set.
    * For a reverse search (toward the beginning of the file), set the RAB$V_REV bit.

• To specify an approximate generic match, do the following:
  - Set a value in the RAB$B_KSZ field equal to the number of leading bytes in the key you want to match on.
  - Specify the type of match by doing the following:
    * If you want to match on a record having a generic key value equal to the specified generic key value or the next generic key value, set the RAB$V_EQNXT bit.
    * If you want to match on a record having the next generic key value but to ignore equal generic key values, set the RAB$V_NXT bit.
  - Specify the search direction by doing the following:
    * For a forward search (toward the end of the file), be sure that the RAB$V_REV bit is not set.
    * For a reverse search (toward the beginning of the file), set the RAB$V_REV bit.

6.11 RAB$B_MBC Field

The multiblock count (MBC) field applies only to sequential disk file operations. This field specifies the number of blocks, in the range of 0 through 127, to be allocated to each process (local) I/O buffer and, correspondingly, the number of blocks of data to be transferred in each I/O unit. If you do not specify this field, or if you specify the value 0, RMS uses the process default for the multiblock count. If the process default is 0, RMS uses the system default, and if the system default is 0, RMS assigns each local buffer one block.

Note

The DCL command SET RMS_DEFAULT is used to set process or system defaults.

When it invokes the Connect service, RMS uses the RAB$B_MBC field to determine the number of blocks in the I/O transfers for this record stream and allocates a local buffer with the appropriate storage capacity. Note that either the RAB$B_MBF (multibuffer count) field or the XAB$_MULTIBUFFER_COUNT XABITM field (see Section 11.6) can be used to allocate multiple local buffers of this size for the record stream.

For shared sequential file operations, the first accessor of the file uses the RAB$B_MBC field to establish the global buffer size for all subsequent accessors.

The use of the RAB$B_MBC field optimizes data throughput for sequential operations, and does not affect the structure of the file. It reduces the number of disk accesses for record operations, resulting in faster program execution. However, the extra buffering increases memory requirements.

Note that the RAB$B_MBC field is not used with block I/O. With multiblocks, the number of blocks in an I/O unit is fixed by the multiblock count, whereas in block I/O operations the number of blocks being transferred is specified by the program.

This field corresponds to the FDL attribute CONNECT MULTIBLOCK_COUNT and it is not supported for DECnet for OpenVMS operations.

6.12 RAB$B_MBF Field

The multibuffer count (MBF) field indicates the number (0 to 255) of process (local) I/O buffers you want RMS to allocate when you invoke the Connect service for a record stream.

Note

You can optionally override the RAB$B_MBF field with the XAB$_MULTIBUFFER_COUNT XABITM (see Section 11.6 for information), which allows you to specify up to 32,767 local buffers.

If this field is not specified or is set to 0, and if the XAB$_MULTIBUFFER_COUNT XABITM is not implemented, RMS uses the process default for the particular file organization and device type. If the process default is also 0, the system default for the particular file organization and device type applies. If the system default is 0, RMS allocates the record stream one local buffer. However, if you specify either the read-ahead option or the write-behind option, RMS allocates at least two buffers.

RMS allocates at least one local buffer for sequential and relative files and at least two buffers for indexed files, unless the file is to be processed with block I/O operations only. Multiple local buffers can be used efficiently to overlap I/O time with compute time, particularly in read-ahead or write-behind processing (see RAB$L_ROP for information on these options), and to increase use of the local buffers (as compared to disk I/O) when performing random processing.

Note that the RAB$B_MBF field is not used when block I/O access is specified with the Open, Create, or Connect services because no local buffers are required.

This field corresponds to the FDL attribute CONNECT MULTIBUFFER_COUNT and it is not supported for DECnet for OpenVMS operations.

6.13 RAB$L_PBF Field

The prompt buffer address (PBF) field points to a character string to be used as a prompt for terminal input. Note that the RAB$L_PBF field has the same offset as the RAB$B_KBF (key buffer address) field but, because the fields are used in mutually exclusive operations, no conflict is presented.

This field contains the symbolic address of the buffer containing the prompt character string. If you select the RAB$V_PMT option of the RAB$L_ROP field when you invoke a Get service, RMS outputs this character string to the terminal before the Get service begins its task.

To perform any carriage control on the terminal, you must insert the appropriate carriage control characters into this character string.

This field is not supported for DECnet for OpenVMS operations; it is ignored.

6.14 RAB$B_PSZ Field

The prompt buffer size (PSZ) field indicates the size of the prompt character string to be used as a terminal prompt. This field contains the size, in bytes, in the range of 0 to 255.

Note that the RAB$B_PSZ field has the same offset as the RAB$B_KSZ (key buffer size) field but no conflict is presented because the fields are used in mutually exclusive operations.

This field is not supported for DECnet for OpenVMS operations; it is ignored.

6.15 RAB$B_RAC Field

The record access mode (RAC) field indicates the method of retrieving or inserting records in the file; that is, whether records are read (or written) sequentially, directly, or by record file address. Only one access method can be specified for any single record operation, but you can change the record access mode between record operations.

The RAB$B_RAC field is a keyword value field in which each record access mode has a symbolic offset. Options are identified using mnemonics. Each access mode in the RAB$B_RAC field has its own symbolic constant. For example, the SEQ (sequential record) access mode has the symbolic constant RAB$C_SEQ.

The RAB$B_RAC field is not applicable to block I/O operations and there is no corresponding FDL attribute for this field.

Options

RAB$C_SEQ
Indicates sequential record access mode (the default); it can be specified with any type of file organization.

Records read from (or written to) sequential files are accessed in chronological order. That is, older records are accessed before newer records.

Records read sequentially from indexed files are accessed by the key of reference according to the key's sort order. Where records have duplicate keys, older records are read before newer records, regardless of the key's sort order.

For example, assume an ascending key indexed file with four 2-byte records (A1, B1, B2, C1), where the key is the first byte of each record. When processed sequentially, the records are encountered in the following order:

```
A1       B1       B2       C1
```

Here, records B1 and B2 have the duplicate key B, but record B1 was inserted chronologically before record B2 and, therefore, is encountered before B2 when the program is reading records sequentially.

If this file is reorganized as a descending-key indexed file, the records are encountered in the following order:

```
C1       B1       B2       A1
```

Note that the chronological order of insertion for the two records with duplicate keys is maintained without regard to sort order.

When records are written sequentially to indexed files, RMS verifies that the key value of each successive record is ordered correctly with respect to the key value in the previously written record. For example, with a descending key of reference, RMS ensures that the key value of the third record written is less than the value of the second record.

RAB$C_KEY
Indicates random access by key. For relative files and sequential files on disk with fixed-length records, random access is by relative record number. Indexed files are accessed directly by specifying the appropriate value for the key of reference.

RAB$C_RFA
Indicates random access by record file address; used for all disk file organizations.

6.16 RAB$L_RBF Field

When a program invokes a service that writes records to a file, the output record buffer address (RBF) field contains the symbolic address of the buffer that holds the record to be written. The record size field (RAB$W_RSZ) specifies the size of the record buffer.

When you invoke the Get or Read service, RMS sets this field to the address of the record just read from the file; you do not need to initialize this field. The record's address can be based on whether the program specifies locate mode (RAB$V_LOC). For locate mode, the address can be within an RMS buffer; otherwise, the address is in the user buffer (RAB$L_UBF) provided by the program.

6.17 RAB$W_RFA Field

The record file address (RFA) field comprises three words that define the physical disk address (not symbolic address) of the current record.

After the successful execution of a record or block I/O operation, RMS sets the RAB$W_RFA field to the address of the record acted on by the operation. This address provides an unambiguous means of directly locating this same record at some later time but is meaningful for disk files *only*.

You can store the contents of the RAB$W_RFA field for future use. When you want to retrieve the record again, merely restore the saved contents of the field, set the record access mode to random by RFA, and invoke a Get or Find service.

The following rules apply to RFA access:

- There are two additional names for portions of this field: RAB$L_RFA0 is the offset of the first of three words; RAB$W_RFA4 is the offset of the last word.
- RFA values remain valid for a record in a sequential file as long as the record is within the space defined by the logical file; that is, until the file is truncated to a point before the record.
- RFA values remain valid for a record in a relative file for the life of the file; that is, until the file is reorganized, using the Convert utility, or deleted.
- With an indexed file, RFA values remain valid until the file is reorganized, using the Convert utility, or deleted. Note that the Convert/Reclaim utility partially reorganizes a file while maintaining RFA values.

6.18 RAB$L_RHB Field

The fixed-length record header buffer (RHB) field contains the symbolic address of the record header buffer, which RMS uses only when processing VFC records.

For a Get service, RMS strips the fixed control area portion of the record and places it in the buffer whose address is specified in this field. For Put or Update services, RMS writes the contents of the specified buffer to the file as the fixed control area portion of the record.

If this field is not specified, RMS assumes the absence of a record header buffer. When no record header buffer exists, the fixed control area is discarded for a Get service, zeroed for a Put service, and left unchanged for an Update service.

The size of the fixed control area is defined in the FAB, using the FAB$B_FSZ field. You must ensure that the buffer size described in the RAB$L_RHB field is equal to the value specified by the FAB$B_FSZ field.

6.19 RAB$L_ROP Field

RAB$L_ROP is the symbolic offset for the RAB's record-processing options (ROP) field. This field specifies which of the various optional record operations are to be implemented for the program.

The ROP is a 32-bit field in which each record-processing option has a corresponding bit assignment to let you specify multiple options (multiple bits can be set), when applicable. Each option has a unique symbolic offset and a unique mask value but you need only specify the appropriate 3-letter mnemonic when coding a function. For example, the end-of-file option is assigned symbolic offset RAB$V_EOF, but to specify the option, you use the following MACRO statement:

```
ROP=EOF
```

The record-processing option bits are never affected by record management services.

This section describes the seven types of record-processing options alphabetically by functional category:

- Connect service options
- Indexed file options
- Miscellaneous options
- Performance options
- Put service options
- Record locking options
- Terminal device options

This field corresponds to the FDL primary attribute CONNECT.

Table 6–2 lists the options alphabetically by category.

Table 6–2 Record Processing Options

Option	Symbolic Offset
Connect Service Options	
Block I/O	RAB$V_BIO
End of file	RAB$V_EOF
Read ahead	RAB$V_RAH
Write behind	RAB$V_WBH
Indexed File Options	
Key greater than or equal	RAB$V_EQNXT (or RAB$V_KGE)
Limit	RAB$V_LIM
Load	RAB$V_LOA
Key greater than	RAB$V_NXT (or RAB$V_KGT)
Reverse search	RAB$V_REV
Miscellaneous Options	
Check for duplicate key	RAB$V_CDK
Timeout	RAB$V_TMO

(continued on next page)

Table 6–2 (Cont.) Record Processing Options

Option	Symbolic Offset
Performance Options	
Asynchronous	RAB$V_ASY
Fast delete	RAB$V_FDL
Locate mode	RAB$V_LOC
Read ahead	RAB$V_RAH
Synchronous status	RAB$V_SYNCSTS
Write behind	RAB$V_WBH
Put Service Options	
Truncate on put	RAB$V_TPT
Update-if	RAB$V_UIF
Record Locking Options	
Do not lock	RAB$V_NLK
Nonexistent record	RAB$V_NXR
Lock for read	RAB$V_REA
Lock for write	RAB$V_RLK
Ignore read lock	RAB$V_RRL
Timeout	RAB$V_TMO
Manual unlock	RAB$V_ULK
Wait to lock	RAB$V_WAT
Terminal Device Options	
Cancel CTRL/O	RAB$V_CCO
Convert	RAB$V_CVT
Extended operation	RAB$V_ETO
Prompt	RAB$V_PMT
Purge type-ahead	RAB$V_PTA
Read, no echo	RAB$V_RNE
Read, no filter	RAB$V_RNF
Timeout	RAB$V_TMO

In the following text, each of the options is described under its symbolic offset. For example, the asynchronous option is described under RAB$V_ASY.

Connect Service Options

RAB$V_BIO Block I/O; specifies that only block I/O operations are to occur, when mixed record I/O and block I/O operations are allowed. This option is meaningful only if the FAB$B_FAC FAB$V_BRO option was specified when the file was opened (by a Create or Open service). When the RAB$V_BIO option is set for the Connect service, only block I/O operations are allowed for this record stream. When the RAB$V_BIO option is clear for the Connect service, only record I/O operations are allowed when accessing a relative or indexed file and mixed (block I/O and record I/O) operations are allowed for sequential files. This option corresponds to the FDL attribute CONNECT BLOCK_IO.

RAB$V_EOF End-of-file; indicates that RMS is to position the record stream to the end of the file for the connect record operation *only*. An application program sets the EOF bit when it requires append access to a file. Append access provides a convenient way of positioning the data entry pointer to point to the end of file in order to append records. Shared append access is supported only for sequential files. Unshared append access is supported for sequential, relative, and indexed file organizations.

This option corresponds to the FDL attribute CONNECT END_OF_FILE.

RABV_RAH, RABV_WBH Read ahead and write behind; see explanation under Performance Options.

Indexed File Options

Note that search operations have limited application with relative files. See the *Guide to OpenVMS File Applications* for details.

Indexed file options can be enabled or disabled during any record operation.

RMS supports four search key options, two forward search and two reverse search, using combinations of the three search bits shown in the following table.

Search Option	Definition
RAB$V_EQNXT	Return the next record having a key value equal to, or greater than, the current key of reference, according to the specified sort order.
RAB$V_NXT	Return the next record having a key value greater than the current key of reference, according to the specified sort order.
RAB$V_REV	When used with either of the first two options, reverses the search direction, that is, searches toward the beginning of the file.

RAB$V_EQNXT When you select the RAB$V_EQNXT option and ascending sort order, RMS returns the next record having a key value equal to or greater than the value specified by the RAB$L_KBF and RAB$B_KSZ fields. If you specify descending sort order, RMS returns the next record that contains a key value equal to or less than the value specified by the RAB$L_KBF and RAB$B_KSZ fields. In either case, the file is searched in the forward direction (toward the end of the file) unless the reverse search key option is selected.

Note

Sort order is established in the data type field (symbolic offset XAB$B_DTP) of the associated XABKEY when the file is created.

If the program specifies a RAB$V_EQNXT search and no record has a key value identical to the specified search value, RMS returns the record with the next key value according to the specified sort order. (See the description of the RAB$V_NXT bit.)

If the program specifies a RAB$V_EQNXT search and a record with a key value identical to the search key is found, RMS returns the record regardless of sort order.

If the program specifies a reverse search by setting the RAB$V_EQNXT bit and the RAB$V_REV bit, RMS reverses the search direction, searching toward the beginning of the file.

Table 6–3 provides the search results for both ascending and descending sort order in a file that has records containing one of three index keys (''B'', ''K'', and ''Q''), and three keys of reference (''A'', ''K'', and ''Z''). For example, assume the current key of reference is ''Z'', the sort order is *ascending*, and the program is looking for the next record having a key that is greater than the key of reference. In this case, RMS returns RNF (record not found) because there are no records that have a key greater than ''Z'' in the file.

Table 6–3 Search Option Results

		Current Key of Reference		
Resultant Search Option	**Sort Order**	**"A"**	**"K"**	**"Z"**
NXT	Ascending	''B''	''Q''	RNF
	Descending	RNF	''B''	''Q''
EQNXT	Ascending	''B''	''K''	RNF
	Descending	RNF	''K''	''Q''
REV + NXT	Ascending	RNF	''B''	''Q''
	Descending	''B''	''Q''	RNF
REV + EQNXT	Ascending	RNF	''K''	''Q''
	Descending	''B''	''K''	RNF

If neither a RAB$V_EQNXT search nor a RAB$V_NXT search is specified, an exact key match is required unless a generic key match is specified. (See the description of the RAB$B_KSZ bit.)

This option corresponds to the FDL CONNECT attribute KEY_GREATER_ EQUAL.

RAB$V_KGE This bit is logically synonymous with the RAB$V_EQNXT option and is described under RAB$V_EQNXT.

RAB$V_KGT This bit is logically synonymous with the RAB$V_NXT option and is described under RAB$V_NXT.

RAB$V_LIM This option is supported for indexed files only. It permits you to use RMS as a limit sensor when accessing a file sequentially. When the RAB$V_LIM bit is set, the key value defined by the RAB$L_KBF and RAB$B_KSZ fields (limit key value) is compared to the key value in each record as it is accessed. When a record is accessed that has a key value different from the limit key value, RMS returns the RMS$_OK_LIM success status code.

This option corresponds to the FDL attribute CONNECT KEY_LIMIT.

RAB$V_LOA This option is supported for indexed files only. It specifies that RMS is to load buckets according to the fill size established at file creation time. The bucket fill size is established in the XAB$W_DFL and XAB$W_IFL fields of the key definition XAB. If the LOA option is not specified, RMS ignores the established bucket fill size; that is, buckets are completely filled.

This option corresponds to the FDL attribute CONNECT FILL_BUCKETS.

RAB$V_NXT If you select the RAB$V_NXT option and ascending sort order, RMS returns the next record that has a key value greater than the value specified by the RAB$L_KBF and RAB$B_KSZ fields. If descending sort order is specified, RMS returns the next record having a key value less than the value specified by the RAB$L_KBF and RAB$B_KSZ fields. In either case, the file is searched in the forward direction (toward the end of the file) unless the reverse search key option is selected.

If you specify neither RAB$V_NXT nor RAB$V_EQNXT, an exact key match is required.

Note

Sort order is established in the data type XAB$B_DTP field of the associated XABKEY when the file is created.

If the program specifies a reverse search by setting the RAB$V_NXT bit and the RAB$V_REV bit, RMS reverses the search direction, searching toward the beginning of the file. This option corresponds to the FDL CONNECT attribute KEY_GREATER_THAN.

RAB$V_REV The RAB$V_REV option reverses the direction of the search when used with either of the forward search key options in random keyed access only. Sequential access still proceeds forward *only*, according to the sort order.

- RAB$V_REV + RAB$V_EQNXT—This combination of search key options searches toward the beginning of the file and returns records having either the same key value as the current key of reference, or a key value less than the current key of reference, according to the specified sort order.
- RAB$V_REV + RAB$V_NXT—This combination of search key options searches toward the beginning of the file and returns records having a key value less than the current key of reference, according to the specified sort order.

When a set of records having duplicate keys is encountered, RMS returns only the first record in the set. An application that needs to access all of the records having duplicate key values requires additional compiler or program logic.

In all cases, you specify search key options symbolically, using the mask bits in the RAB$L_ROP field of the RAB structure. Table 6–4 identifies the results of all combinations of the REV, EQNXT, and NXT bits.

Table 6–4 Keyed Search Combinations

Search Result	REV	EQNXT	NXT
Equal (default)	0	0	0
Greater-than or equal	0	1	0
Greater-than (next)	0	0	1
RMS$_ROP error	0	1	1
RMS$_ROP error	1	0	0
Less-than or equal	1	1	0

(continued on next page)

Table 6–4 (Cont.) Keyed Search Combinations

Search Result	REV	EQNXT	NXT
Less than (previous)	1	0	1
RMS$_ROP error	1	1	1

The reverse-search option includes a performance caching optimization to improve performance when retrieving successive previous records. To take advantage of this feature, specify full key sizes (RAB$B_KSZ) and select the RAB$V_NXT search option.

Miscellaneous Options

RAB$V_CDK During RMS $GET operations, the CDK option allows applications to look ahead for indexed file records with keys that duplicate the current key of reference. If the RAB$V_CDK bit is set, and a record with a duplicate key is detected during a successful $GET operation, RMS returns an alternate success status, RMS$_OK_DUP. Note that this does not guarantee access to the record with the duplicate key. For example, if the record is in a shared file, another accessor may delete the record before your application can access it.

RAB$V_TMO Timeout; in addition to its use for terminals and preventing delays due to record locks (described later), the RAB$V_TMO option serves a special purpose for mailbox devices. If specified in combination with a timeout value of 0 (RAB$B_TMO), Get and Put services to mailbox devices use the IO$M_NOW modifier. Doing so causes the operation to complete immediately, instead of synchronizing with another cooperating writer or reader of the mailbox.

This option corresponds to the FDL attribute CONNECT TIMEOUT_ENABLE and is not supported for DECnet for OpenVMS operations.

See the *OpenVMS I/O User's Reference Manual* for a further discussion of mailboxes.

Performance Options

RAB$V_ASY Asynchronous; indicates that this I/O operation is to be performed asynchronously. When you specify RAB$V_ASY, you pass the address of the RAB as an argument to the AST routine. RMS returns control to your program as soon as an I/O operation is initiated, even though that operation might not yet be completed. This option is normally used with the Wait service to synchronize with operation completion.

The RAB$V_ASY option corresponds to the FDL attribute CONNECT ASYNCHRONOUS.

RAB$V_FDL Fast delete; this option reduces the time required to delete records in indexed files when you are using duplicate alternate keys. The saving in time is achieved by temporarily avoiding the processing needed to eliminate pointers from alternative indexes to the record. However, there is a cost associated with this option. First, the structural linkage from the alternate indexes remains in place but has no utility; therefore, a certain amount of space is wasted. Second, if the program tries to access the deleted record from an alternate index, RMS traverses the linkage, finds the record no longer exists, and then generates a ''nonexistent record'' error message that the program must process.

You should take the fast delete option only if the immediate saving in time is of greater benefit to you than the associated costs in space and overhead.

This option corresponds to the FDL attribute CONNECT FAST_DELETE.

RAB$V_LOC Locate mode; under specified conditions, you have the option of specifying locate mode instead of move mode, which is the default method of buffer handling. In locate mode, your program accesses records directly in an I/O buffer, thus eliminating the overhead of moving records between I/O buffers and application program buffers.

The RAB$V_LOC option activates locate mode. RMS supports the locate mode option for the Get service *only* and ignores requests for the option if one of the following conditions exists:

- Record crosses one or more block boundaries
- The FAB$B_FAC field FAB$V_UPD option is set
- Process-permanent files are accessed indirectly
- Records are compressed (applies to indexed Prolog 3 files only)
- Global buffers
- Multistreaming

In move mode, RMS transfers individual records between I/O buffers and the application's program buffer. When the application program invokes a Get service, RMS reads a block or set of blocks (for sequential files) or a bucket (for relative and indexed files) into an I/O buffer. It then selects the desired record from the I/O buffer and moves it into the program-specified location. Locate mode eliminates the last step by accessing the record directly in the I/O buffer.

This option corresponds to the FDL attribute CONNECT LOCATE_MODE. It is not supported for DECnet for OpenVMS operations, which always use move mode.

RAB$V_RAH Read-ahead; used with multiple local buffers to indicate read-ahead operations. RMS issues I/O requests as soon as possible when a local buffer is needed. When the first local buffer is filled, the I/O operation takes place for the first local buffer as the second local buffer receives the next record; the second local buffer soon becomes filled and the next record is read into the first local buffer as the I/O operation for the second local buffer occurs. The system does not have to wait for I/O completion, which permits an overlapping of input and computing. Read-ahead is ignored for unit record device I/O and is supported only for the nonshared sequential file organization. If the RAB$V_RAH option is specified when the multibuffer count (specified by RAB$B_MBF or by the XAB$_MULTIBUFFER_COUNT XABITM) is 0, two local buffers are allocated to allow multibuffering. If two or more local buffers are specified, multibuffering is allowed regardless of what was specified to the Connect service. Conversely, if a buffer count of 1 is specified, multibuffering is disabled regardless of what was specified to the Connect service.

This option corresponds to the FDL attribute CONNECT READ_AHEAD and it is not supported for DECnet for OpenVMS operations.

RAB$V_SYNCSTS When you select this option, RMS returns the success status RMS$_SYNCH if the requested service completes its task immediately. The most common reason for not completing a task immediately is that the task involves I/O operations. If the service completes synchronously (that is, it has not returned to caller's execution mode prior to completion), RMS returns RMS$_SYNCH as the completion status in R0, stores the true completion status (success or failure) in RAB$L_STS, and does *not* deliver an AST.

The status RMS$_SYNCH is returned in R0 only. Refer to the RAB$L_STS field for the actual success status or failure status of the task.

The RAB$V_SYNCSTS option should be used in conjunction with the RAB$V_ASY option.

RAB$V_WBH Write-behind; used with multiple local buffers. When one local buffer is filled, the next record is written into the next local buffer while the I/O operation takes place for the first local buffer. The system does not have to wait for I/O completion, which allows for an overlapping of computing and output. Write-behind is ignored for unit record devices. This option is implemented only for the nonshared sequential file organization. If the RAB$V_WBH option is specified when the multibuffer count (specified by RAB$B_MBF or by the XAB$_MULTIBUFFER_COUNT XABITM) is 0, two local buffers are allocated to allow multibuffering. If two or more local buffers are specified, multibuffering is allowed regardless of what was specified to the Connect service. Conversely, if you specify a multibuffer count of 1, RMS disables multibuffering regardless of what was specified to the Connect service.

This option corresponds to the FDL attribute CONNECT WRITE_BEHIND and it is not supported for DECnet for OpenVMS operations.

Put Service Options

RAB$V_TPT Truncate-on-put; specifies that a Put or Write service using sequential record access mode can occur at any point in the file, truncating the file at that point. The end-of-file mark is set to the position immediately following the last byte written.

Truncating a file deletes all records beyond the point of truncation. In a shared environment, the application must ensure proper interpretation of a truncate operation. The process must have specified truncate access by setting the FAB$V_TRN option in the FAB$B_FAC field when the file was opened or created.

This option applies only to sequential files and corresponds to the FDL attribute CONNECT TRUNCATE_ON_PUT.

RAB$V_UIF Update-if; indicates that if a Put service is invoked for a record that already exists in the file, the operation is converted to an Update. This option is necessary to overwrite (as opposed to update) an existing record in relative and indexed files. Indexed files using this option must not allow duplicates on the primary key. The process must have specified Update access by setting the FAB$V_UPD option in the FAB$B_FAC field when the file was opened or created.

When using this option with shared files and automatic record locking, you should be aware that the Put service, unlike the Update service, briefly releases record locks until it is determined that an Update should take place. At that point, the record is relocked for the Update operation. Note that during the time the Put operation is being converted into an Update operation, it is possible that another record stream could update or delete the record.

This option corresponds to the FDL attribute CONNECT UPDATE_IF.

Record Locking Options

This section describes the record-processing options related to controlling record locking. Except as noted, these options apply to all file organizations and can be selected for each operation.

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

This option applies only to relative files and it corresponds to the FDL attribute CONNECT NONEXISTENT_RECORD.

RAB$V_REA Lock record for read; specifies that the record is to be locked for a read operation for this process, while allowing other accessors to read the record (but not to modify the record). Use this option only when you do not want the file to be modified by any subsequent activities. Use the RAB$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$V_REA are specified, the RAB$V_REA option is ignored. The RAB$V_NLK option 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 this bit is set for either a Find or Get service request, the record is returned without a lock for all three file organizations. The returned status, however, varies, depending on the file organization:

- The service returns RMS$_SUC for sequential files.
- The service returns alternate success status, RMS$_OK_RRL, for relative and indexed files.

For relative and indexed file applications, the RMS$_OK_RRL status has the added value of indicating when the record is returned without a lock.

This option corresponds to the FDL attribute CONNECT READ_REGARDLESS.

RAB$V_TMO Timeout; specifies that if the RAB$V_WAT option was specified, the RAB$B_TMO field contains the maximum time value, in seconds, to be allowed for a record input wait caused by a locked record. If the timeout period expires and the record is still locked, RMS aborts the record operation with the RMS$_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 and it is not supported for DECnet for OpenVMS operations.

RAB$V_ULK Manual unlocking; prohibits RMS from automatically unlocking records. Instead, once locked (through a Get, Find, or Put service), a record must be specifically unlocked by a Free or Release service. The RAB$V_NLK option takes precedence over the RAB$V_ULK option.

This option corresponds to the FDL attribute CONNECT MANUAL_UNLOCKING.

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

This option corresponds to the FDL attribute CONNECT WAIT_FOR_RECORD.

Terminal Device Options

This section describes the record-processing options related to terminal devices. These options map directly into equivalent modifiers to the QIO function code. For a further discussion of their effects, see the *OpenVMS I/O User's Reference Manual*. These options can be selected for each operation.

RAB$V_CCO Cancel CTRL/O; guarantees that terminal output is not discarded if the operator presses Ctrl/O.

This option corresponds to the FDL attribute CONNECT TT_CANCEL_CONTROL_O and it is not supported for DECnet for OpenVMS operations.

RAB$V_CVT Convert; changes characters to uppercase on a read from a terminal.

This option corresponds to the FDL attribute CONNECT TT_UPCASE_INPUT and it is not supported for DECnet for OpenVMS operations.

RAB$V_ETO Extended terminal operation; indicates presence of a terminal XAB (XABTRM) to describe terminal input using extended terminal characteristics. Note that if this option is specified, all other RAB$L_ROP options specific to terminal devices are ignored (including the RAB$V_TMO option).

This option is not supported for DECnet for OpenVMS operations.

RAB$V_PMT Prompt option; indicates that the contents of the prompt buffer are to be used as a prompt for reading data from a terminal (see RAB$L_PBF field).

This option corresponds to the FDL attribute CONNECT TT_PROMPT and it is not supported for DECnet for OpenVMS operations.

RAB$V_PTA Purge type-ahead; eliminates any information that might be in the type-ahead buffer on a read from a terminal.

This option corresponds to the FDL attribute CONNECT TT_PURGE_TYPE_AHEAD and it is not supported for DECnet for OpenVMS operations.

RAB$V_RNE Read no echo; indicates that input data is not echoed (displayed) on the terminal as it is entered on the keyboard.

This option corresponds to the FDL attribute CONNECT TT_READ_NOECHO and it is not supported for DECnet for OpenVMS operations.

RAB$V_RNF Read no filter; indicates that CTRL/U, CTRL/R, and DELETE are not to be considered control commands on terminal input but are to be passed to the application program.

This option corresponds to the FDL attribute CONNECT TT_READ_NOFILTER and it is not supported for DECnet for OpenVMS operations.

RAB$V_TMO Timeout; for terminal operations, indicates that the content of the RAB$B_TMO field is to be used to determine the number of seconds allowed between characters received during terminal input. If the timeout period expires, RMS returns an error status (see RAB$B_TMO). Other functions of the RAB$V_TMO option are listed under Miscellaneous Options and Record Locking Options.

This option corresponds to the FDL attribute CONNECT TIMEOUT_ENABLE and is not supported for DECnet for OpenVMS operations.

6.20 RAB$W_RSZ Field

The record size (RSZ) field contains the size, in bytes, of the record to which the record buffer address (RAB$L_RBF) field points. For a Write service, the range is 1 through 65,535. For a Put or Update service, the range is 0 to the maximum value shown in Table 4–4. The record size field specifies the size of a record that a Put or Update service can write, or the number of bytes that a Write (block I/O) service can write. The Update service requires the record size field only if the file contains variable-length records or VFC records.

On input from a file, RMS sets this field to indicate the length, in bytes, of the record that a Get service transfers or that a Read service (block I/O) reads.

Three notes apply to this field:

- After a Get service, RMS places the record size in the RAB$W_RSZ field. On a Read service, RMS sets the RAB$W_RSZ field to the number of bytes actually transferred.
- For variable with fixed-length control records, RMS does not include the size of the fixed control area in the RAB$W_RSZ field.
- For block I/O operations, some devices require that an even number of bytes be transferred. No user action is required because RMS automatically rounds the number of transfers up to an even number.

6.21 RAB$L_STS Field

RMS sets the completion status code (STS) field together with the success or failure status codes for a record operation before returning control to your program. For an asynchronous operation that has been initiated but not yet completed, this field is 0. When the operation is complete, the field is updated with the completion status. See Chapter 2 for additional details about signaling RMS status codes. Potential error codes for specific operations are listed with their descriptions in Part III.

6.22 RAB$L_STV Field

The completion status value (STV) field communicates additional completion information to your program, on the basis of the type of operation and the contents of the completion status code field. For additional information on the RAB$L_STS and RAB$L_STV fields, see Part I.

The RAB$L_STV field can be accessed using alternate symbolic offsets; RAB$W_STV0 is the location of the first word and RAB$W_STV2 is the location of the second word within RAB$L_STV.

6.23 RAB$B_TMO Field

The timeout (TMO) field indicates the maximum number of seconds, in the range 0 to 255, that RMS should wait for an operation to conclude. If the operation does not conclude within the specified timeout period, RMS returns an error status code.

To use this field, you must also specify the RAB$V_TMO record-processing option.

For a Get service using a terminal device, this value specifies the number of seconds to wait between the characters being typed. If you specify 0 along with RAB$V_TMO, the current contents of the type-ahead buffer are returned.

When you use a wait-on-record lock (RAB$V_WAT) with a Get, Find, or Put service, this value specifies the maximum number of seconds for RMS to wait for the record to become available.

Note that if the RAB$B_TMO field contains a value of 0 and RAB$V_TMO is set when you invoke either a Get or Put service to a mailbox device, the operation terminates immediately, rather than waiting for another process. For example, if you invoke the Put service to a mailbox device with the RAB$B_TMO field clear, the Put service does not wait for the receiving process to get the record.

This field corresponds to the FDL attribute CONNECT TIMEOUT_PERIOD and it is not supported for DECnet for OpenVMS operations.

6.24 RAB$L_UBF Field

The user record buffer address (UBF) field indicates the location of a record or block buffer.

Note

When you invoke the Get service, RMS takes control of the record buffer and can modify it. RMS returns the record size and only guarantees the contents from where it accessed the record to the completion of the record.

This field contains the symbolic address of a work area (buffer) within your program. The size of this buffer must be defined in the RAB$W_USZ (user record area size) field.

When you invoke a Get service, this field must contain the buffer address, regardless of the record transfer mode (locate or move). This option also applies when you invoke the Read service for block I/O. However, a Put or Write service never needs a user buffer.

6.25 RAB$W_USZ Field

The user record buffer size (USZ) field indicates the length (1 through 65,535 bytes) of the user record buffer (RAB$L_UBF).

The user record buffer should be large enough to contain the largest record in the file. If the buffer is not large enough to accommodate a record obtained by a Get service operation, RMS moves as much of the record as possible into the buffer and returns a warning status code.

The value in this field specifies the transfer length, in bytes, for block I/O operations with a Read service and for a Get service to UDF (undefined) format sequential files.

6.26 RAB$L_XAB Field

The extended attribute block address (XAB) field contains the symbolic address of a XAB control block that you want to use when performing an operation such as a Get service for a terminal device. A value of 0 (the default) indicates no XABs for this record stream.

For certain record operations, you can associate XABs with a RAB to convey additional attributes about an operation. (See Section 1.2.2 for the description of a XAB.)

7
64-Bit Record Access Block (RAB64)

Alpha

The 64-bit record access block (RAB64) is an extension of the RAB (described in Chapter 6) that allows OpenVMS Alpha users to use 64-bit addresses for the following I/O buffers:

- UBF (user record buffer)
- RBF (record buffer)
- RHB (fixed-length record header buffer; fixed portion of VFC record format)
- KBF (key buffer containing the key value for random access)

RAB64 has fields corresponding to all the RAB fields, plus seven additional fields to accommodate 64-bit addressing.

7.1 Summary of RAB64 Fields

The additional fields in the extended RAB64 data structure are summarized at the beginning of Table 7–1 and are described in this chapter. All the other RAB64 fields are exactly like their RAB counterparts described in Chapter 6, unless noted otherwise in this chapter.

Table 7–1 gives the symbolic offset, the size, the RAB cross-reference, and a brief description for each RAB64 field.

Table 7–1 RAB64 Fields

Field Offset	Size (Bytes)	RAB Cross-Reference	Description
Alpha-Only RAB64 Fields Described in this Chapter[1]:			
RAB64$Q_CTX	8	None	User context (64-bit)
RAB64$PQ_KBF	8	None	Key buffer 64-bit address
RAB64$PQ_RBF	8	None	Record buffer 64-bit address
RAB64$PQ_RHB	8	None	Record header buffer 64-bit address
RAB64$Q_RSZ	8	None	Record buffer size
RAB64$PQ_UBF	8	None	User record buffer 64-bit address
RAB64$Q_USZ	8	None	User record buffer size

[1]None of these fields have FDL equivalents.

(continued on next page)

Table 7–1 (Cont.) RAB64 Fields

Field Offset	Size (Bytes)	RAB Cross-Reference	Description
RAB64 Fields Equivalent to RAB Fields Described in Chapter 6:			
RAB64$B_BID [2]	1	See RAB$B_BID in Section 6.2	Block identifier
RAB64$L_BKT	4	See RAB$L_BKT in Section 6.3	Bucket code
RAB64$B_BLN [3]	1	See RAB$B_BLN in Section 6.4	Block length
RAB64$L_CTX	4	See RAB$L_CTX in Section 6.5	User context
RAB64$L_FAB	4	See RAB$L_FAB in Section 6.6	File access block address
RAB64$W_ISI [4]	2	See RAB$W_ISI in Section 6.7	Internal stream identifier
RAB64$L_KBF	4	See RAB$L_KBF in Section 6.8	Key buffer address
RAB64$B_KRF	1	See RAB$B_KRF in Section 6.9	Key of reference
RAB64$B_KSZ	1	See RAB$B_KSZ in Section 6.10	Key size
RAB64$B_MBC	1	See RAB$B_MBC in Section 6.11	Multiblock count
RAB64$B_MBF	1	See RAB$B_MBF in Section 6.12	Multibuffer count
RAB64$L_PBF	4	See RAB$L_PBF in Section 6.13	Prompt buffer address
RAB64$B_PSZ	1	See RAB$B_PSZ in Section 6.14	Prompt buffer size
RAB64$B_RAC	1	See RAB$B_RAC in Section 6.15	Record access mode
RAB64$L_RBF	4	See RAB$L_RBF in Section 6.16	Record buffer address
RAB64$W_RFA	6	See RAB$W_RFA in Section 6.17	Record file address
RAB64$L_RHB	4	See RAB$L_RHB in Section 6.18	Record header buffer address
RAB64$L_ROP	4	See RAB$L_ROP in Section 6.19	Record-processing options
RAB64$W_RSZ	2	See RAB$W_RSZ in Section 6.20	Record buffer size
RAB64$L_STS [4]	4	See RAB$L_STS in Section 6.21	Completion status code
RAB64$L_STV [4]	4	See RAB$L_STV in Section 6.22	Status value
RAB64$W_STV0 [5]	2	See RAB$L_STV in Section 6.22	Low-order word status value
RAB64$W_STV2 [5]	2	See RAB$L_STV in Section 6.22	High-order word status value
RAB64$B_TMO	1	See RAB$B_TMO in Section 6.23	Timeout period
RAB64$L_UBF	4	See RAB$L_UBF in Section 6.24	User record buffer address
RAB64$W_USZ	2	See RAB$W_USZ in Section 6.25	User record buffer size
RAB64$L_XAB	4	See RAB$L_XAB in Section 6.26	Next XAB address

[2]The $RAB64 macro statically initializes this field to identify this control block as a RAB.

[3]The $RAB64 macro statically initializes this field to identify this control block as a RAB64. If the user initializes this field, it must be initialized to RAB64$C_BLN64.

[4]This field cannot be initialized by the $RAB64 macro.

[5]Alternate definition of RAB64$L_STV field.

The RAB64 $PQ_ fields can hold either 64-bit addresses or 32-bit addresses sign-extended to 64-bits. Therefore, you can use these fields in all applications whether or not you are using 64-bit addresses.

If your application already uses RAB fields, only minimal source code changes are required to use 64-bit RMS support. The RAB64 64-bit counterpart to a 32-bit address field is used only if the following two conditions exist:

- The RAB64$B_BLN field has been initialized to RAB64$C_BLN64.
- The 32-bit address cell in the 32-bit portion of the RAB64 contains -1.

Because the RAB64 is an upwardly compatible version of the RAB, RMS allows you to use the RAB64 wherever you can use a RAB. For example, a RAB64 can be used in place of a RAB as the first argument passed to any of the RMS record or block I/O services.

While RMS allows you to use the RAB64 wherever you can use a RAB, some source languages may impose other restrictions. Consult the documentation for your source language for more information.

DECnet for OpenVMS support for RAB64 fields is identical to support described for RAB fields in Chapter 6.

The format and arguments of the $RAB64 macro and the $RAB64_STORE macro are described in Appendix A.

7.2 RAB64$Q_CTX Field

This Alpha-only, 8-byte field is a 64-bit extension of the user context field and serves the same purpose as RAB$L_CTX (see Section 6.5 for details). The only difference between these fields, other than size, is that there is no corresponding FDL attribute for RAB64$Q_CTX.

7.3 RAB64$PQ_KBF Field

This Alpha-only, 8-byte field is a key buffer 64-bit address field that serves the same purpose as RAB$L_KBF (see Section 6.8 for details). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

To use this field:	Put a -1 in this field:	Use this size field:
RAB64$PQ_KBF	RAB64$L_KBF	RAB64$B_KSZ

7.4 RAB64$PQ_RBF Field

This Alpha-only, 8-byte field is a record buffer 64-bit address field that serves the same purpose as RAB$L_RBF (see Section 6.16 for details). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

To use this field:	Put a -1 in this field:	Use this size field:	Put a 0 in this field:
RAB64$PQ_RBF	RAB64$L_RBF	RAB64$Q_RSZ	RAB64$W_RSZ

For most record I/O service requests, there is an RMS internal buffer between the device and the user's data buffer. The one exception occurs with the RMS service $PUT. If the device is a unit record device and it is not being accessed over the network, RMS passes the address of the user record buffer to the $QIO system service. If you inappropriately specify a record buffer allocated in 64-bit address space for a $PUT service to a unit record device that does not support 64-bit address space (for example, a terminal), the $QIO service returns error status SS$_NOT64DEVFUNC. RMS returns error status RMS$_SYS with SS$_NOT64DEVFUNC as the secondary status value in RAB64$L_STV.

7.5 RAB64$PQ_RHB Field

This Alpha-only, 8-byte field is a record header buffer 64-bit address field that serves the same purpose as RAB$L_RHB (see Section 6.18 for details). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

To use this field:	Put a -1 in this field:	Use this size field:
RAB64$PQ_RHB	RAB64$L_RHB	FAB$B_FSZ

7.6 RAB64$Q_RSZ Field

This Alpha-only, 8-byte field is an extended record size field that serves the same purpose as RAB$W_RSZ (see Section 6.20 for details). Use this field whenever you use the RAB64$PQ_RBF field.

The buffer size maximum for the RMS block I/O service $WRITE has been increased from 65535 bytes to $2 * *31 - 1$ bytes for OpenVMS Alpha users, with two exceptions:

- For RMS journaling, a journaled $WRITE service is restricted to the current maximum (65535 minus 99 bytes of journaling overhead). An RSZ error is returned (RAB$L_STS) if the maximum is exceeded.
- Magnetic tape is still limited at the device driver level to 65535 bytes.

7.7 RAB64$PQ_UBF Field

This Alpha-only, 8-byte field is a user record buffer 64-bit address field that serves the same purpose as RAB$L_UBF (see Section 6.24 for details). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

To use this field:	Put a -1 in this field:	Use this size field:	Put a 0 in this field:
RAB64$PQ_UBF	RAB64$L_UBF	RAB64$Q_USZ	RAB64$W_USZ

7.8 RAB64$Q_USZ Field

This Alpha-only, 8-byte field is an extended user record buffer size field that serves the same purpose as RAB$W_USZ (see Section 6.25 for details). Use this field whenever you use the RAB64$PQ_UBF field.

The buffer size maximum for the RMS block I/O service $READ has been increased from 65535 bytes to $2 * *31 - 1$ bytes for OpenVMS Alpha users, with one exception:

- Magnetic tape is still limited at the device driver level to 65535 bytes. ♦

8
Allocation Control XAB (XABALL)

The allocation control XAB (XABALL) provides additional control over file or area space allocation on disk devices in order to optimize performance. In the following descriptions, the terms *file* and *area* are synonymous for sequential and relative files because these file organizations are limited to a single area (Area 0).

8.1 Summary of Fields

When RMS uses a XAB to create or extend an area, the following XABALL fields duplicate and take precedence over associated fields in the related FAB:

- The allocation quantity (ALQ) field, XAB$L_ALQ, overrides FAB$L_ALQ
- The bucket size (BKZ) field, XAB$B_BKZ, overrides FAB$B_BKS
- The default extension quantity (DEQ) field, XAB$W_DEQ, overrides FAB$W_DEQ
- The XAB$V_CBT and XAB$V_CTG options of the allocation options field, XAB$B_AOP, override the FAB$V_CBT and FAB$V_CTG options of the file-processing options field, FAB$L_FOP

When opening a file or displaying a file's attributes, RMS outputs appropriate information to your program using these fields.

The symbolic offset, the size, the FDL equivalent, and a brief description of each XABALL field are presented in Table 8–1.

Table 8–1 XABALL Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
XAB$B_AID	1	AREA *n*	Area identification number
XAB$B_ALN	1	AREA POSITION[1]	Alignment boundary type
XAB$L_ALQ	4	AREA ALLOCATION	Allocation quantity
XAB$B_AOP	1	AREA[1]	Allocation options
XAB$B_BKZ	1	AREA BUCKET_SIZE	Bucket size
XAB$B_BLN[2]	1	None	Block length
XAB$B_COD[2]	1	None	Type code
XAB$W_DEQ	2	AREA EXTENSION	Default extension quantity
XAB$L_LOC	4	AREA POSITION	Location

[1]This field contains options; corresponding FDL equivalents are listed in the description of the field.

[2]This field is statically initialized by the $XABALL macro to identify this control block as a XABALL.

(continued on next page)

Table 8–1 (Cont.) XABALL Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
XAB$L_NXT	4	None	Next XAB address
XAB$W_RFI	6	AREA POSITION FILE_ID or FILE_NAME	Related file identifier
XAB$W_VOL	2	AREA VOLUME	Related volume number

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

The format and arguments of the $XABALL macro and the $XABALL_STORE macro are described in Appendix A.

8.2 XAB$B_AID Fields

RMS uses the area identification number (AID) field to determine which area within a file is supported by this XAB. Note that sequential and relative files are limited to area 0.

The area is identified by a numeric value in the range 0 through 254 (default is 0) and is most appropriate for use with index files having multiple areas allocated.

This field corresponds to the FDL attribute AREA *n*, where *n* indicates the area number.

8.3 XAB$B_ALN Field

The alignment boundary type (ALN) field gives you several options for aligning the allocated area. RMS uses this field in conjunction with the XAB$L_LOC field and with the XAB$W_RFI field to provide precise positioning control of the area or area extension.

The XAB$B_ALN field is a keyword value field in which an alignment boundary option is defined by a symbolic constant value. For example, the cylinder boundary option has a symbolic constant value of XAB$C_CYL.

Note that if no value is defined for this field, RMS assumes the XAB$C_ANY option (no additional positioning control desired). Additional positioning control is not supported for DECnet for OpenVMS operations.

The XAB$B_ALN field corresponds to the FDL attribute AREA POSITION.

Options

XAB$C_ANY

Any allocation; specifies that no positioning control over the area is desired. If this option is selected, the XAB$L_LOC and XAB$W_RFI fields are ignored.

This option corresponds to the FDL attribute AREA POSITION NONE.

XAB$C_CYL

Specifies that the area boundary begin at the cylinder number identified by the location field XAB$L_LOC.

This option corresponds to the FDL attribute AREA POSITION CYLINDER.

XAB$C_LBN
Specifies that the area boundary begin at the logical block number identified by the location field XAB$L_LOC.

This option corresponds to the FDL attribute AREA POSITION LOGICAL.

XAB$C_RFI
This option is used only for extending an area. It specifies that the area extension begin as close as possible to the file identified by the related-file-identification field (XAB$W_RFI), and that the extent begin with the VBN specified by the location field XAB$L_LOC.

This option corresponds to the FDL attribute AREA POSITION FILE_ID or AREA POSITION FILE_NAME. If you try to use this option with DECnet for OpenVMS operations, RMS replaces it with the XAB$C_ANY option.

XAB$C_VBN
This option applies to area extension *only*. It specifies that the area extension begin as close as possible to the virtual block number identified by the location field XAB$L_LOC.

This option corresponds to the FDL attribute AREA POSITION VIRTUAL.

8.4 XAB$L_ALQ Field

The allocation quota field defines the number of blocks to be initially allocated to an area when it is created using a Create service, or the number of blocks to be added to an area when it is explicitly extended using an Extend service (as opposed to automatic extension). In both cases, this field overrides the allocation quantity in the associated FAB (see Chapter 4).

The XAB$L_ALQ field accepts numeric values of up to 4,294,967,295 for initial allocation, but the allocation is limited by the number of blocks available on the device.

When you create a new area using the Create service, RMS interprets the value in the XAB$L_ALQ field as the number of blocks for the area’s initial extent. If the value is 0, RMS allocates a minimum number of blocks depending on the file organization. For example, RMS allocates only the number of blocks needed for the key and area definitions with indexed files.

When you use either the Create or Extend services with indexed files, RMS rounds the allocation quantity value up to the next cluster boundary and returns this value to the program in the XAB$L_ALQ field. RMS uses this value as the extension value when you extend an existing area using the Extend service, unless the program changes the value before invoking the Extend service. Note that the value 0 is not acceptable for extending an area.

8.5 XAB$B_AOP Field

The allocation options (AOP) field lets you specify a particular type of allocation.

This field is a binary options field where one or more options may be selected by setting the appropriate bits. Each option is identified by a symbolic offset and has a mask value; for example, the CBT option has an offset of XAB$V_CBT and a mask value of XAB$M_CBT.

Options

XAB$V_CBT
Contiguous best try; indicates that the allocation or extension should use contiguous blocks, on a ‘‘best effort’’ basis. This option overrides the FAB$L_FOP field FAB$V_CBT option.

This option corresponds to the FDL attribute AREA BEST_TRY_CONTIGUOUS.

XAB$V_CTG
Contiguous; indicates that the initial allocation extension must use contiguous blocks only; the allocation fails if the requested number of contiguous blocks is not available. If this is the initial allocation and only a single area is specified, the file is marked contiguous. This option overrides the FAB$L_FOP field FAB$V_CTG option.

This option corresponds to the FDL attribute AREA CONTIGUOUS.

XAB$V_HRD
Hard; indicates that if the requested alignment cannot be performed, an error will be returned. By default, the allocation is performed as near as possible to the requested alignment.

Note that the XAB$V_HRD option is applicable only to XAB$C_CYL and XAB$C_LBN alignment boundary types, specified by the XAB$B_ALN field.

This option corresponds to the FDL attribute AREA EXACT_POSITIONING.

XAB$V_ONC
On cylinder boundary; indicates that RMS is to begin the allocation on any available cylinder boundary.

This option corresponds to the FDL attribute AREA POSITION ANY_CYLINDER.

8.6 XAB$B_BKZ Field

When RMS creates relative and indexed files, this field specifies bucket size using numeric values ranging from 0 through 63 to represent the number of blocks in a bucket. If this argument is omitted, or if a value of 0 is inserted, RMS uses a default size equal to the minimum number of blocks required to contain a single record. For RMS–11 processing, the bucket size must be less than or equal to 32 blocks.

When calculating the bucket size, you must consider the control information (overhead) associated with each bucket. Note that some record types contain control bytes and to calculate the overhead you must multiply the number of records per bucket by the number of control bytes per record. See the *Guide to OpenVMS File Applications* for more information.

The Edit/FDL utility can be used to calculate efficient bucket sizes for indexed files. (For information about the Edit/FDL utility, see the *OpenVMS Record Management Utilities Reference Manual*.)

When you create a file, RMS uses this field as input to determine the specified bucket size and, upon conclusion, uses the field to output the bucket size. Because relative files are limited to one area, this field specifies the size for all buckets in the file. For indexed files, you can specify a different size for each area using this field in the appropriate XABALL.

When you open an existing file, RMS uses this field to output the bucket size to your program.

The value in this field overrides the contents of the bucket size field of the FAB on a Create service (see Chapter 4).

You can specify multiple areas for a single index by having the XAB$B_IAN and XAB$B_LAN fields in the key definition XAB (XABKEY) indicate that the lowest level of the index and the remainder of that index occupy different areas (defined by different XABALLs). However, the number of blocks per bucket (XAB$B_BKZ value) must be the same for the entire index of a particular key. If multiple areas are specified for an index and if the XAB$B_BKZ values are not the same, RMS returns an error because the values for one index must be the same. However, if you specify the XAB$B_BKZ field for at least one area and use the default (0) for the XAB$B_BKZ field of a different area (or areas) of the same index, RMS uses the largest XAB$B_BKZ value specified among the XABALL control blocks to determine the bucket size for that index.

This field corresponds to the FDL attribute AREA BUCKET_SIZE.

8.7 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABALL, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_ALLLEN (this is done by the $XABALL macro).

8.8 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABALL. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_ALL (this is done by the $XABALL macro).

8.9 XAB$W_DEQ Field

The default extension quantity (DEQ) field specifies the number (0 to 65,535) of blocks to be added when RMS extends the area. If you specify 0, the RMS provides a default extension value.

The value in this field overrides the contents of the default extension quantity field of the FAB (see Chapter 4).

This field corresponds to the FDL attribute AREA EXTENSION.

8.10 XAB$L_LOC Field

The location (LOC) field contains a numeric value that indicates the beginning point for area allocation. RMS refers to the location field when executing a Create or Extend service, but only if the XAB$B_ALN field specifies an alignment option. The way the XAB$L_LOC field is used depends on the value specified for the XAB$B_ALN field (a binary options field). The beginning point for the allocation is determined as follows:

- If the XAB$B_ALN field XAB$C_CYL option is specified, the location number is the cylinder number (0 through the maximum cylinder number on the volume) where the allocation begins.

- If the XAB$B_ALN field XAB$C_LBN option is specified, the location number is the logical block number (0 through the maximum number of blocks on the volume) where the allocation begins.
- If the XAB$B_ALN field XAB$C_VBN or XAB$C_RFI option is specified, the location number is the virtual block number (1 through the maximum number of blocks in the file) where the allocation begins. This applies only to the Extend service. If the number 0 is specified, or if the number is omitted during an Extend service, RMS places the file extension as near to the end of the file as possible.

This field corresponds to the FDL attribute AREA POSITION.

8.11 XAB$L_NXT Field

The next XAB address (NXT) field specifies the symbolic address of the next XAB in the XAB chain. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

8.12 XAB$W_RFI Field

The related file identification (RFI) field allows you to position files or areas of an indexed file close to a specified file.

This field contains the 3-word file identification value of the related file. A value of 0,0,0 (the default) indicates that the current file is to be used. Specifying the XAB$B_ALN field XAB$C_RFI option and specifying the XAB$W_RFI field as 0,0,0 are equivalent to specifying the XAB$B_ALN field XAB$C_VBN option.

You can view the file identification of a file using the DCL command DIRECTORY with the /FULL qualifier.

The file is created or extended as near to the specified related file as possible at the virtual block number specified by the LOC argument.

The XAB$W_RFI field is ignored unless the XAB$B_ALN field is set to XAB$C_RFI. It is also ignored for DECnet for OpenVMS operations.

This field corresponds to the FDL attribute AREA POSITION FILE_ID or AREA POSITION FILE_NAME.

8.13 XAB$W_VOL Field

The relative volume number (VOL) field indicates the specific member of a volume set upon which the file is to be allocated.

This field contains an integer in the range 0 through 255. The default is 0, specifying the ‘‘current’’ member of the volume set.

Note that volume placement will be performed only if an alignment type in the XAB$B_ALN field is either XAB$C_CYL or XAB$C_LBN (you cannot specify XAB$C_VBN or XAB$C_RFI alignment types). If the XAB$B_ALN field contains a value of 0, placement of the file within the volume set will be at the discretion of the system, regardless of the contents of the XAB$W_VOL field.

This field corresponds to the FDL attribute AREA VOLUME.

9
Date and Time XAB (XABDAT)

VAX

For Files–11 B (ODS–2) media, the date and time XAB (XABDAT) block provides extended control of the date and time of the file's creation, revision (update), backup, and expiration. For Files–11 C/D (CD–ROM) media, the date and time XAB (XABDAT) block also provides *read-only* access to the date and time of the file's creation, revision (modification), expiration, effectiveness, and recording.♦

9.1 Summary of Fields

RMS sets certain values for date and time and returns them in XABDAT fields for your inspection. You can override these system-supplied values by using a XABDAT as input to a Create service. Note that date-time values are expressed in either absolute (positive) or delta (negative) format, and several system services are available for date-time conversion and use (see Example B–1 in Appendix B of this manual and the *OpenVMS System Services Reference Manual*).

The symbolic offset, the size, the FDL equivalent (where applicable), and a brief description of each XABDAT field are presented in Table 9–1.

Table 9–1 XABDAT Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
XAB$Q_BDT[1]	8	DATE BACKUP	Backup date and time
XAB$B_BLN[2]	1	None	Block length
XAB$Q_CDT[1]	8	DATE CREATION	Creation date and time
XAB$B_COD[2]	1	None	Type code
XAB$Q_EDT	8	DATE EXPIRATION	Expiration date and time
XAB$L_NXT	4	None	Next XAB address
XAB$Q_RDT[1]	8	DATE REVISION	Revision date and time
XAB$W_RVN[1]	2	FILE REVISION	Revision number
XAB$Q_RCD[1]	8	None	Time and date file was recorded
XAB$Q_EFF[1]	8	None	Time and date file information was effective

[1]This field cannot be initialized by the $XABDAT macro.

[2]This field is statically initialized by the $XABDAT macro to identify this control block as a XABDAT.

The Display service and the Open service always use the XABDAT fields as output. The Create service uses the XABDAT fields as input when it creates a new file. However, when it opens an existing file (see the description of the FAB$V_CIF option in Section 4.17), the Create service also uses the XABDAT fields as output.

No other RMS services use the XABDAT block.

Each XABDAT field is described in the following sections. Unless indicated otherwise, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

The format and arguments for the $XABDAT macro and the $XABDAT_STORE macro are described in Appendix A.

9.2 XAB$Q_BDT Field

The backup date and time (BDT) field contains a 64-bit binary value expressing the date and time when the file was most recently backed up. Note that this field is limited to a granularity of 1 second for remote files.

This field corresponds to the FDL attribute DATE BACKUP.

9.3 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABDAT in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_DATLEN (this is done by the $XABDAT macro).

9.4 XAB$Q_CDT Field

The creation date and time (CDT) field contains a 64-bit binary value expressing the date and time when the file was created. Note that this field is limited to a granularity of 1 second for remote files. If the application program specifies this field as 0 (either explicitly or by default), the Create service uses the current date and time.

This field corresponds to the FDL attribute DATE CREATION.

9.5 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABDAT. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_DAT (this is done by the $XABDAT macro).

9.6 XAB$Q_EDT Field

The expiration date and time (EDT) field contains a 64-bit binary value that indicates the date and time after which a file residing on a disk device may be considered for deletion by the system manager. For files residing on magnetic tape devices, the XAB$Q_EDT field sets the date and time after which you can overwrite the file. Note that this field is limited to a granularity of 1 second for remote files.

This field corresponds to the FDL attribute DATE EXPIRATION.

9.7 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB to be used. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

9.8 XAB$Q_RDT Field

The revision date and time (RDT) field contains a 64-bit binary value representing the date and time when the file was last revised. The Open and Display services use this field to read the revision date and time. The Create service uses this field to set the revision date and time. However, a subsequent Close service overrides the value set by the Create service by using the value in the XAB$Q_RDT field of the XABRDT.

Note

The Close service uses the current date and time when the XAB$Q_RDT field of the XABRDT contains 0 or no value.

If you want to avoid having the Close service override the revision date and time, use the XAB$Q_RDT field in the XABRDT (see Chapter 15) to establish the revision date and time.

If the application program specifies this field as 0 (either explicitly or by default), the Create service uses the current date and time as the revision date and time. Note that this field is limited to a granularity of 1 second for remote files.

This field corresponds to the FDL attribute DATE REVISION.

9.9 XAB$W_RVN Field

The revision number (RVN) field contains a numeric value that indicates the number of times this file was opened for write operations.

This field corresponds to the FDL attribute FILE REVISION.

9.10 XAB$Q_RCD Field (VAX Only)

VAX

On VAX systems, the XAB$Q_RCD (RCD) field contains a 64-bit binary value expressing the date and time that the file was recorded.

This field is applicable only to ISO 9660 files and has no corresponding FDL attribute.♦

9.11 XAB$Q_EFF Field (VAX Only)

VAX

On VAX systems, the XAB$Q_EFF (EFF) field contains a 64-bit binary value expressing the date and time when the file information may be used. If no value is specified in this field, the data may be used immediately.

This field is applicable only to ISO 9660 files and has no corresponding FDL attribute.♦

10

File Header Characteristic XAB (XABFHC)

The file header characteristic XAB (XABFHC) contains file header information that is output by the Open service and the Display service. The Create service can output information in this XAB when it opens an existing file through use of the Create-if option.

The only input field is the longest record length (XAB$W_LRL) field. The Create service uses this field when it creates a sequential file that does not use a fixed-length record format.

Note that, for unshared sequential files or sequential files shared using the FAB$V_UPI option, the values in the end-of-file block (XAB$L_EBK), first free byte in the end-of-file block (XAB$W_FFB), and longest record length (XAB$W_LRL) fields correspond to the values at the time of the last Close or Flush service.

10.1 Summary of Fields

The symbolic offset, size, and a brief description of each RAB field are presented in Table 10–1. Note that many of these fields are also available in the FAB.

Table 10–1 XABFHC Fields

Field Offset	Size (Bytes)	Description
XAB$B_ATR[1]	1	Record attributes; equivalent to FAB$B_RAT
XAB$B_BKZ[1]	1	Bucket size; equivalent to FAB$B_BKS
XAB$B_BLN[2]	1	Block length
XAB$B_COD[2]	1	Type code
XAB$W_DXQ[1]	2	Default file extension quantity; equivalent to FAB$W_DEQ
XAB$L_EBK[1]	4	End-of-file block
XAB$W_FFB[1]	2	First free byte in the end-of-file block
XAB$W_GBC[1]	2	Default global buffer count
XAB$L_HBK[1]	4	Highest virtual block in the file; equivalent to FAB$L_ALQ
XAB$B_HSZ[1]	1	Fixed-length control header size; equivalent to FAB$B_FSZ
XAB$W_LRL[1]	2	Longest record length

[1]This field cannot be initialized by the $XABFHC macro.

[2]This field is statically initialized by the $XABFHC macro to identify this control block as a XABFHC.

(continued on next page)

Table 10–1 (Cont.) XABFHC Fields

Field Offset	Size (Bytes)	Description
XAB$W_MRZ[1]	2	Maximum record size; equivalent to FAB$W_MRS
XAB$L_NXT	4	Next XAB address
XAB$B_RFO[1]	1	File organization and record format; combines FAB$B_RFM and FAB$B_ORG
XAB$L_SBN[1]	4	Starting logical block number for the file if it is contiguous; otherwise this field is 0
XAB$W_VERLIMIT[1]	2	Version limit for the file

[1]This field cannot be initialized by the $XABFHC macro.

There are no FDL equivalents for the XABFHC fields. Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. See the *DECnet for OpenVMS Networking Manual* for information about the support of RMS options for remote file access to other systems.

The format and arguments of the $XABFHC macro and the $XABFHC_STORE macro are described in Appendix A.

10.2 XAB$B_ATR Field

The record attributes (ATR) field indicates the record attributes (special control information) associated with each record in this file. This field is equivalent to the FAB$B_RAT field.

This field is a binary options field where each record attribute has a corresponding bit assignment. Options are identified using mnemonics. Each option in the field has its own symbolic offset and constant value. For example, the CR record attribute has the symbolic offset XAB$V_CR and the mask value XAB$M_CR. The record attribute options are described in the following list.

Options

XAB$V_BLK
Records do not cross block boundaries in sequential files.

XAB$V_CR
Each record is preceded by a line feed and followed by a carriage return.

XAB$V_FTN
Each record contains a FORTRAN (ASA) carriage return in the first byte.

XAB$V_PRN
Print file format.

For more information about the XAB$B_ATR field, refer to the description of the FAB$B_RAT field in Section 4.27.

10.3 XAB$B_BKZ Field

The bucket size (BKZ) field specifies the number of blocks in each bucket of the file. It is equivalent to the FAB$B_BKS (or XAB$B_BKZ) field and is used only for relative or indexed files.

This field contains a numeric value in the range of 0 to 63.

For more information about the XAB$B_BKZ field, refer to the description of the FAB$B_BKS field in Section 4.5 and the description of the XAB$B_BKZ field in Section 8.6.

10.4 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABFHC, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_FHCLEN (this is done by the $XABFHC macro).

10.5 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABFHC. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_FHC (this is done by the $XABFHC macro).

10.6 XAB$W_DXQ Field

The default file extension quantity (DXQ) field specifies the number of blocks to be added when a disk file is extended automatically. This automatic extension occurs whenever your program performs a Put or Write service and the currently allocated file space is exhausted.

This field is equivalent to the FAB$W_DEQ (or XAB$W_DEQ) field; it contains a numeric value in the range 0 through 65,535, which is rounded up to the value of the next cluster boundary.

For more information about the XAB$W_DXQ field, refer to the description of the FAB$W_DEQ field in Section 4.10 and the description of the XAB$W_DEQ field in Section 8.9.

10.7 XAB$L_EBK Field

When you open a file, RMS stores the VBN of the physical block where the next record will be written in the XAB$L_EBK field. For example, assume that a file is allocated five physical blocks and that the last record written to the file is at byte $0FF_{16}$ in the file's second physical block. When your program opens this file, RMS stores the VBN of the second physical block in XAB$L_EBK and it stores 100_{16} in the XAB$W_FFB field.

If the previous block is full when you open the file, RMS stores the first location (000_{16}) of the next block in XAB$W_FFB and the VBN of the next block in XAB$L_EBK. By way of contrast, in a similar situation RMS–11 stores the last byte (200_{16}) of the filled block in the XAB$W_FFB field and the VBN of the filled block in the XAB$L_EBK field.

The XAB$L_EBK field is meaningful for sequential files only.

10.8 XAB$W_FFB Field

The first free byte in the end-of-file block (FFB) field contains the byte location in the end-of-file block where the next record will be written. The XAB$W_FFB field is meaningful for sequential files only.

10.9 XAB$W_GBC Field

The default global buffer count (GBC) field contains the current global buffer count for this file. For more information about the XAB$W_GBC field, refer to the description of the FAB$W_GBC field in Section 4.19.

This field is not supported for DECnet for OpenVMS operations; it is ignored.

10.10 XAB$L_HBK Field

The highest virtual block (HBK) field contains the virtual block number currently allocated to this file. It is equivalent to the FAB$L_ALQ field after a Create, Open, or Display service executes. For sequential files, the difference between XAB$L_HBK and XAB$L_EBK equals the number of blocks in the file available for additional records without extending the file.

10.11 XAB$B_HSZ Field

The fixed-length control header size (HSZ) field indicates the length of the fixed portion for records in the VFC format. It is equivalent to the FAB$B_FSZ field.

This field contains a numeric value (1 to 255) that indicates, in bytes, the size of the fixed-length control area. This field is not applicable to indexed files.

For more information about the XAB$B_HSZ field, refer to the description of the FAB$B_FSZ field in Section 4.18.

10.12 XAB$W_LRL Field

The longest record length (LRL) field contains a numeric value that indicates the longest record currently in the file, in bytes. This value is meaningful for sequential files only. If you specify the XAB$W_MRZ field, the LRL field takes the same value as the XAB$W_MRZ field.

10.13 XAB$W_MRZ Field

The maximum record size (MRZ) field indicates the size of all records in a file with fixed-length records, the maximum size of variable-length records, the maximum size of the data area for variable with fixed-length control records, and the cell size for relative files. It is equivalent to the FAB$W_MRS field.

This field contains a numeric value in the range applicable to the file type and record format (see Table 4–4), in bytes.

For fixed-length records, the value represents the actual size of each record in the file.

For variable-length records, the value represents the size of the largest record that can be written into the file. If the file is not a relative file, a value of 0 is used to suppress record size checking, thus indicating that there is no user limit on record size.

For variable with fixed-length control records, the value includes only the data portion; it does not include the size of the fixed control area.

For more information about the XAB$W_MRZ field, refer to the description of the FAB$W_MRS field in Section 4.24.

10.14 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

10.15 XAB$B_RFO Field

The file organization and record format (RFO) field combines the FAB$B_RFM and FAB$B_ORG fields using an inclusive OR.

The following table lists the record formats.

Record Format	Description
FIX	Fixed length
STM	Stream, delimited by FF, VT, LF, or CR LF
STMCR	Stream, delimited by CR
STMLF	Stream, delimited by LF
UDF	Undefined
VAR	Variable length
VFC	Variable length with fixed control area

The following table lists the file organizations.

File Organization	Description
IDX	Indexed sequential
REL	Relative
SEQ	Sequential

For more information about the XAB$B_RFO field, refer to the description of the FAB$B_ORG field and the FAB$B_RFM field in Section 4.28.

10.16 XAB$L_SBN Field

The starting logical block number (SBN) field contains the starting logical block number for a contiguous file; if the file is not contiguous, this field contains 0.

10.17 XAB$W_VERLIMIT Field

The file version limit (VERLIMIT) field contains the version limit for this file. This value is not available if the file was opened by file ID.

This field is not supported for DECnet for OpenVMS operations; it is ignored.

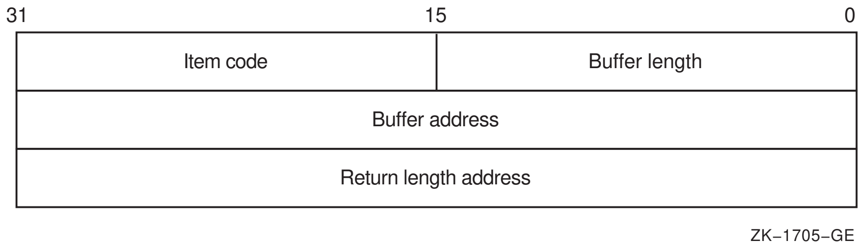
11
Item List XAB (XABITM)

The item list XAB (XABITM) provides a convenient means for using item list information to support RMS functions. Each XABITM points to an item list that includes one or more entries representing either a set function or a sense function that can be passed to the application program by way of the RMS interface.

Because the mode field in a XABITM can be used to either set or sense the items in the list, you cannot use a single XABITM to both set and sense a particular function. However, you may use multiple XABITMs, some for setting functions and other for sensing functions. RMS logically ignores items that are irrelevant to any particular function while acting on any item that is relevant.

Each entry in the item list includes three longwords and a longword 0 terminates the list. See Figure 11–1. Note that RMS does not validate the item list. If the item list is invalid, RMS indicates that the XABITM is not valid by returning the invalid XAB status (RMS$_XAB) in the RAB$L_STS field.

Figure 11–1 Item Descriptor Data Structure



You can store the item list anywhere within process readable address space, but any buffers required by the related function must be in read/write memory.

The format and arguments of the $XABITM macro are defined in Appendix A.

The XABITM control block currently supports the following functions:

- Enhancements to network file access functions
- Passing of file user characteristic information
- Enhancements to RMS performance monitoring functions
- Support for compound documents
- Specifying the number of local buffers
- Expiration date and time suppression

Although the benefits derived from these enhancements are readily apparent, functional details are transparent to most users.

11.1 Summary of Fields

The symbolic offset, the size, and a brief description of each XABITM field are presented in Table 11–1.

Table 11–1 XABITM Fields

Field Offset	Description
XAB$B_BLN[1]	Block length
XAB$B_COD[1]	Type code
XAB$L_ITEMLIST	Item list address
XAB$B_MODE	Set/sense control
XAB$L_NXT	Next XAB address

[1]This field is statically initialized by the $XABITM macro to identify the control block as a XABITM.

11.1.1 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABITM, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_ITMLEN by the $XABITM macro.

11.1.2 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABITM. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_ITM by the $XABITM macro.

11.1.3 XAB$L_ITEMLIST Field

The item list address (ITEMLIST) field contains the symbolic address of the item list.

11.1.4 XAB$B_MODE Field

The item list mode (MODE) field specifies whether the items in the item list can be set or sensed by the program. It contains either the symbolic value XAB$K_SETMODE or the symbolic value XAB$K_SENSEMODE (default).

11.1.5 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB to be used. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

11.2 Network File Access Items (XAB$_NET_... and XAB$_CAP_...)

This section lists and briefly describes the items that support network file access features.

Network items are effectively ignored for local operations. Although the application program may include network items in the XAB chain for the related FAB, RMS does not consider any of the network-specific fields during local processing. Nor does RMS return remote file contents to the application program during local file processing.

Table 11–2 lists the entries in the XABITM item list relating to network file access features together with the buffer size required to store the data and a brief functional description. Note that although the application program can sense all of the item values from the RMS interface, it can set only the following item values:

- XAB$_NET_BLOCK_COUNT
- XAB$_NET_EXTPROT
- XAB$_NET_LINK_TIMEOUT
- XAB$_NET_LINK_CACHE_ENABLE
- XAB$_NET_DATA_CRC_ENABLE

Table 11–2 XABITM Item List

Item Value	Description
XAB$_NET_BUFFER_SIZE	The size of the buffer allocated for DAP messages between the local and remote node is a negotiated value that is decided by DAP. This informational item returns the actual buffer size, in bytes, allocated for DAP messages. The buffer size is slightly larger than the limit specified for the records being transferred. A 4-byte buffer is needed to store the net buffer size.
XAB$_NET_BLOCK_COUNT	This is the value in blocks that the local node wants to use for buffering messages between itself and the remote node. DAP tries to allocate this buffer space at the local node; however, if the maximum buffer size at the remote node is smaller, DAP allocates buffer space based on the smaller value. When the remote system incorporates the file access listener, it allows any size buffer up to 32,767 bytes. The minimum buffer size for task-to-task network operations is 4096 bytes. A 4-byte buffer is needed to store the net block count.
XAB$_NET_REMOTE_SYSTEM	This informational item returns the identity of the remote operating system. A 4-byte buffer is needed to store the symbolic constants representing the remote system identities listed in the following table:

Symbolic Constant	Operating System
XAB$K_RT11	RT–11
XAB$K_RSTS	RSTS/E
XAB$K_RSX11S	RSX–11S
XAB$K_RSX11M	RSX–11M
XAB$K_RSX11D	RSX–11D
XAB$K_IAS	IAS
XAB$K_VAXVMS	VMS

(continued on next page)

Table 11–2 (Cont.) XABITM Item List

Item Value	Description
	XAB$K_TOPS10 — TOPS–10 XAB$K_TOPS20 — TOPS–20 XAB$K_RSX11MP — RSX–11M–PLUS XAB$K_P_OS — P/OS XAB$K_VAXELN — VAXELN XAB$K_MS_DOS — MS–DOS XAB$K_ULTRIX_32 — ULTRIX–32 XAB$K_SNA_OS — SNA gateway to IBM
XAB$_NET_REMOTE_FILE_SYSTEM	This informational item returns the identity of the remote file system. A 4-byte buffer is needed to store the symbolic constants listed in the following table:

Symbolic Constant	File System
XAB$K_RMS11	RMS–11
XAB$K_RMS20	RMS–20
XAB$K_RMS32	RMS–32
XAB$K_FCS11	FCS–11
XAB$K_RT11FS	RT–11
XAB$K_NO_FS	No file system present
XAB$K_TOPS20FS	TOPS–20
XAB$K_TOPS10FS	TOPS–10
XAB$K_RMS32S	RMS–32 subset (VAXELN)
XAB$K_MS_DOSFS	MS–DOS
XAB$K_ULTRIX32_FS	ULTRIX–32
XAB$K_SNA_FS	SNA gateway to IBM

Item Value	Description
XAB$_NET_EXTPROT	This item permits the application program to specify or to sense the extended file protection that is likely to be mapped to a protection subset supported by the remote system. An 8-byte buffer is needed to store protection mask specification. The application program implements extended file protection as part of either a Create or Close service by specifying the appropriate protection mask in the related subfield:

(continued on next page)

Table 11–2 (Cont.) XABITM Item List

Item Value	Description

Subfield	Protection
XAB$W_SYSTEM_ACC	System access
XAB$W_OWNER_ACC	Owner access
XAB$W_GROUP_ACC	Group access
XAB$W_WORLD_ACC	World access

Each of the protection mask fields provides the following mask values for further defining access:

Mask Value	Protection Function
XAB$M_RED_ACC	Deny read access
XAB$M_WRT_ACC	Deny write access
XAB$M_EXE_ACC	Deny execute access
XAB$M_DLT_ACC	Deny delete access
XAB$M_APP_ACC	Deny append access
XAB$M_DIR_ACC	Deny directory access
XAB$M_UPD_ACC	Deny update access
XAB$M_CHG_ACC	Deny change protection access
XAB$M_EXT_ACC	Deny extend access

Note that not all systems support all of the protection mask fields.

XAB$_NET_SYSCAP_LOCAL

This informational item permits the application program to read the network capabilities of the local system by returning symbolic bit vector values. An 8-byte buffer is needed to store the symbolic bit vector values.

See Table 11–3 for a description of the network capabilities bit vectors used by the local and remote systems.

XAB$_NET_SYSCAP_REMOTE

This informational item permits the application program to read the network capabilities of the remote system by returning symbolic bit vector values. An 8-byte buffer is needed to store the symbolic bit vector values.

See Table 11–3 for a description of the network capabilities bit vectors used by the local and remote systems.

XAB$_NET_DAPVER_LOCAL

This informational item returns the version of DAP on the local system using five symbolic bytes, thus requiring a 5-byte buffer:

Symbolic Byte	Version Information
XAB$B_VER_DAP	DAP protocol version
XAB$B_VER_ECO	DAP protocol ECO level

(continued on next page)

Table 11–2 (Cont.) XABITM Item List

Item Value	Description	
	XAB$B_VER_CUS	Customer modification level of DAP protocol; set to 0 by Digital
	XAB$B_VER_DSV	Digital software version (release number)
	XAB$B_VER_CSV	Customer software version number; set to 0 by Digital
XAB$_NET_DAPVER_REMOTE	This informational item returns the version of DAP on the remote system using five symbolic bytes:	
	Symbolic Byte	**Version Information**
	XAB$B_VER_DAP	DAP protocol version
	XAB$B_VER_ECO	DAP protocol ECO level
	XAB$B_VER_CUS	Customer modification level of DAP protocol; set to 0 by Digital
	XAB$B_VER_DSV	Digital software version (release number)
	XAB$B_VER_CSV	Customer software version number; set to 0 by Digital
XAB$_NET_LINK_TIMEOUT	This item permits the application program to set the timeout interval for logical link caching. The setting is passed as the number of seconds used to cache the logical link. A zero (0) setting enables caching until image rundown. The default interval is 30 seconds. A 4-byte buffer is needed to store the timeout interval value.	
XAB$_NET_DATA_CRC_ENABLE	This item allows the application program to enable cyclic redundancy checking at the DAP level. The symbolic value XAB$K_ENABLE enables CRC checking at the DAP level (the default state); the symbolic value XAB$K_DISABLE disables CRC checking at the DAP level.	
XAB$_NET_LINK_CACHE_ENABLE	This item is used to enable or to disable logical link caching. The symbolic value XAB$K_ENABLE enables link caching (the default state); the symbolic value XAB$K_DISABLE disables link caching. A 4-byte buffer is required.	

The system capabilities supported by various DAP implementations are described using a vector of bits wherein a bit is set if the corresponding capability is supported. Any attempt to implement a feature at the local node that is not supported at the remote node is treated as a protocol error. Table 11–3 describes the bit vectors that RMS uses to return the networking capabilities for both the local and remote nodes to the calling program.

Table 11–3 System Networking Capabilities

Bit Value	Capability
XAB$V_CAP_FILALL	Allocation of space at file creation
XAB$V_CAP_SEQORG	Sequential file organization
XAB$V_CAP_RELORG	Relative file organization
XAB$V_CAP_EXTEND	Manual file extension
XAB$V_CAP_SEQFIL	Sequential file access (file transfer mode)
XAB$V_CAP_RANRRN	Random access by relative record number
XAB$V_CAP_RANVBN	Random access by virtual block number
XAB$V_CAP_RANKEY	Random access by key value
XAB$V_CAP_RANRFA	Random access by record file address
XAB$V_CAP_IDXORG	Multikeyed indexed file organization
XAB$V_CAP_SWMODE	Dynamic switching of access modes
XAB$V_CAP_APPEND	Records appended to end of file
XAB$V_CAP_SUBMIT	Command file submission/execution
XAB$V_CAP_MDS	Multiple data streams for each file
XAB$V_CAP_DISPLAY	Display of file attributes on request
XAB$V_CAP_MSGBLK	Blocking of DAP messages up to response (less than 256 bytes)
XAB$V_CAP_UNRBLK	Unrestricted blocking of DAP messages
XAB$V_CAP_BIGBLK	Blocking of DAP messages up to response (greater than or equal to 256 bytes)
XAB$V_CAP_DAPCRC	DAP message CRC checksum
XAB$V_CAP_KEYXAB	Key definition XAB message
XAB$V_CAP_ALLXAB	Allocation XAB message
XAB$V_CAP_SUMXAB	Summary XAB message
XAB$V_CAP_DIRECTORY	Directory list operation
XAB$V_CAP_TIMXAB	Date and time XAB message
XAB$V_CAP_PROXAB	File protection XAB message
XAB$V_CAP_FOPSPL	Spool file on Close FOP option
XAB$V_CAP_FOPSCF	Submit command file on Close FOP option
XAB$V_CAP_FOPDLT	Delete file on Close FOP option
XAB$V_CAP_SEQRAC	Sequential record access
XAB$V_CAP_BITOPT	Bit count option in the FLAGS field
XAB$V_CAP_WARNING	Warning status message and error recovery message exchange
XAB$V_CAP_RENAME	File rename operation
XAB$V_CAP_WILDCARD	Wildcard operations (excluding directory)
XAB$V_CAP_GNGOPT	Go/Nogo option in the ACCOPT field
XAB$V_CAP_NAMMSG	Name message
XAB$V_CAP_SEGMSG	Segmented DAP messages

(continued on next page)

Table 11–3 (Cont.) System Networking Capabilities

Bit Value	Capability
XAB$V_CAP_CHGATTCLS	Changing file attributes on Close using ATT message
XAB$V_CAP_CHGTIMCLS	Changing file attributes on Close using TIM message
XAB$V_CAP_CHGPROCLS	Changing file attributes on Close using PRO message
XAB$V_CAP_CHGNAMCLS	Changing file attributes on Close using NAM message
XAB$V_CAP_MODATTCRE	Modified attributes returned when file is created
XAB$V_CAP_NAM3PART	Three-part name message format in DISPLAY field of both Access and Control messages
XAB$V_CAP_CHGATTREN	Changing file attributes on Rename using ATT message
XAB$V_CAP_CHGTIMREN	Changing file attributes on Rename using TIM message
XAB$V_CAP_CHGPROREN	Changing file attributes on Rename using PRO message
XAB$V_CAP_CTLBLKCNT	BLKCNT field in Control message
XAB$V_CAP_OCTALVER	Octal version numbers only in file specifications

11.3 File User Characteristics Items (XAB$_UCHAR_...)

This section describes the use of the XABITM to pass ODS-2 file information outside the RMS interface level in contrast to the file header information (see Chapter 10) which is used internally by the record management system. The information in this XABITM is independent of any record management system or database management system.

Table 11–4 lists the file user information you can include in a XABITM. You can sense all of the functions listed in the table using either $OPEN or $DISPLAY, but you can set only the following functions using a XABITM:

- XAB$_UCHAR_CONTIGB
- XAB$_UCHAR_ERASE
- XAB$_UCHAR_LOCKED
- XAB$_UCHAR_NOBACKUP
- XAB$_UCHAR_NOMOVE
- XAB$_UCHAR_NOSHELVABLE
- XAB$_UCHAR_READCHECK
- XAB$_UCHAR_WRITECHECK

You can only set these functions when you use the $CREATE operation for a new file. The buffer associated with the item code must contain the symbolic value XAB$K_ENABLE and the XAB$B_MODE field must contain the symbolic value XAB$K_SETMODE.

You cannot enable or disable these functions for an existing file using the XABITM interface; you can only enable or disable these functions for an existing file from the DCL interface using the SET FILE command. See the *OpenVMS DCL Dictionary* for details.

Table 11–4 File User Characteristics

Item	Description
XAB$_UCHAR_BADACL	File's ACL is corrupt.
XAB$_UCHAR_BADBLOCK	File contains bad blocks.
XAB$_UCHAR_CONTIG	File is contiguous.
XAB$_UCHAR_CONTIGB	Keep the file as contiguous as possible.
XAB$_UCHAR_DIRECTORY	File is a directory.
XAB$_UCHAR_ERASE	Erase the file's contents before deleting it.
XAB$_UCHAR_LOCKED	File is deaccess-locked.
XAB$_UCHAR_MARKDEL	File is marked for deletion.
XAB$_UCHAR_NOBACKUP	Do not back up the file.
XAB$_UCHAR_NOCHARGE	Do not charge file space.
XAB$_UCHAR_NOMOVE	Disable movefile operations on the file.
XAB$_UCHAR_NOSHELVABLE	File is not shelvable.
XAB$_UCHAR_READCHECK	Verify read operations to the file.
XAB$_UCHAR_SHELVED	File is shelved.
XAB$_UCHAR_SPOOL	File is an intermediate spool file.
XAB$_UCHAR_WASCONTIG	File was (and should be) contiguous.
XAB$_UCHAR_WRITEBACK	File may be write-back cached.
XAB$_UCHAR_WRITECHECK	Verify write operations to the file.

11.4 RMS Performance Monitoring (XAB$_STAT_ENABLE)

This section describes the implementation of performance monitoring from the RMS interface using a XABITM.

To explicitly obtain performance statistics for a file through the RMS interface, the application program enables the statistics function using the XAB$_STAT_ENABLE item. This item may be used with a $OPEN or $DISPLAY operation to sense the statistics monitoring state. You can only set the statistics function when you use a $CREATE operation to create a new file. The buffer associated with the item code must contain the symbolic value XAB$K_ENABLE and the XAB$B_MODE field must contain the symbolic value XAB$K_SETMODE.

You cannot enable or disable this function for an existing file using the XABITM interface. You can only enable or disable this function for an existing file from the DCL interface using the SET FILE command. See the *OpenVMS DCL Dictionary* for details.

For details about using the Monitor utility for gathering performance statistics, see the *OpenVMS System Manager's Manual*.

Example 11–1 illustrates the use of XABITM to enable statistics monitoring.

Example 11–1 Using XABITM to Enable RMS Statistics

```
   .
   .
   .
ITEMLIST : BLOCK [ITM$S_ITEM+4, BYTE]
  INITIAL( REP (ITM$S_ITEM+4) OF (0) ),

ITEM_XAB : $XABITM( mode = SETMODE,
      itemlist = ITEMLIST ),

ITEM_BUFFER : LONG INITIAL ( XAB$K_ENABLE);
FILE_FAB : $FAB(
   .
   .
   .
                           XAB = ITEM_XAB,
   .
   .
   .
                                       );
ITEMLIST[ITM$W_ITMCOD] = XAB$_STAT_ENABLE;
ITEMLIST[ITM$W_BUFSIZ] = 4;
ITEMLIST[ITM$L_BUFADR] = ITEM_BUFFER;
$CREATE( fab = FILE_FAB );
   .
   .
   .
```

11.5 Compound Document Support (XAB$_..._SEMANTICS)

The term *compound documents* refers to files that can contain a number of integrated components including text, graphics, and scanned images. To support the use of text in compound documents, RMS implements the file attribute, *stored semantics*. The value of the stored semantics attribute is called the file *tag*, and it specifies how file data is to be interpreted.

RMS support for compound documents requires that compound document files be tagged. You can tag a file from the RMS interface by using the Create service in conjunction with a $XABITM macro, and you can sense the tagged status of a file using a $XABITM macro. Tagged file support involves the use of the two item codes shown in Table 11–5. Each of these item codes requires buffers up to 64 bytes long.

Table 11–5 Tag Support Item Codes

Item	Function
XAB$_STORED_SEMANTICS	Defines the file semantics established when the file is created
XAB$_ACCESS_SEMANTICS	Defines the file semantics desired by the accessing program

The entries XAB$_STORED_SEMANTICS and XAB$_ACCESS_SEMANTICS in the item list can represent either a control (set) function or a monitor (sense) function that can be passed to RMS from the application program by way of the RMS interface. You can use the symbolic value XAB$K_SEMANTICS_MAX_LEN, representing the tag length, to allocate buffer space for sensing and setting stored semantics for the file.

Within any one $XABITM, you can activate either the set function or the sense function for the XAB$_STORED_SEMANTICS and XAB$_ACCESS_SEMANTICS items because a common field (XAB$B_MODE) determines which function is active. If you want to activate both the set function and the sense function for either or both items, you must use two $XABITM control blocks, one for setting the functions and one for sensing the functions.

11.6 Specifying the Number of Local Buffers (XAB$_MULTIBUFFER_COUNT)

This section describes how to use the item list XAB to specify up to 32,767 local buffers. Prior to implementation of the XAB$_MULTIBUFFER_COUNT XABITM, you could only specify up to 127 local buffers for a record stream from the RMS interface using the RAB multibuffer count field (RAB$B_MBF). When you use the multibuffer count XABITM, the value specified overrides any value that resides in the RAB$_MBF for the related record stream.

The XAB$_MULTIBUFFER_COUNT XABITM requires a 4-byte buffer to store the value specifying the number of local buffers. Before you increase the size of the local buffer pool, your current memory management parameters should be considered because excessively large buffer pools can introduce additional paging that reduces performance.

You cannot sense the value stored in the XAB$_MULTIBUFFER_COUNT XABITM, and any attempt to sense this value leaves the user buffer unchanged.

11.7 Expiration Date and Time Suppression

The file system, in conjunction with parameters established through the DCL interface (see the SET VOLUME command in the *OpenVMS DCL Dictionary*), gives users the capability to determine whether the contents of a data file have grown stale and whether the file is a candidate for less costly and less accessible storage, typically archived tape.

The file system determines whether a file has grown stale by evaluating the Expiration Date and Time flag. This value should reflect real file activity; that is, it should indicate when a file is no longer being actively used for informational purposes. The flag should not be affected by maintenance functions or for any function that does not involve data access.

This capability is also available to all user application programs through the RMS interface using the XAB$_NORECORD XABITM.

11.7.1 XAB$_NORECORD XABITM

When an application program reads data from a disk file or writes data to a disk file, the $CLOSE service updates the Expiration Date and Time value to the current date and time. This effectively pushes back the expiration date and time to reflect user interest in the file.

When the user program accesses a file for maintenance or monitoring purposes, it should use the XAB$_NORECORD XABITM as an input to the appropriate file service to inhibit the update of the Expiration Date and Time field and thereby maintain the true expiration status of the file. For example, the DCL command DIRECTORY/FULL uses the XAB$_NORECORD XABITM when it opens files to access prolog data containing key information. In this case, DIRECTORY displays prolog information but does not display or modify the file data and therefore should not modify the Expiration Date and Time.

Maintenance utilities should also consider using this XABITM. For example, a disk defragmentation utility should not affect the expiration status of a disk file because the file is not accessed for informational purposes, but rather for maintenance purposes.

The XAB$_NORECORD XABITM uses a 4-byte buffer to set the NORECORD flag to logic 1 using the symbol XAB$_ENABLE. Any other value in this XABITM buffer returns an RMS$_XAB error. An application cannot disable this option because the ODS-2 ACP does not support disabling the option when it is selected on a $OPEN or $CREATE.

11.7.2 Application

The XAB$_NORECORD function can be enabled on input to the $CLOSE, $OPEN, and $CREATE services. However, Digital recommends using the XAB$_NORECORD XABITM with the $OPEN service instead of with the $CLOSE service in order to insure that the Expiration Date and Time flag is updated should the file deaccess or should a close occur because of process deletion or RMS rundown.

The XAB$_NORECORD XABITM can also be used when the $CREATE service opens an existing file through the Create-if option and the user does not want to change the Expiration Date. When the XAB$_NORECORD XABITM is used on a $CREATE that creates a file, it disables the update on the subsequent $CLOSE but does not prevent initialization of the Expiration Date and Time on the file creation in the ACP.

An application typically senses the XAB$_NORECORD XABITM to determine if the XABITM was specified on a previous $OPEN or $CREATE option or if it is specified by the current RMS operation. The XAB$_NORECORD can be sensed on output from RMS for the $OPEN, $CREATE, $DISPLAY, and $CLOSE services.

12
Journaling XAB (XABJNL)

The journaling XAB (XABJNL) control block supports file journaling operations. See the *RMS Journaling for OpenVMS Manual* for details.

13
Key Definition XAB (XABKEY)

You must provide a key definition XAB (XABKEY) for each key in an indexed file in order to define the key's characteristics. Before you create an indexed file, you must establish the contents of the XABKEY fields for the primary key and for each alternate key.

When you invoke an Open or Display service for an existing indexed file, you can use XABKEYs if you want to provide your program with one or more of the key definitions specified when the file was created. Alternatively, the summary XAB (see Chapter 17) provides the number of keys, the number of allocated areas, and the prolog version assigned to the file.

13.1 Summary of Fields

Table 13–1 lists the symbolic offset, the size, the FDL equivalent, and a brief description of each XABKEY field.

Table 13–1 XABKEY Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
XAB$B_BLN[1]	1	None	Block length
XAB$B_COD[1]	1	None	Type code
XAB$L_COLNAM	4	None	Collating sequence name
XAB$L_COLSIZ	4	None	Collating sequence table size
XAB$L_COLTBL	4	COLLATING_SEQUENCE	Collating sequence table address
XAB$B_DAN	1	KEY DATA_AREA	Data bucket area number
XAB$B_DBS[2]	1	None	Data bucket size
XAB$W_DFL	2	KEY DATA_FILL	Data bucket fill size
XAB$B_DTP	1	KEY TYPE[3]	Data type of the key
XAB$L_DVB[2]	4	None	First data bucket virtual block number
XAB$B_FLG	1	KEY[3]	Key options flag
XAB$B_IAN	1	KEY INDEX_AREA	Index bucket area number
XAB$B_IBS[2]	1	None	Index bucket size
XAB$W_IFL	2	KEY INDEX_FILL	Index bucket file size
XAB$L_KNM	4	KEY NAME	Key name buffer address

[1]This field is statically initialized by the $XABKEY macro to identify this control block as a XABKEY.

[2]This field cannot be initialized by the $XABKEY macro.

[3]This field contains options; corresponding FDL equivalents are listed in the description of the field.

(continued on next page)

Table 13–1 (Cont.) XABKEY Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
XAB$B_LAN	1	KEY LEVEL1_INDEX_AREA	Lowest level of index area number
XAB$B_LVL[2]	1	None	Level of root bucket
XAB$W_MRL[2]	2	None	Minimum record length
XAB$B_NSG[2]	1	None	Number of key segments
XAB$B_NUL	1	KEY NULL_VALUE	Null key value
XAB$L_NXT	4	None	Next XAB address
XAB$W_POSn	2	KEY POSITION and SEGn_POSITION	Key position, XAB$W_POS0 to XAB$W_POS7
XAB$B_PROLOG	1	KEY PROLOG	Prolog level
XAB$B_REF[4]	1	KEY n	Key of reference
XAB$L_RVB[2]	4	None	Root bucket virtual block number
XAB$B_SIZn	1	KEY LENGTH and SEGn_LENGTH	Key size XAB$B_SIZ0 to XAB$B_SIZ7
XAB$B_TKS[2]	1	None	Total key field size

[2]This field cannot be initialized by the $XABKEY macro.

[4]For BLISS-32, this field is designated XAB$B_KREF.

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS nodes. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

The format and arguments of the $XABKEY macro and the $XABKEY_STORE macro are defined in Appendix A.

13.2 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABKEY, in bytes. Once set, this field must not be altered unless the control block is no longer needed. The $XABKEY macro initializes the XAB$B_BLN field to the symbolic value XAB$C_KEYLEN.

13.3 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABKEY. Once set, this field must not be altered unless the control block is no longer needed. The $XABKEY macro initializes the XAB$B_COD field to the symbolic value XAB$C_KEY.

13.4 XAB$L_COLNAM Field

When you invoke the Display service, RMS uses this field to return a pointer to a memory buffer containing the name of the collating sequence for this key.

The name buffer is in the form of an ASCII counted string where the first byte indicates the length of the name and the remaining bytes are the ASCII representation of the name itself. The maximum length of the buffer is 32 bytes, 1 byte for the count and 31 bytes for the name.

13.5 XAB$L_COLSIZ Field

When you invoke the Display service, RMS returns the size, in bytes, of the collating sequence used with this key to this field.

13.6 XAB$L_COLTBL Field

RMS provides you with a way to use alternative (non-ASCII) collating sequences with indexed file keys. You can define a collating sequence for each key of reference, yielding, for example, a file sorted in German by one key, French by another key, and so forth.

This feature is based on the National Character Set utility, which permits you to define alternative collating sequences for special characters and to establish and maintain a library of collating sequences. This eliminates having to redefine an alternative collating sequence when the application requires it. See the *OpenVMS National Character Set Utility Manual* for details.

Note

Key compression and index compression are not permitted with collating keys.

To access an alternative collating sequence for a key, enter the symbolic address of the appropriate collating table in the XAB$L_COLTBL field. For example, you might enter the following:

```
DST_KEY0:
        $XABKEY
                .
                .
                .
                COLTBL=FRENCH, - ;symbolic address of French collating table
                .
                .
                .
```

RMS responds by storing the specified collating table in the initial blocks of the indexed file immediately following the area descriptors. Collating tables are typically about one block long.

When you invoke the Display or the Open service, RMS returns the address of the collating table in this field.

This field corresponds to the FDL attribute COLLATING_SEQUENCE.

13.7 XAB$B_DAN Field

The data bucket area number (DAN) field contains a numeric value that identifies the area where the data buckets for this key reside. The number reflects the value in the XAB$B_AID field of the XABALL for this XAB chain. The numeric value may range from 0 through 254, but the default is 0; that is, area 0.

When you create a new indexed file or when you use allocation XABs to define areas (see Chapter 8), you must specify a value for this field to identify the file area where the data buckets are to reside.

When a XABKEY describes the primary key, the data level of the index consists of buckets that contain the actual data records of the file. However, when the key definition describes an alternate key, the data level of the index consists of buckets in which RMS maintains pointers to the actual data records.

The XAB$B_DAN field corresponds to the FDL attribute KEY DATA_AREA.

13.8 XAB$B_DBS Field

After an Open or Display service, the data bucket size (DBS) field contains the size of the data level (level 0) buckets, in virtual blocks, for the key described by the XAB.

13.9 XAB$W_DFL Field

The data bucket fill size (DFL) field contains a numeric value that indicates the maximum number of bytes (of data) in a data bucket. The largest possible fill size is the bucket size, in blocks, multiplied by 512. The default value is 0, which is interpreted as the maximum available space (that is, no unused space). If the specified size is not 0, but is less than one-half of the bucket size (in bytes), then the fill size used is one-half of the bucket size.

When you create an indexed file, you use this field to specify the number of bytes of data you want in each data level bucket. If you specify a value that is less than the actual bucket size, the data buckets contain some amount of free space. At run time, RMS uses the fill size specified when the file was created only if the RAB$L_ROP (record-processing options) field RAB$V_LOA option is specified in the RAB; otherwise, RMS fills the buckets.

When a XABKEY describes the primary key, the XAB$W_DFL field describes the space in the buckets containing actual user data records. When a XABKEY describes an alternate key, the XAB$W_DFL field describes the space in the buckets containing pointers to the user data records.

It is advantageous to use the XAB$W_DFL field if you expect to execute numerous random Put and Update services on the file after it has been initially populated. You can minimize the movement of records (bucket splitting) by specifying less than the maximum bucket fill size when you create the file. To use the free space reserved in the buckets, programs that execute Put or Update services on the file should not specify the RAB$L_ROP field RAB$V_LOA option.

This field corresponds to the FDL attribute KEY DATA_FILL (which is expressed as a percentage).

13.10 XAB$B_DTP Field

The XAB$B_DTP field specifies the key data type and the key sort order, ascending or descending.

In this keyword value field, each key data type option is defined by a symbolic value. If the key sort order is descending, the letter *D* is prefixed to the symbolic value; if the sort order is ascending, the prefix is omitted. For example, a XAB$B_DTP field having the value XAB$C_DBN2 is an unsigned, 2-byte binary number that is sorted in descending order. On the other hand, a XAB$B_DTP field having the value XAB$C_BN2 is an unsigned, 2-byte binary number that is sorted in ascending order.

Only one option can be specified. It is identified by a symbolic constant value; for example, the STG (string) option has the constant value XAB$C_STG.

The options for the XAB$B_DTP field are listed in the following table:

Keyword	Data Type	Sort Order
XAB$C_BN2	Unsigned 2-byte binary	Ascending
XAB$C_DBN2	Unsigned 2-byte binary	Descending
XAB$C_BN4	Unsigned 4-byte binary	Ascending
XAB$C_DBN4	Unsigned 4-byte binary	Descending
XAB$C_BN8	Unsigned 8-byte binary	Ascending
XAB$C_DBN8	Unsigned 8-byte binary	Descending
XAB$C_IN2	Signed 2-byte integer	Ascending
XAB$C_DIN2	Signed 2-byte integer	Descending
XAB$C_IN4	Signed 4-byte integer	Ascending
XAB$C_DIN4	Signed 4-byte integer	Descending
XAB$C_IN8	Signed 8-byte integer	Ascending
XAB$C_DIN8	Signed 8-byte integer	Descending
XAB$C_COL	Collating key	Ascending
XAB$C_DCOL	Collating key	Descending
XAB$C_PAC	Packed decimal string	Ascending
XAB$C_DPAC	Packed decimal string	Descending
XAB$C_STG[1]	Left-justified string of unsigned 8-bit bytes	Ascending
XAB$C_DSTG	Left-justified string of unsigned 8-bit bytes	Descending

[1]This is the default value.

The string data type may consist of from one to eight detached key field segments that collectively make up the key. For more information about segmented keys, see the descriptions of the XAB$W_POS0 through XAB$W_POS7 field and the XAB$B_SIZ0 through XAB$B_SIZ7 field.

Integer, binary, and packed decimal key fields must be a contiguous set of bytes.

The formats of the binary and integer key field data types are presented in the following table.

Key Type	Format
XAB$C_BN2	LSB at A, MSB at A+1
XAB$C_DBN2	LSB at A, MSB at A+1
XAB$C_BN4	LSB at A, MSB at A+3
XAB$C_DBN4	LSB at A, MSB at A+3
XAB$C_BN8	LSB at A, MSB at A+7
XAB$C_DBN8	LSB at A, MSB at A+7
XAB$C_IN2	LSB at A, MSB and sign at A+1
XAB$C_DIN2	LSB at A, MSB and sign at A+1
XAB$C_IN4	LSB at A, MSB and sign at A+3
XAB$C_IN4	LSB at A, MSB and sign at A+3
XAB$C_IN8	LSB at A, MSB and sign at A+7
XAB$C_DIN8	LSB at A, MSB and sign at A+7

The collating key data types are used in conjunction with collating sequences located in the indexed file prolog. Collating sequences are used with multinational characters and are specified for each key. Note that key compression and index compression are not permitted with collating keys.

Note that although a collating key affects the stored order for records, the collating value does not govern record lookups. For example, a collating sequence may assign the same ordering for the keys ‘‘dog’’ and ‘‘DOG’’. However, both keys do not have the same access (lookup) value. Therefore, when doing lookups, a program should specify either the specific key value or a range of values that include the uppercase and lowercase combinations of the key. See the *Guide to OpenVMS File Applications* for more information about accessing indexed records.

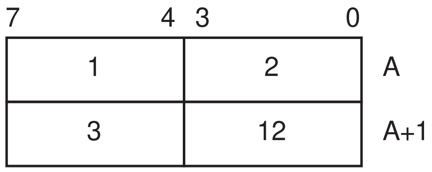
A packed decimal string is a contiguous sequence of bytes specified by two attributes: the address (A) of the first byte of the string and a length (L) that is the number of digits in the packed decimal. The bytes of a packed decimal are divided into two 4-bit fields that must contain decimal digits, except for the first four bits (0 through 3) of the last (highest addressed) byte, which must contain a sign. The representation for the digits and signs is shown in the following table.

Digit or Sign	Decimal Value	Hexadecimal Value
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7

Digit or Sign	Decimal Value	Hexadecimal Value
8	8	8
9	9	9
+	10, 12, 14, or 15	A, C, E, or F
–	11 or 13	B or D

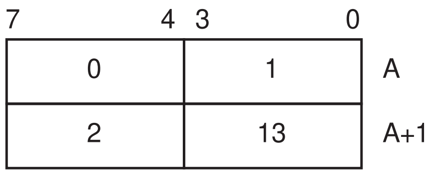
The preferred sign representation is 12 for plus (+) and 13 for minus (–). The length (L) is the number of digits in the packed decimal string (not counting the sign) and must be in the range 0 through 31. When the number of digits is even, an extra 0 digit must appear in the last four bits (4 through 7) of the first byte. Again the length in bytes of the packed decimal is L/2 + 1. The value of a zero-length packed decimal is 0; it contains only the sign byte, which also includes the extra 0 digit.

The address, A, of the packed decimal specifies the byte containing the most significant digit in its high order. Digits of decreasing significance are assigned to increasing byte addresses and from high to low within a byte. Thus, +123 has length 3 and is represented as follows:



ZK–0873–GE

Similarly, –12 has length 2 and is represented as follows:



ZK–0874–GE

This field corresponds to the FDL attribute KEY TYPE.

13.11 XAB$L_DVB Field

After an Open or Display service, the DVB field contains the starting virtual block number of the first data level bucket for the key described by the XAB.

13.12 XAB$B_FLG Field

The key options flag (FLG) field specifies the following conditions:

- Whether duplicate keys are permitted in the file
- Whether a key value can change
- Whether a null value has been defined for a key
- Whether data is compressed

- Whether string key options apply

Note

The string data-type keys include STRING, DSTRING, COLLATED, and DCOLLATED keys.

Primary key values cannot change, but alternate key values may change, depending on application requirements. Primary and alternate keys may be duplicated depending on the key field and the application. An alternate key field is more likely than a primary key field to use duplicates.

This field is a binary options field where each key characteristic has a corresponding bit assignment. Multiple key characteristics can be associated with each key (multiple bits can be set). Each option in the field has its own symbolic offset and mask value. For example, the CHG key characteristic has a symbolic offset of XAB$V_CHG and a mask value of XAB$M_CHG.

When you create an indexed file and are defining a string key, you can specify the XAB$V_IDX_NCMPR and XAB$V_KEY_NCMPR options, which are described in the following list.

Options

XAB$V_CHG
The key value within the record in the file can be changed by a program during an Update service. This option can be specified only for alternate keys.

This option corresponds to the FDL attribute KEY CHANGES.

XAB$V_DAT_NCMPR
Do not compress data. This option can be specified to override compression of data for Prolog 3 files for the primary key; that is, when XAB$_REF is 0.

This option corresponds to the FDL attribute KEY DATA_RECORD_COMPRESSION.

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

XAB$V_DUP
The key value within the record in the file may have the same key value as another record (or other records) within the file.

This option corresponds to the FDL attribute KEY DUPLICATES.

XAB$V_IDX_NCMPR
Do not compress index. This option can be specified to override compression of keys in the index for Prolog 3 files. This option is valid only if a string key is being defined.

This option corresponds to the FDL attribute KEY INDEX_COMPRESSION and it is not supported for DECnet for OpenVMS operations.

XAB$V_KEY_NCMPR
Do not compress key. This option can be specified to override compression of each key for Prolog 3 files. For a primary key (XAB$_REF is 0), the primary keys at the data level are not compressed; for each alternate key (XAB$_REF is greater than 0), the secondary index data records (SIDRs) that point to the data record

location are not compressed. This option is valid only if a string key is being defined.

This option corresponds to the FDL attribute KEY DATA_KEY_COMPRESSION and it is not supported for DECnet for OpenVMS operations.

XAB$V_NUL
The null key option permits you to exclude records from an alternate index by effectively removing the related key. Typically, you would use this in an application where performance is more critical than the indexing capability. This option can be specified only for alternate key indexes but it can be used with all key types.

When you set the XAB$V_NUL bit for a string-type key (string, descending string, collated, descending collated) RMS checks the XAB$B_NUL field to determine the null character you have defined for the related key. When you use the XAB$V_NUL option with the integer, binary, and packed decimal data types, RMS assigns a default null value of 0 and does not check the XAB$B_NUL field (see XAB$B_FLG and XAB$B_NUL).

The defaults and combinations of allowing changeable key values (XAB$V_CHG option) and duplicate key values (XAB$V_DUP option) depend on whether a primary or alternate key is being defined by this XABKEY. The allowed combinations and defaults for duplicate and changeable key values are described in the following table.

Combinations	Primary Key	Alternate Key
XAB$V_CHG and XAB$V_DUP both set	Error	Allowed
XAB$V_CHG set, XAB$V_DUP clear	Error	Allowed
XAB$V_CHG clear, XAB$V_DUP set	Allowed	Allowed
XAB$V_CHG and XAB$V_DUP both clear	Default	Default

By default, duplicate keys are not allowed for the primary key and its value cannot change.

If the XABKEY control block is not initialized by the $XABKEY macro, then the defaults for alternate keys are the same as for primary keys and null key values are not used. However, if the XABKEY control block is initialized by the $XABKEY macro, the following defaults apply to alternate keys:

- Duplicate key values are allowed.
- Key values can change.
- Null key values are not allowed.

These defaults are applied only if the entire XAB$B_FLG field is defaulted.

Note that RMS supports alternate indexes that do not allow duplicate key values but do allow key values to change for Update services. Older versions of RMS–11 (in contrast to RMS) do not allow this particular combination of attributes for alternate indexes. This factor should be considered when you create files with RMS that may also be processed by RMS–11.

This option corresponds to the FDL attribute KEY NULL_KEY.

13.13 XAB$B_IAN Field

The index bucket area number (IAN) field contains a numeric value in the range 0 through 254, representing an area identification number contained in the XAB$B_AID field of a XABALL present in the same chain. The default is 0 (that is, area 0).

When you create an indexed file, you use this argument to specify the area of the file that the index buckets are to reside in only when both of the following are true:

- You are creating a new indexed file.
- You are using allocation XABs to define areas.

When the XABKEY describes the primary key, the index level of the index consists of all levels of the tree-structured primary index down to and including the level containing pointers to the user data records themselves. However, when the key definition describes an alternate key, the index level of the index comprises all levels of the tree-structured alternate index down to, but not including, the level containing buckets with pointer arrays that describe the user data records. For directions about how to place the lowest level of the index in a location separate from the higher levels, see the description of the XAB$B_LAN field.

This field corresponds to the FDL attribute KEY INDEX_AREA.

13.14 XAB$B_IBS Field

After an Open or Display service, the index bucket size (IBS) field contains the size of the index level (level 1 to *n* buckets, in virtual blocks, for the key described by the XAB).

13.15 XAB$W_IFL Field

The index bucket fill size (IFL) field contains a numeric value representing the maximum number of bytes in an index bucket. The maximum possible fill size is the bucket size, in blocks, multiplied by 512. The default value is 0, meaning the maximum available space (that is, no unused space). If the specified size is not 0, but is less than one-half of the bucket size (in bytes), then the fill size used is one-half of the bucket size.

When you create an indexed file, you use this argument to specify the number of bytes you want in each index bucket. If you specify less than the total possible bucket size, you indicate that the index buckets are to contain some amount of free space. At run time, RMS uses the fill size specified at creation time if the LOA option is specified in the RAB$L_ROP (record-processing options) field of the RAB; otherwise, RMS fills the buckets.

When a XABKEY describes the primary key, the XAB$W_IFL field describes the space in the buckets in all levels of the primary index down to and including the level containing pointers to the user data records. When a XABKEY describes an alternate key, the XAB$W_IFL field describes the space in the buckets in all levels of the alternate index down to, but not including, the level containing buckets with pointer arrays that describe the user data records.

It is advantageous to use the XAB$W_IFL field if you expect to perform numerous random Put and Update services on the file after it has been initially populated. You can minimize the movement of index records (bucket splitting) by specifying less than the maximum bucket fill size when a file is created. To use the free space thereby reserved in the buckets, programs that invoke the Put or Update services for writing to the file should not specify the RAB$L_ROP field RAB$V_LOA option.

This field corresponds to the FDL attribute KEY INDEX_FILL (which is expressed as a percentage).

13.16 XAB$L_KNM Field

The key name buffer address (KNM) field contains the symbolic address of a buffer that is available for assigning a user-specified name to the key being defined. The name buffer must be at least 32 bytes in length and you may use any 32-character string you choose to name the key field.

If the default value is taken (0), no name is to be assigned to the key. RMS does not use this string but retains it in the file as part of the key definition information for documentation purposes.

This field corresponds to the FDL attribute KEY NAME.

13.17 XAB$B_LAN Field

The lowest level of index area number (LAN) field contains a numeric value (0 through 254) representing an area identification number contained in the XAB$B_AID field of a XABALL present in the same XAB chain. If the XAB$B_LAN field is not specified (that is, if the value is 0), the value in the XAB$B_IAN field is used as a default; in other words, the lowest level of the index occupies the same area of the file as the remainder of the index.

This field permits you to separate the lowest level (level 1) of the index from all higher levels (levels 2+) of the index in an indexed file; you can use the XAB$B_LAN field to specify an area of the index wherein the lowest level of the index resides, separate from the area (or areas) specified by the XAB$B_IAN field (wherein all other levels of the index reside). See XAB$B_IAN for additional information.

You can specify the XAB$B_LAN field only when both of the following conditions exist:

- You are creating a new indexed file.
- You are using allocation XABs to define areas.

Note that the area specified by the XAB$B_LAN field must have the same bucket size as the area specified by the XAB$B_IAN field.

This field corresponds to the FDL attribute KEY LEVEL1_INDEX_AREA.

13.18 XAB$B_LVL Field

Following an Open or Display service, the level of root bucket (LVL) field contains the level of the root bucket for the key described by the XAB.

13.19 XAB$W_MRL Field

Following an Open or Display service, the minimum record length (MRL) field contains the minimum record length (in bytes) needed to contain the key field for the key described by the XAB.

If the key described by the XAB is the primary key (XAB$_REF is 0), then a record must be equal to or greater than the minimum record length returned in XAB$W_MRL to be inserted or updated in the file.

If the key described by the XAB is an alternate key (XAB$_REF is greater than 0), then a record must be equal to or greater than the minimum record length returned in the XAB$W_MRL field to be recorded in the associated index for that alternate key.

13.20 XAB$B_NSG Field

Following an Open or Display service, the number of key segments (NSG) field contains the number of key segments that make up the key field for the key described by the XAB (see the XAB$W_POS0 through XAB$W_POS7 field).

13.21 XAB$B_NUL Field

Normally, RMS updates all indexes to reflect the values in the corresponding key fields of the records written to an indexed file. The XAB$B_NUL field permits you to instruct RMS not to make an entry in an alternate index if a record being entered in an indexed file contains a specified null alternate key value. To specify the XAB$B_NUL field, three conditions must be satisfied:

- The XABKEY must define an alternate key.
- The XAB$B_FLG field XAB$V_NUL option must be set when you create the file (see XAB$B_FLG).
- The key data type must be string.

You can use any ASCII character in the null (NUL) field when you define a string-type alternate key (string, descending string, collated, descending collated). If you do not specify a null value, RMS assigns the key a default null value of 0.

This field corresponds to the FDL attribute KEY NULL_VALUE.

13.22 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

13.23 XAB$W_POS0 Through XAB$W_POS7 Field

There are two types of keys: simple keys and segmented keys.

A simple key is made up of one or more contiguous bytes and it may be used with any data type, including the string data type. For simple keys, the first byte of the key position field contains a byte offset relative to the beginning of the record buffer that defines the starting position of the key. The remaining bytes contain zeros.

Segmented keys include two through eight strings of key data (segments) and can only be used with string data type key fields. The key segments need not be contiguous nor must they be in a particular order. Key segments may overlap except for primary keys used with Prolog 3 files. If your application requires overlapping key segments in a Prolog 3 file, consider using an alternate segmented key. If you must have a primary key with overlapping segments, RMS requires you to use either a Prolog 2 or Prolog 1 structure (which it automatically assigns if the XAB$B_PROLOG field is not specified).

For segmented keys, the first word of the key position field specifies the starting position of the first segment and each succeeding byte specifies the starting position of one of the remaining segments. When processing records that contain segmented keys, RMS regards a segmented key field as a single, logically contiguous string beginning with the first segment and ending with the last.

You should note that the XAB$W_POS0 through XAB$W_POS7 and the XAB$B_SIZ0 through XAB$B_SIZ7 (key size) fields must define the same number of key position values and key size values.

This field corresponds to the FDL attributes KEY POSITION and SEGn_LENGTH.

13.24 XAB$B_PROLOG Field

The prolog (PROLOG) field defines the version or structure level of the file index. It contains a numeric value from 0 through 3.

The XAB$B_PROLOG field is input to the Create service, and it is returned by the Display and Open services.

This field must only be used to define a primary key.

Prolog 3 is the default prolog level, unless the primary key contains overlapping segments. RMS examines the key characteristics and determines the correct prolog structure to apply to the file. If the XAB$B_PROLOG field is not specified (that is, if the value is 0), the process default prolog level is examined, then the system default prolog level is used. These default values are set by the DCL command SET RMS_DEFAULT/PROLOG.

You should not specify a prolog level 1 because RMS decides whether a Prolog 1 or Prolog 2 file should be created, depending on the key type defined for the file. If you want to select a prolog level other than Prolog 3, you should select either 0 or 2.

For more detailed information regarding the options for selecting a specific prolog level, see the description of the Create service in Part III.

This field corresponds to the FDL attribute KEY PROLOG and it is not supported for DECnet for OpenVMS operations; the default prolog in effect at the remote node is used.

13.25 XAB$B_REF Field

The key of reference (REF) field defines a key as either the primary key or some alternate key.

Note

For BLISS-32, this field is designated XAB$B_KREF.

This field contains a numeric value in the range 0 through 254. A value of 0 indicates that this is the primary key; a value of 1 indicates the first alternate key; a value of 2 indicates the second alternate key, and so on. The order of the XABKEYs is irrelevant.

Note that RMS can process an indexed file with 255 defined keys; each defined key field, however, has an associated cost in processing and I/O time. The time required to build and maintain the index for the key field and the disk storage required to contain the index for each key field should be considered when you decide whether the field should be an alternate key field. A file with six to eight defined keys (the primary key and five to seven alternate keys) should be considered as a maximum; a file with two or three defined keys is typical.

This field corresponds to the FDL attribute KEY *n*, where *n* is the number of the key being defined.

13.26 XAB$L_RVB Field

After an Open or Display service, the root index bucket virtual block number (RVB) field contains the virtual block number for the root bucket of the index for the key described by the XAB.

13.27 XAB$B_SIZ0 Through XAB$B_SIZ7 Field

The key size (SIZ) field defines the length of the key field within each record. This field contains a numeric value representing the length, in bytes, of the key within the record. Up to eight values can be assigned; maximum values depend on the type of key.

The XAB$B_SIZ0 through XAB$B_SIZ7 field defines the length (in bytes) of the key whose starting position is defined in the key position field of the XAB. Two types of keys can be defined: simple and segmented (see the XAB$W_POS0 through XAB$W_POS7 field).

For a simple key, the XAB$B_SIZ0 through XAB$B_SIZ7 field contains only one key size value (in XAB$B_SIZ0).

For a segmented key, the XAB$B_SIZ0 through XAB$B_SIZ7 field contains a key size value for each segment of the key. You should note that the XAB$B_SIZ0 through XAB$B_SIZ7 field and the XAB$W_POS0 through XAB$W_POS7 field must contain the same number of key size values and key position values. RMS associates the first key position value with the first key size value to define the location and length of the first segment of a segmented key, and so forth.

When the data type of the key is string, the total size (sum of all sizes) of the key must be less than 256 bytes.

When the data type of the key is 2-byte integer or 2-byte binary, XAB$B_SIZ0 must equal 2 and XAB$B_SIZ1 through XAB$B_SIZ7 must contain 0. If the size is 0, it defaults to 2.

When the data type of the key is 4-byte integer or 4-byte binary, XAB$B_SIZ0 must equal 4 and XAB$B_SIZ1 through XAB$B_SIZ7 must contain 0. If the size is 0, it defaults to 4.

When the data type of the key is 8-byte integer or 8-byte binary, XAB$B_SIZ0 must equal 8 and XAB$B_SIZ1 through XAB$B_SIZ7 must contain 0. If the size is 0, it defaults to 8.

When the data type of the key is packed decimal, the size specified by XAB$B_SIZ0 must be from 1 through 16, and XAB$B_SIZ1 through XAB$B_SIZ7 must contain 0.

This field corresponds to the FDL attribute KEY LENGTH or KEY SEG *n*_LENGTH, where *n* is the number of the segment being defined.

13.28 XAB$B_TKS Field

After an Open or Display service, the total key size (TKS) field contains the total key size (the sum [in bytes] of XAB$B_SIZ0 through XAB$B_SIZ7) for the key described by the XAB.

14
Protection XAB (XABPRO)

The protection XAB (XABPRO) specifies the ownership, accessibility, and protection for a file. Although an application program typically uses a XABPRO as input to establish file protection when it creates a file, it can also use the XABPRO to change file protection when it closes a file. The program that opened the file can change file protection when it closes the file *only* if it accessed the file to make modifications (FAC = PUT, UPDATE, DELETE, or TRUNCATE) and has control access. Control access grants to the file accessor all of the file access privileges of the file owner.

For more information about control access, see the *Security Guide*.

RMS also uses the XABPRO to return file protection information to the application program by way of the Display service and the Open service.

14.1 Summary of Fields

The symbolic offset, the FDL equivalent, and a brief description of each XABPRO field are presented in Table 14–1.

Table 14–1 XABPRO Fields

Field Offset	FDL Equivalent	Description
XAB$L_ACLBUF[3]	None	Address of buffer that contains ACL
XAB$L_ACLCTX[3]	None	ACL positioning context
XAB$W_ACLLEN[3]	None	Receives the length of an ACL during an Open or Display service
XAB$W_ACLSIZ[3]	None	Length of buffer containing binary ACEs
XAB$L_ACLSTS[3]	None	System error status for ACL processing
XAB$B_BLN[1]	None	Block length
XAB$B_COD[1]	None	Type code
XAB$W_GRP[2]	FILE OWNER	Group number of file owner
XAB$W_MBM[2]	FILE OWNER	Member number of file owner
XAB$B_MTACC	FILE MT_PROTECTION	Magnetic tape accessibility
XAB$L_NXT	None	Next XAB address

[1]This field is statically initialized by the $XABPRO macro to identify this control block as a XABPRO.

[2]This field cannot be initialized by the $XABPRO macro.

[3]ACL operations apply only to Files–11 ODS-2 files.

(continued on next page)

Table 14–1 (Cont.) XABPRO Fields

Field Offset	FDL Equivalent	Description
XAB$W_PRO	FILE PROTECTION	File protection; contains four separate fields denoting protection for system, owner, group, and world
XAB$B_PROT_OPT	None	File protection options
XAB$L_UIC	FILE OWNER	User identification code; contains both the group and member fields

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

The format and arguments of the $XABPRO macro and the $XABPRO_STORE macro are defined in Appendix A.

14.2 XAB$L_ACLBUF Field

The ACL buffer field (ACLBUF) stores the address of a buffer area that contains an access control list (ACL) for this file. The ACL buffer contains one or more access control entries (ACE) in binary format. The system processes the ACL until it encounters an ACE with a length byte value of 0 or until it reaches the end of the buffer as indicated by XAB$W_ACLSIZ. The ACL buffer is used as input to a Create service and as output from an Open or Display service. The address in XAB$L_ACLBUF is used only as input to these services.

During a Create operation, if the XAB$L_ACLBUF field has a value other than 0, RMS attempts to create the file using the value in the ACL buffer. When the XAB$L_ACLBUF field has a value of 0 during a Create operation, the file has an ACL only if an ACL is specified by the systemwide defaults. Once a file has been created, the ACL cannot be changed using RMS.

During an Open or a Display operation, if the XAB$L_ACLBUF field has a value other than 0, RMS passes this address to the file system. The file system then fills the user's buffer with the file's ACL (in binary format). If the entire ACL does not fit into the user's buffer, the file system puts only as many ACEs as possible into the buffer. (See the XAB$L_ACLCTX field for more information.)

You can convert an ASCII ACL to binary format by using the $PARSE_ACL system service, and you can convert an ACL from binary format to ASCII using the $FORMAT_ACL system service. For information about using the $PARSE_ACL and $FORMAT_ACL services, see the *OpenVMS System Services Reference Manual*.

The use of this field for DECnet for OpenVMS remote file access is not supported.

14.3 XAB$L_ACLCTX Field

The XAB$L_ACLCTX field is used as a placeholder by RMS, and it is used as an input and output field by RMS during Open and Display operations when the XAB$L_ACLBUF field has a value other than 0. In order to read an ACL beginning with the first ACE, the XAB$L_ACLCTX field must have a value of 0. When the initial Open or Display operation is complete, RMS fills the XAB$L_ACLCTX field with a value that serves as a context field, allowing subsequent

Open or Display operations that read the remainder of the ACL (if the entire list of ACEs did not fit into the user's buffer).

For example, suppose you perform an Open operation, find that the value of XAB$W_ACLLEN is greater than the ACL buffer, and then perform Display operations until all of the ACEs in the ACL have been returned. You can then reread the entire ACL on subsequent Opens or Displays only if you set the value of the XAB$L_ACLCTX field to 0.

The use of this field for DECnet for OpenVMS remote file access is not supported.

14.4 XAB$W_ACLLEN Field

The ACL length (ACLLEN) field receives the length (in bytes) of the access control list for the file during an Open or a Display operation. If the file has no ACL, the XAB$W_ACLLEN field has a value of 0.

If the file has an ACL that fits in the user's buffer, the value of the XAB$W_ACLLEN field is equal to the number of bytes in the ACL. Even if the file's ACL does not fit into the user's buffer, the value of the XAB$W_ACLLEN field is still equal to the number of bytes in the ACL (not just the length of that portion that fits into the buffer).

To determine the number of ACL entries that are in the user's buffer, you must process binary ACEs until you find an ACE with a value of 0 or until you come to the end of the buffer.

The use of this field for DECnet for OpenVMS remote file access is not supported.

14.5 XAB$W_ACLSIZ Field

The ACL buffer size (ACLSIZ) field specifies the length of the user buffer pointed to by the XAB$L_ACLBUF field.

RMS passes all information, including the ACL buffer, to and from the file system using buffered I/O operations. RMS limits buffered I/O transfers to 512 bytes, excluding the ACL buffer. Therefore, the size of the ACL buffer plus 512 bytes cannot exceed either the BYTLM quota for the process or the MAXBUF value for the system.

The use of this field for DECnet for OpenVMS remote file access is not supported.

14.6 XAB$L_ACLSTS Field

The ACL error status (ACLSTS) field contains a system error status relating to the processing of ACLs. A value is returned to this field upon a successful return from a Create, Open, or Display service.

Whenever you use the XABL_ACLBUF, XABL_ACLCTX, XAB$W_ACLLEN, or XAB$W_ACLSIZ fields, be sure to use the following error-handling guidelines:

- If the FAB$L_STS field (R0) contains an error status, handle the error in the usual manner.
- If the FAB$L_STS field (R0) contains a success status, then you must check the value in XAB$L_ACLSTS. If XAB$L_ACLSTS contains a success status, then the entire operation completed successfully and no further action is required; if XAB$L_ACLSTS contains an error status, handle the error appropriately. Note that a value is placed in the XAB$L_ACLSTS field *only* when a success status is returned in FAB$L_STS (R0).

This extra level of error checking is necessary because the success or failure of reading and writing ACLs is independent of the success or failure of the whole operation. Thus, in the absence of this additional error checking, it is possible to create a file successfully even though an ACL error occurred.

This field is relevant only when an ACL is used with a Create service or when an ACL is returned from an Open or Display service. The use of this field for DECnet for OpenVMS remote file access is not supported.

14.7 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABPRO, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_PROLEN (this is done by the $XABPRO macro).

14.8 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABPRO. Once set, this field must not be altered unless the control block is no longer needed. This field must be initialized to the symbolic value XAB$C_PRO (this is done by the $XABPRO macro).

14.9 XAB$W_GRP Field

The file owner group number (GRP) field contains the half of the XAB$L_UIC field that defines the group number. Refer to the XAB$L_UIC field description for additional information. The contents of the XAB$L_UIC field, rather than the $XABPRO macro, establish the initial value of the XAB$W_GRP field.

This field corresponds to the FDL attribute FILE OWNER.

14.10 XAB$W_MBM Field

The file owner member number (MBM) field contains the half of the XAB$L_UIC field that defines the member number. Refer to the XAB$L_UIC field description for additional information. The contents of the XAB$L_UIC field, rather than the $XABPRO macro, establish the initial value of the XAB$W_MBM field.

This field corresponds to the FDL attribute FILE OWNER.

14.11 XAB$B_MTACC Field

The magnetic tape accessibility (MTACC) field enables you to access HDR1 labels for ANSI-labeled magnetic tapes, in compliance with ANSI standards. The value specified in the XAB$B_MTACC field is input to the Create service and output from the Open and Display services for magnetic tape only.

The character to be inserted in the accessibility field of the HDR1 label must be one of the following:

- An uppercase letter from A through Z
- A digit from 0 through 9

- One of the following special characters: (!), (%), (&), (’), (( ), ( )), (*), (+), (,), (-), (.), (/), (:), (;), (<), (=), (>), (?)

Note that if this field is not specified or if the specification is invalid, a space character is inserted into the HDR1 accessibility field.

This field corresponds to the FDL attribute FILE MT_PROTECTION, and it is not supported for DECnet for OpenVMS operations.

14.12 XAB$L_NXT Field

The next XAB address (NXT) field specifies the symbolic address of the next XAB in the XAB chain. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

14.13 XAB$W_PRO Field

The file protection (PRO) field specifies the file access privileges that RMS grants to the four classes of users: System, Owner, Group, and World.

This field consists of four 4-bit subfields, each of which specifies file access privileges for one of the four user classes.

You can specify the user class using the $XABPRO macro or through the use of symbolic offsets, if you do not have access to the VAX macro libraries. This is the syntax for representing the four user classes as arguments to a $XABPRO macro:

<SYSTEM,OWNER,GROUP,WORLD>

Alternatively, you can specify the user class using the appropriate symbolic offset from the following list:

- System—XAB$V_SYS
- Owner—XAB$V_OWN
- Group—XAB$V_GRP
- World—XAB$V_WLD

You can also specify access privileges for each user class using the $XABPRO macro or through the use of symbolic mask values. Using a $XABPRO macro, you insert the appropriate initial letters for each user class in the macro argument:

- R—read access
- W—write (modify) access
- E—execute access
- D—delete access

For example, to grant each user class read and write privileges, you would use this line of code:

```
PRO = <RW,RW,RW,RW>
```

Alternatively, you can specify user class privileges using the appropriate symbolic mask value from the following list:

- No read access—XAB$M_NOREAD
- No write access—XAB$M_NOWRITE

- No execute access—XAB$M_NOEXE
- No delete access—XAB$M_NODEL

If you do not specify access privileges for a file, by default RMS grants all access privileges (read, write, execute, delete) to all user classes by zeroing all bits in the XAB$W_PRO field.

Note

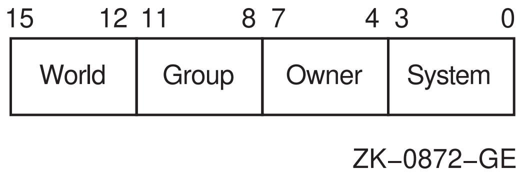
A logical 0 enables access; a logical 1 denies access.

You cannot deny all access to all user classes through the RMS interface. If you set all bits in the XAB$W_PRO field to logical 1, RMS assigns the file process default protection.

Users are granted the maximum number of types of access rights for each of the classes to which they belong.

Figure 14–1 illustrates the organization of the subfields in the XAB$W_PRO field.

Figure 14–1 File Protection Field



If you do not explicitly specify file protection when you invoke the Create service, RMS attempts to determine file protection by default in the following order:

1. Using the protection assigned to an existing file of the same name
2. Using the default file protection of the directory
3. Using the process-default protection

The following table provides detailed descriptions of the four user classes.

User Class	Description
System	Specifies access rights for users executing under a system UIC, that is, users whose group number is less than the value for a system UIC, which is defined by the system manager (usually 10 or less).
Owner	Specifies access rights for the owner of the file. A user is considered the owner of the file only if both the group and member number fields of the accessing process match the group and member number fields of the file owner's UIC stored with the file.
Group	Specifies the access rights for users whose group number matches the group number field of the file owner.
World	Specifies access rights for all users. World access is used for granting access to users who are not in the system, owner, or group classifications.

This field corresponds to the FDL attribute FILE PROTECTION.

14.14 XAB$B_PROT_OPT Field

The ACL file protection (PROT_OPT) field provides a single option, the XAB$V_PROPAGATE option, which is used as input during an Enter or a Rename operation. (During a Rename operation, the protection XAB is assumed to be attached to FAB2.)

The XAB$B_PROT_OPT field is a binary options field where each file protection option has a corresponding bit assignment. Options are identified using mnemonics, and each option has its own symbolic offset and mask value. For example, the PROPAGATE option has a symbolic offset of XAB$V_PROPAGATE and a mask value of XAB$M_PROPAGATE.

If the XAB$V_PROPAGATE bit is set in this field during either an Enter or Rename operation, the file receives new security attributes when the new directory entry is made. These security attributes follow the same rules as apply during a Create operation. For example, if a lower version of a new file exists, the new file inherits the security attributes of the next lower version of the file. If the XAB$V_PROPAGATE bit is not set, the security attributes of the new file do not change.

This field is not supported for DECnet for OpenVMS operations.

14.15 XAB$L_UIC Field

The user identification code (UIC) field combines the two XABPRO fields that define the UIC of the owner of a file: the XAB$W_GRP (group number) and XAB$W_MBM (member number) fields. Both numbers are octal numbers. The valid range for a group number is 0 to 37777; the valid range for a member number is 0 to 177777. Note that the maximum value in each case (37777 and 177777) is reserved for Digital use only. This field corresponds to the FDL attribute FILE OWNER.

The symbolic offsets for the group number field and the member number field respectively are XAB$W_GRP and XAB$W_MBM.

The total user identification field, including both the group and member number fields, has a symbolic offset of XAB$L_UIC.

Note that if no file protection XAB is provided or if the user identification field is null for a Create service, RMS determines the owner's UIC using the following logical order:

1. The owner UIC of an existing version of the file if the creating process has ownership rights to the previous version
2. The owner UIC of the parent directory, if the creating process has ownership privileges to the parent directory
3. The UIC of the creating process

If you wish to create an output file with a UIC different from your own, you must have system privilege (SYSPRV).

15
Revision Date and Time XAB (XABRDT)

The revision date and time XAB (XABRDT) complements the date and time XAB (XABDAT) by providing revision time and date input to the Close service when RMS closes a file that has been modified. Like the XABDAT, the XABRDT can be used as input to the Create Service or can be used to store revision data returned by the Open service or the Display service. The distinction is that XABDAT cannot be used to modify the revision data. Only the XABRDT can be used to update revision data when a file is closed after being modified. Typically, a process would use the two data structures when it wants to sense and set revision data within a single file operation.

To change the revision data, the process must have Control access to the file and must have opened the file for modification using the Put, Update, Truncate, or Delete service. Normally when RMS closes a file and these conditions prevail, it uses the current date and time as the revision date and time and increments the revision number. Thus, the previous revision values for the file are superseded. (See the *Security Guide* for more information about CONTROL access.)

Using the XABRDT as an input to a Close service, you can specify a set of revision values other than the current time and date and the next revision number.

15.1 Summary of Fields

The two XABRDT fields that specify revision values are the XABQ_RDT (revision date and time) field and the XAB$W_RVN (revision number) field.

- The 64-bit XAB$Q_RDT binary field indicates the date and time when the file was last opened for modifications.
- The XAB$W_RVN field indicates how many times the file is opened for modifications.

The following table indicates how RMS uses the XABRDT fields for various file-processing various services.

Service	Input/Output
Close	Input
Create	Input
Display	Output
Erase	Not used
Extend	Not used
Open	Output

The Open service overwrites the XAB$Q_RDT and XAB$W_RVN fields with the file's existing revision values. If you do not change these values, the existing values are subsequently input to the Close service and the file's revision data is not updated.

To change the revision data, you must set the fields between the time you open the file and the time you close the file. If you specify a revision date and time of 0 or if you do not include a XABRDT as input to the Close service, RMS uses the current date and time for the revision date and time and increments the revision number.

To sense the contents of the XAB$Q_RDT and XAB$W_RVN fields before you specify new values with the XABRDT, examine the XAB$Q_RDT field and the XAB$W_RVN field in the XABDAT block.

The symbolic offset, size, FDL equivalent, and a brief description of each XABRDT field are presented in Table 15–1.

Table 15–1 XABRDT Fields

Field Offset	Size (Bytes)	FDL Equivalent	Description
XAB$B_BLN[1]	1	None	Block length
XAB$B_COD[1]	1	None	Type code
XAB$L_NXT	4	None	Next XAB address
XAB$Q_RDT[2]	8	DATE REVISION	Revision date and time
XAB$W_RVN[2]	2	FILE REVISION	Revision number

[1]This field is statically initialized by the $XABRDT macro to identify this control block as a XABRDT.

[2]This field cannot be initialized by the $XABRDT macro; it must be specified before you invoke the Close service to be used as input to the Close service.

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. For information about the support of RMS options for remote file access to other systems, see the *DECnet for OpenVMS Networking Manual*.

The format and arguments of the $XABRDT and $XABRDT_STORE services are defined in Appendix A.

15.2 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABRDT, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field is initialized to the symbolic value XAB$C_ RDTLEN by the $XABRDT macro.

15.3 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABRDT. Once set, this field must not be altered unless the control block is no longer needed. This field is initialized to the symbolic value XAB$C_RDT by the $XABRDT macro.

15.4 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB to be used. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

15.5 XAB$Q_RDT Field

The revision date and time (RDT) field contains a 64-bit binary value expressing the date and time when the file was last opened for modifications. Note that this field is limited to a granularity of 1 second for remote files.

This field corresponds to the FDL attribute DATE REVISION.

15.6 XAB$W_RVN Field

The revision number (RVN) field contains a numeric value that indicates the number of times this file was opened for modifications.

This field corresponds to the FDL attribute FILE REVISION.

16

Recovery Unit XAB (XABRU)

The recovery unit XAB (XABRU) control block supports the use of recovery units to assure data file integrity. See the *RMS Journaling for OpenVMS Manual* for details.

17
Summary XAB (XABSUM)

The summary XAB (XABSUM) can be associated with a FAB at the time a Create, Open, or Display service is invoked. The presence of this XAB during these calls allows RMS to return to your program the total number of keys and allocation areas defined and the version number when the file was created. Note that a XABSUM is used only with indexed files.

17.1 Summary of Fields

The symbolic offset, the size, and a brief description of each XABSUM field are presented in Table 17–1.

Table 17–1 XABSUM Fields

Field Offset	Size	Description
XAB$B_BLN[1]	Byte	Block length
XAB$B_COD[1]	Byte	Type code
XAB$B_NOA[2]	Byte	Number of allocation areas defined for the file
XAB$B_NOK[2]	Byte	Numbers of keys defined for the file
XAB$L_NXT	Longword	Next XAB address
XAB$W_PVN[2]	Word	Prolog version number

[1]This field is statically initialized by the $XABSUM macro to identify this control block as a XABSUM.
[2]This field cannot be initialized by the $XABSUM macro.

Unless otherwise indicated, each field is supported for DECnet for OpenVMS operations on files at remote OpenVMS systems. See the *DECnet for OpenVMS Networking Manual* for information about the support of RMS options for remote file access to other systems.

The format and arguments of the $XABSUM and $XABSUM_STORE macros are defined in Appendix A.

17.2 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABSUM, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field is initialized to the symbolic value XAB$C_SUMLEN by the $XABSUM macro.

17.3 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABSUM. Once set, this field must not be altered unless the control block is no longer needed. This field is initialized to the symbolic value XAB$C_SUM by the $XABSUM macro.

17.4 XAB$B_NOA Field

The number of allocation areas (NOA) field indicates the number of allocation areas defined when the file was created. Refer to Chapter 8 for information about multiple allocation areas.

17.5 XAB$B_NOK Field

The number of keys (NOK) field indicates the number of keys defined when the file was created. Refer to Chapter 13 for more information.

17.6 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

17.7 XAB$W_PVN Field

The prolog version number (PVN) contains a numeric value that indicates the prolog number defined when the file was created. For more information about prolog numbers, refer to Chapter 13.

18
Terminal XAB (XABTRM)

The terminal XAB (XABTRM) allows extended terminal read operations to occur when a Get service is used for a terminal device. Unlike most other XABs, the XABTRM is associated with a RAB (record stream). The XABTRM provides information that the terminal driver uses to process a user-defined item list that defines the terminal read operation.

18.1 Summary of Fields

The symbolic offset, the size, and a brief description of each XABTRM field are presented in Table 18–1.

Table 18–1 XABTRM Fields

Field Offset	Size (Bytes)	Description
XAB$B_BLN[1]	1	Block length
XAB$B_COD[1]	1	Type code
XAB$L_ITMLST	4	Item list address
XAB$W_ITMLST_LEN	2	Item list length
XAB$L_NXT	4	Next XAB address

[1]This field is statically initialized by the $XABTRM macro to identify this control block as a XABTRM.

To perform the extended terminal read operation, the following information is required:

- In the RAB, the RAB$L_ROP field RAB$V_ETO option must be specified (set).
- In the RAB, the RAB$L_XAB field must contain the address of the XABTRM.
- In the XABTRM, the XAB$L_ITMLST and XAB$W_ITMLST_LEN fields must contain the starting address and length of a valid terminal driver read function item list.
- The item list must be supplied according to the conventions described for creating an item list for the terminal driver in the *OpenVMS I/O User's Reference Manual*.

An item list consists of one or more item list entries, where each item defines an attribute of the terminal read operation. Instead of defining terminal read arguments in the RAB, all such arguments (including certain arguments only available with the item list method) are defined in the item list. The following list shows the RAB$L_ROP options related to a terminal read operation and the equivalent item codes:

Option	Item Code
RAB$V_CVT	TRM$_MODIFIERS, bit TRM$M_TM_CVTLOW
RAB$V_PMT	TRM$_PROMPT
RAB$V_PTA	TRM$_MODIFIERS, bit TRM$M_TM_PURGE
RAB$V_RNE	TRM$_MODIFIERS, bit TRM$M_TM_NOECHO
RAB$V_RNF	TRM$_MODIFIERS, bit TRM$M_TM_NOFILTR
RAB$V_TMO	TRM$_TIMEOUT

Each item code required for the terminal read operation is placed in an item list along with other required information. Each item code is made up of three longwords. Note that RMS does not validate the item list. If the item list is invalid, RMS returns RMS$_QIO status in the RAB$L_STS field and the specific terminal driver QIO status in the RAB$L_STV field (see the *OpenVMS I/O User's Reference Manual*).

The XABTRM is *not* supported for DECnet for OpenVMS operations between two OpenVMS systems. There are no equivalent FDL attributes for the XABTRM fields.

The format and arguments of the $XABTRM macro and the $XABTRM_STORE macro are defined in Appendix A.

18.2 XAB$B_BLN Field

The block length (BLN) field is a static field that defines the length of the XABTRM, in bytes. Once set, this field must not be altered unless the control block is no longer needed. This field is initialized to the symbolic value XAB$C_TRMLEN by the $XABTRM macro.

18.3 XAB$B_COD Field

The type code (COD) field is a static field that identifies this control block as a XABTRM. Once set, this field must not be altered unless the control block is no longer needed. This field is initialized to the symbolic value XAB$C_TRM by the $XABTRM macro.

18.4 XAB$L_ITMLST Field

The item list address (ITMLST) field contains the symbolic address of the item list that defines the extended terminal read operation.

18.5 XAB$W_ITMLST_LEN Field

The item list length (ITMLST_LEN) field contains a numeric value that indicates the length of the item list, in bytes.

18.6 XAB$L_NXT Field

The next XAB address (NXT) field contains the symbolic address of the next XAB to be used. A value of 0 (the default) indicates that the current XAB is the last (or only) XAB in the chain.

Part III

OpenVMS RMS Services

Part III lists the format of each record management service and describes each service in detail. Each service is documented in a structured format. See *OpenVMS Programming Interfaces: Calling a System Routine* for a discussion of the format and how it is used.

Note that the calling format for each service requires a placeholder (a comma) if you omit the first optional argument (**err**) but include the second optional argument (**suc**).

$CLOSE

The Close service terminates file processing and closes the file. This service performs an implicit Disconnect service for all record streams associated with the file.

Format

SYS$CLOSE fab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage:	fab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

FAB control block whose contents are to be used as indirect arguments for the Close service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

You can invoke the Close service only when no operation is currently under way (by your process) for the file being processed; that is, when no RMS requests for the file are outstanding.

When the Close service is invoked properly, RMS disconnects all RABs for you, performs the various cleanup procedures (including file option processing and XAB processing), and closes the file. The only types of XABs that the Close service processes are the file protection XAB (XABPRO) and revision date and time XAB (XABRDT). It processes these XABs only if the file was opened or created for write access.

On a $CLOSE, the EOF value written to the file header is ‘‘seen’’ only by subsequent accessors. Any process that has the file open at the time of the $CLOSE does not ‘‘see’’ the new EOF value without some explicit action.

If a process tries to implement the Truncate service when closing a sequential file, it must have sole write access to the file. If other processes have write access to the file, it remains accessible until all processes have completed. If other processes have the file open for read access, RMS defers the truncation until the final process having read access closes the file.

Table RMS–1 lists the control block fields read as input by the Close service. Note that if the FABV_DLT, FABV_SCF, or the FAB$V_SPL bits are set by the associated Open or Create service, RMS does not act on them for the Close service. For example, if you open the file and specify that it be deleted on close by setting the FAB$V_DLT bit, RMS deletes it when it is closed regardless of the bit’s state when the Close service is invoked.

For additional information on the fields accessed by this service, see Part II.

Table RMS–1 Close Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
FAB$W_IFI[1]		Internal file identifier.
FAB$L_FOP		File-processing options.
	FAB$V_ASY	Asynchronous; indicates that the specified task is to be done asynchronously.
	FAB$V_DLT	Deletes file on close.
	FAB$V_RWC	Rewinds a magnetic tape volume.
	FAB$V_SCF[2]	Submits a file as a batch job (sequential files only).
	FAB$V_SPL[2]	Submits a file to the print queue (sequential files only).
	FAB$V_TEF	Truncates data at the end of the file (sequential files only).

[1]This field is required input to the FAB.

[2]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–1 (Cont.) Close Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
FAB$L_XAB		Next XAB address.
	XABPRO	Modifies file protection and ownership.
	XABRDT	Modifies revision date and number.

Table RMS–2 lists the control block fields written as output by the Close service.

Table RMS–2 Close Service FAB and XAB Output Fields

Field Name	Option or XAB Type	Description
FAB$W_IFI		Internal file identifier (cleared only when the file is closed).
FAB$L_STS		Completion status (also returned in register 0).
FAB$L_STV		Status value.
FAB$L_XAB		Next XAB address.
	XABRDT	New revision date and number returned.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BUG_DAP	RMS$_BUSY
RMS$_CCF	RMS$_CDA	RMS$_COD
RMS$_CRC	RMS$_DAC	RMS$_DME
RMS$_DNR	RMS$_EXENQLM	RMS$_FAB
RMS$_IFI	RMS$_IMX	RMS$_MKD
RMS$_NET	RMS$_NETFAIL	RMS$_NORMAL
RMS$_PRV	RMS$_SPL	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS	RMS$_WBE	RMS$_WER
RMS$_WLK	RMS$_WPL	RMS$_XAB

Note that even though a failure is indicated by the completion status code value, the file is truly closed only if RMS clears the internal file identifier value (FAB$W_IFI).

$CONNECT

The Connect service establishes a record stream by associating and connecting a RAB with a FAB. You can invoke the Connect service only for files that are already open.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Connect service on OpenVMS Alpha systems. ♦

Format

SYS$CONNECT rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Connect service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the Connect service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the Connect service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

Any number of RABs can be connected to a FAB if the multistream (FAB$V_MSE) option is selected when the file is opened or created. Each RAB represents an independent record stream.

When you issue a Connect service, RMS allocates an internal counterpart for the RAB. This counterpart consists of the internal controls needed to support the stream, such as record pointers and request status information. All required I/O buffers are also allocated at this time.

The Connect service also initializes the next record pointer to the first record. In indexed files, the key of reference establishes the index of the next record pointer.

If you set the end-of-file (RAB$V_EOF) option in the RAB$L_ROP field when issuing a Connect service, RMS examines the organization of the file being processed to determine the end-of-file positioning strategy.

For sequential or relative files, RMS goes to the next record beyond the last currently existing record in the file. (The next record is inserted at the logical end of the file, and the service returns RMS$_EOF status in response to a request for sequential access.)

For indexed files, RMS verifies that the first record is inserted in the proper sort order. If the record cannot be inserted in the proper sort order because of user action, RMS returns a sequence error (RMS$_SEQ).

Get services that specify the sequential record access mode (RAB$B_RAC is RAB$C_SEQ) return an RMS$_EOF status. Get services that specify the random access mode (RAB$B_RAC is RAB$C_KEY) ignore (turn off) the end-of-file positioning. Positioning to end-of-file is supported for all indexed files, regardless of how many indexes the file contains. However, the EOF positioning is supported only when you access a file by the primary key. If the specified key of reference is a secondary key, an RMS$_ROP message is returned.

In most cases, setting the RAB$V_EOF bit guarantees that the next record is inserted at the logical end of the file. However, if a relative file or an indexed file is shared by two or more active processes, the following scenario may develop.

Assume that process A has invoked the Connect service after setting the RAB$V_EOF bit and is positioned to the end of the file. Before process A can do a $PUT, process B inserts a record into the file and changes the current record position. When process A attempts to do a $PUT into the position that was formerly the end of the file, the record may be inserted improperly. It may be inserted either before or after the record inserted by process B, depending on the respective key values. Or, the $PUT operation may even fail if the keys have the same value and duplicates are not allowed.

Table RMS–3 lists the control block fields read as input by the Connect service. For additional information about the fields accessed by this service, see Part II.

Table RMS–3 Connect Service RAB Input Fields

Field Name	Option or XAB Type	Description
FAB$W_GBC		Global buffer count.
RAB$L_FAB[1]		File access block address (required to access the internal file identifier field, FAB$W_IFI).
RAB$W_ISI[1]		Internal stream identifier (must be 0).
RAB$B_KRF		Key of reference (applies only to indexed files).
RAB$B_MBC[2]		Multiblock count (applies only to sequential files residing on disk devices).
RAB$B_MBF[2]		Multibuffer count.[4]
RAB$L_ROP		Record-processing options:
	RAB$V_ASY	Asynchronous: performs Connect service asynchronously.
	RAB$V_BIO	Block I/O: specifies that only block I/O operations are permitted. The FAB$B_ FAC field FAB$V_BRO or FAB$V_BIO option must be specified to the Open or Create service.
	RAB$V_EOF[3]	End-of-file: positions to the end of the file upon execution of the Connect service.
	RAB$V_RAH[2]	Read ahead: allocates at least two buffers for multibuffering (applies only to sequential files on disk devices).
	RAB$V_WBH[2]	Write behind: allocates at least two buffers for multibuffering (applies only to sequential files on disk devices).

[1]This field is a required input to the Connect service.

[2]This field is not supported for DECnet for OpenVMS operations.

[3]Refer to text for exceptions.

[4]Optionally, you can specify the multibuffer count using the XAB$_MULTIBUFFER_COUNT XABITM.

Table RMS–4 lists the control block fields written as output by the Connect service.

Table RMS–4 Connect Service RAB Output Fields

Field Name	Description
RAB$W_ISI	Internal stream identifier
RAB$L_STS	Completion status code (also returned in Register 0)
RAB$L_STV	Status value

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BUG_DAP
RMS$_CCR	RMS$_CDA	RMS$_CRMP
RMS$_DME	RMS$_EXT_ERR	RMS$_EXTNOTFOU
RMS$_FAB	RMS$_GBC	RMS$_IAL
RMS$_IFA	RMS$_IFI	RMS$_KRF
RMS$_MBC	RMS$_NET	RMS$_NETFAIL
RMS$_NORMAL	RMS$_OPNOTSUP	RMS$_PENDING
RMS$_RAB	RMS$_RFM	RMS$_ROP
RMS$_RPL	RMS$_STR	RMS$_SUC
RMS$_SUP	RMS$_SUPPORT	

$CREATE

The Create service constructs a new file according to the attributes you specify in the FAB. If any XABs are chained to the FAB, then the characteristics described in the XABs are applied to the file. This service performs implicit Open and Display services.

Format

SYS$CREATE fab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab
OpenVMS usage: fab
type: longword (unsigned)
access: modify
mechanism: by reference

FAB control block whose contents are to be used as indirect arguments for the Create service call. The **fab** argument is the address of the FAB control block.

err
OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc
OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Create service first uses the information from the specified FAB. If an allocation control XAB is present, however, its allocation quantity (XAB$L_ALQ), allocation options (XAB$B_AOP, only for the XAB$V_CTG and XAB$V_CBT options), bucket size (XAB$B_BKZ), and default extension quantity (XAB$W_DEQ) fields are used instead of the corresponding fields of the FAB. When either key definition or allocation XABs are present, they can be grouped in any order. If a name block (NAM) is also connected to the FAB, RMS fills in its fields with information about the created file. The Create service leaves the file opened.

When a search list logical name is used, the file is placed in the first resultant search list file specification unless the create-if (FAB$V_CIF) option is specified. If you select the FAB$V_CIF option, RMS searches all search list file specifications to locate the file. If it finds the file, RMS opens it rather than create a new file. If RMS does not find the file, it creates a new file using the first resultant search list file specification.

You do not have to explicitly specify the FAB$V_PUT option when invoking a Create service because *write* is the default access mode when you create a file.

Table RMS–5 lists the control block fields read as input by the Create service. For additional information on the fields accessed by this service, see Chapter 4.

Table RMS–5 Create Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
FAB$B_ACMODES		File access modes.
	FAB$V_CHAN_MODE[3]	Assigns the channel access mode by setting either the FAB$V_UFO or the FAB$V_NFS bit in the FAB$L_FOP field (see Section 4.17). If neither bit is set, this field can be used to override the access mode protection for a specified I/O operation. See Section 4.8.
	FAB$V_LNM_MODE[3]	Specifies the logical name translation access mode.
FAB$L_ALQ		Allocation quantity; ignored if an allocation XAB is present.
FAB$B_BKS		Bucket size; ignored if an allocation XAB is present.
FAB$W_BLS		Block size (applies to magnetic tape only).
FAB$W_DEQ		Default file extension quantity; ignored if an allocation XAB is present.
FAB$L_DNA		Default file specification string address.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–5 (Cont.) Create Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
FAB$B_DNS		Default file specification string size.
FAB$B_FAC		File access.
	FAB$V_BIO	Block I/O access to file.
	FAB$V_BRO	Block or record I/O access to file.
	FAB$V_DEL	Delete access to file.
	FAB$V_GET[1]	Read access to file.
	FAB$V_PUT[1]	Write access to file and explicit file extension.
	FAB$V_TRN	Truncate access to file.
	FAB$V_UPD	Update access to file and explicit file extension.
FAB$L_FNA[2]		File specification string address.
FAB$B_FNS[2]		File specification string size.
FAB$L_FOP		File-processing options.
	FAB$V_ASY	Asynchronous; indicates that the specified task is to be done asynchronously.
	FAB$V_CBT	Contiguous best try: indicates that the file is to be allocated contiguously on a ‘‘best effort’’ basis. To specify a single extent, use the FAB$V_CTG option.
	FAB$V_CIF	Create-if: opens a file if it already exists or creates a file if it does not already exist.
	FAB$V_CTG	Contiguous: indicates that the space for a file is to be allocated contiguously.
	FAB$V_DFW[3]	Deferred write: writing back to the file from the modified buffer is deferred. Applies to relative and indexed files and sequential files opened for shared access.
	FAB$V_DLT	Delete: indicates that the file is to be deleted when closed.
	FAB$V_MXV	Maximize version: indicates that the created file be given the specific version number requested or a version number that is one greater than the highest version number of an existing file.

[1]These are the default values supplied by RMS.

[2]These fields must be specified unless you select the FAB$V_TMD or the FAB$V_TMP option.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–5 (Cont.) Create Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
	FAB$V_NAM[3]	Name block inputs: indicates that the NAM$W_DID and NAM$T_DVI fields in the specified NAM block are used as input.
	FAB$V_NFS[3]	Non-file-structured: indicates that the accessed volume is to be processed in a non-file-structured manner.
	FAB$V_OFP	Output file parse: specifies that the resultant file specification string of the related file, if used, is to provide file name and file type defaults only.
	FAB$V_POS	Current position (applies to magnetic tapes only).
	FAB$V_RCK	Read-check: indicates that transfers from disk are to be followed by a read-compare operation.
	FAB$V_RWC	Rewind on close (applies to magnetic tape only).
	FAB$V_RWO	Rewind on open (applies to magnetic tape only).

Note

If you specify this option as input to the Create service, RMS overwrites the tape beginning with the first file.

	FAB$V_SCF	Submit command file: indicates that the file is to be submitted as a batch-command file to the process default batch queue (SYS$BATCH) when the file is closed (applies to sequential files only).
	FAB$V_SPL	Spool: indicates that the file is to be spooled to the process default print queue (SYS$PRINT) when the file is closed (applies to sequential files only).
	FAB$V_SQO	Sequential only: indicates that the file can be processed in a sequential manner only, usually to enable DECnet for OpenVMS file transfer.
	FAB$V_SUP	Supersede: allows an existing file to be superseded by a new file of the same name, type, and version.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–5 (Cont.) Create Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
	FAB$V_TEF	Truncate at end of file: indicates that the unused space allocated to a file is deallocated when that file is closed (applies to sequential files only).
	FAB$V_TMD	Temporary marked for delete: indicates that a temporary file is to be created, and then deleted when the file is closed.
	FAB$V_TMP	Temporary: indicates that a temporary file is to be created and retained, but no directory entry is made for this file.
	FAB$V_UFO[3]	User file open: indicates that the file is to be created or opened only (no further processing of that file is allowed).
	FAB$V_WCK	Write-check: indicates that transfers to disk are to be followed by a read-compare operation.
FAB$B_FSZ		Fixed control area size.
FAB$W_GBC[3]		Global buffer count for shared files.
FAB$W_IFI		Internal file identifier (must be 0).
FAB$L_MRN		Maximum record number (applies to relative files only).
FAB$W_MRS		Maximum record size.
FAB$L_NAM		Name block address.
FAB$B_ORG		File organization: sequential (FAB$C_SEQ[1]), relative (FAB$C_REL), or indexed (FAB$C_IDX).
FAB$B_RAT		Record attributes.
FAB$B_RFM		Record format: fixed-length (FAB$C_FIX), variable-length (FAB$C_VAR), VFC (FAB$C_VFC), stream (FAB$C_STM), stream with line feed terminator (FAB$C_STMLF), stream with carriage return terminator (FAB$C_STMCR), or unidentified format (FAB$C_UDF[1]).
FAB$B_RTV[3]		Retrieval window size.
FAB$B_SHR		File sharing.
	FAB$V_SHRDEL	Allows other users to delete records from the file.

[1]These are the default values supplied by RMS.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–5 (Cont.) Create Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
	FAB$V_SHRGET	Allows other users to read the file; also used with the FAB$V_MSE and FAB$V_GET bits to specify a read-only global buffer cache when global buffering is enabled.
	FAB$V_MSE[3]	Allows multistream access.
	FAB$V_NIL	Prohibits any type of file sharing by other users.
	FAB$V_SHRPUT	Allows other users to write records to the file and extend it.
	FAB$V_SHRUPD	Allows other users to update records in the file and extend it.
	FAB$V_UPI	Allows one or more users write access to a shared file open for block I/O (applies to sequential files only).
FAB$L_XAB		Extended attribute block address.
	XABALL	Allocation XAB; see Chapter 8.
	XABDAT	Date and time XAB; see Chapter 9.
	XABFHC	File header characteristics XAB; see Chapter 10.
	XABITM	Item list XAB; see Chapter 11.
	XABKEY	Key definition XAB; see Chapter 13.
	XABPRO	Protection XAB; see Chapter 14.
	XABRDT	Revision date and time XAB; see Chapter 15.
	XABSUM	Summary XAB; see Chapter 17.

[3]This field is not supported for DECnet for OpenVMS operations.

Table RMS–6 lists the control block fields written as output by the Create service.

Table RMS–6 Create Service FAB and XAB Output Fields

Field Name	Option or XAB Type	Description
FAB$L_ALQ		Allocation quantity: contains actual number of blocks allocated.
FAB$B_BKS		Bucket size: applies only to relative and indexed files. When multiple areas are defined for an indexed file, the largest bucket size is returned.

(continued on next page)

Table RMS–6 (Cont.) Create Service FAB and XAB Output Fields

Field Name	Option or XAB Type	Description
FAB$W_BLS		Device block size (applies to files of sequential organization only).
FAB$W_DEQ		Default file extension quantity.
FAB$L_DEV		Device characteristics.
FAB$B_FAC		File access.
FAB$B_FSZ		Fixed-length control area size for VFC format.
FAB$W_GBC		Global buffer count.
FAB$W_IFI		Internal file identifier.
FAB$L_MRN		Maximum record number.
FAB$W_MRS		Maximum record size.
FAB$B_ORG		File organization.
FAB$B_RAT		Record attributes.
FAB$B_RFM		Record format.
FAB$L_SDC		Secondary device characteristics.
FAB$B_SHR		File sharing.
FAB$L_STS		Completion status code (also returned in register 0).
FAB$L_STV		Status value: contains the I/O channel number if the operation is successful.
FAB$L_XAB		Next XAB field.
	XABALL	Allocation XAB; see Chapter 8.
	XABDAT	Date and time XAB; see Chapter 9.
	XABFHC	File header characteristics XAB; see Chapter 10.
	XABITM	Item list XAB; see Chapter 11.
	XABKEY	Key definition XAB; see Chapter 13.
	XABPRO	Protection XAB; see Chapter 14.
	XABRDT	Revision date and time XAB; see Chapter 15.
	XABSUM	Summary XAB; see Chapter 17.

Use of the NAM Block for Creating Files

Table RMS–7 and Table RMS–8 list the NAM block fields used as input and output for the Create service (provided that the name block address field [FAB$L_NAM] is specified).

Table RMS–7 Create Service NAM Block Input Fields

Field Name	Option	Description
NAM$W_DID[1]		Directory identification (input only if the FAB$L_FOP FAB$V_NAM option is set).
NAM$T_DVI[1]		Device identification (input only if the FAB$L_FOP FAB$V_NAM option is set).
NAM$L_ESA		Expanded string area address.
NAM$B_ESS		Expanded string area size.
NAM$B_NOP		NAM block options.
	NAM$V_PWD	Password: indicates that a password contained in a DECnet for OpenVMS access control string, if present in a file specification, is to be left unaltered in the expanded and resultant strings (instead of being replaced by the word ‘‘password’’).
	NAM$V_NOCONCEAL	Do not conceal device name: indicates that when a concealed device logical name is present, the concealed device logical name is to be replaced by the actual physical device name in the resultant string.
NAM$L_RLF		Related file NAM block address.
NAM$L_RSA		Related file NAM block resultant string address.
NAM$B_RSL		Related file NAM block resultant string length.
NAM$L_FNB		Related file NAM block file name status bits.
NAM$B_RSS		Resultant string area size.

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–8 Create Service NAM Block Output Fields

Field Name	Description
NAM$W_DID[1]	Directory identification.
NAM$T_DVI[1]	Device identification.
NAM$B_ESL	Expanded string length. If the NAM$L_ESA field and the NAM$B_ESS field are nonzero, and you do not select the FAB$V_NAM option, or if the NAM$W_DID field is clear when you invoke the Create service, RMS copies the expanded file specification string to the buffer specified by the NAM$L_ESA field.
NAM$W_FID[1]	File identification.
NAM$L_FNB	File name status bits. This is an output field from the Create service only if the NAM bit in FAB$L_FOP field is clear, or if the NAM$W_DID field is clear when you invoke the Create service.
NAM$B_RSL	Resultant string length. If the NAM$L_RSA field and the NAM$B_RSS field are both nonzero on input, the resultant file specification is copied to the buffer specified by NAM$L_RSA.
NAM$L_DEV	Address of file specification device string.
NAM$B_DEV	Size of file specification device string.
NAM$L_DIR	Address of file specification directory string.
NAM$B_DIR	Size of file specification directory string.
NAM$L_NAME	Address of file specification name string.
NAM$B_NAME	Size of file specification name string.
NAM$L_NODE	Address of file specification node string.
NAM$B_NODE	Size of file specification node string.
NAM$L_TYPE	Address of file specification type string.
NAM$B_TYPE	Size of file specification type string.
NAM$L_VER	Address of file specification version string.
NAM$B_VER	Size of file specification version string.

[1]This field is not supported for DECnet for OpenVMS operations.

Creating Files with the Create-If Option

Note that setting the create-if (FAB$V_CIF) option in the FAB$L_FOP field specifies that if a new file has the same file specification as an existing file, RMS opens the existing file and no new file is created. Some fields in the FAB, such as the file organization (FAB$B_ORG) and record format (FAB$B_RFM) fields, are input to a Create service, but are output from an Open service. For example, the indexed file organization could be specified in the FAB$B_ORG field on a create-if operation. However, if an existing sequential file has the same file specification as the indexed file that the user is attempting to create, then the existing file is opened and the FAB$B_ORG field is set to sequential.

Creating Indexed Files

An indexed file consists of a prolog, with which it begins, and one or more index structures. RMS supplies the prolog with certain information about the file, including file attributes.

RMS supports two prolog levels, called Prolog 2 and Prolog 3. Unlike Prolog 2 files, Prolog 3 files allow for file compression and additional key types. For compatibility with RMS–11 data files that are transported or copied (without conversion) between systems, you may want to choose Prolog 2.

If you want to create a Prolog 3 file, you must be sure that records in the file are not larger than 32,224 bytes and, if the primary key is segmented, that the segments of the primary key do not overlap (one or more bytes of the record are used in more than one segment). If the primary key contains overlapping segments, you can consider using that key as an alternate key instead of a primary key or you can either request, or let RMS assign you, a Prolog 2 indexed file.

Prolog 3 is the default prolog for RMS, although RMS creates a Prolog 2 file *only* if the key characteristics are not compatible with Prolog 3 files. You can, however, override this default by requesting a specific prolog version. The option you choose in requesting a specific prolog level affects the behavior of RMS with regard to creating the file and returning error messages.

If you explicitly request a prolog version using the XABKEY XAB$B_PROLOG field in an application program, and if other file characteristics are incompatible with that prolog, then RMS returns an error message and does not attempt to create the file. For example, if you explicitly specify Prolog 2 in the XAB$B_PROLOG field and have requested a key type that is available only with Prolog 3 (such as an 8-byte integer key type), an error is returned and the file is not created.

However, if a specific prolog version is not explicitly requested in the XAB$B_PROLOG field, RMS selects the greatest prolog level that can support the specified key characteristics and does not return an error completion code.

In summary, there are two ways in which you can specify a particular prolog version:

- Specify the XAB$B_PROLOG field in a XABKEY block in an application program, affecting only the file being created.
- Use the DCL command SET RMS_DEFAULT/PROLOG to change the process default.

If you do not specify the XAB$B_PROLOG field in your application program, RMS examines your process defaults to check for prolog information. If this information is not specified in your process defaults, RMS examines the system defaults. If no prolog information is specified at the system level, RMS attempts to create a Prolog 3 file.

You need not be concerned with the distinctions between Prolog 2 and Prolog 1 files. To create an indexed file with a prolog version other than Prolog 3, specify a Prolog 2 file. If all keys in the file are string keys, RMS provides a default of Prolog 1; in all other cases, Prolog 2 is the default.

Note

String keys include the STRING, DSTRING, COLLATED, and DCOLLATED data-type keys.

If the file contains all string keys and Prolog 2 is requested, RMS attempts to create a Prolog 1 file only if no binary keys are present.

Note that RMS–11 and previous versions of RMS return error messages if requested to process Prolog 3 files.

If a failure is indicated, the file may be created, but it may not be opened for processing, depending on the nature of the failure.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACS	RMS$_ACT	RMS$_AID
RMS$_ALN	RMS$_ALQ	RMS$_AOP
RMS$_ATR	RMS$_ATW	RMS$_BKS
RMS$_BKZ	RMS$_BLN	RMS$_BUG
RMS$_BUG_DAP	RMS$_BUG_DDI	RMS$_CDA
RMS$_CHN	RMS$_COD	RMS$_CRE
RMS$_CREATED	RMS$_CRE_STM	RMS$_CRMP
RMS$_DAN	RMS$_DEV	RMS$_DFL
RMS$_DIR	RMS$_DME	RMS$_DNA
RMS$_DNF	RMS$_DNR	RMS$_DTP
RMS$_DVI	RMS$_ENQ	RMS$_ENV
RMS$_ESA	RMS$_ESS	RMS$_EXENQLM
RMS$_EXP	RMS$_EXTNOTFOU	RMS$_FAB
RMS$_FEX	RMS$_FLG	RMS$_FLK
RMS$_FNA	RMS$_FNF	RMS$_FNM
RMS$_FOP	RMS$_FSZ	RMS$_FUL
RMS$_GBC	RMS$_IAL	RMS$_IAN
RMS$_IBK	RMS$_IFA	RMS$_IFI
RMS$_IFL	RMS$_IMX	RMS$_IOP

RMS$_KNM	RMS$_KSI	RMS$_LAN
RMS$_LNE	RMS$_MRN	RMS$_MRS
RMS$_NAM	RMS$_NET	RMS$_NETFAIL
RMS$_NOD	RMS$_NORMAL	RMS$_NPK
RMS$_ORG	RMS$_POS	RMS$_PRV
RMS$_QUO	RMS$_RAT	RMS$_REF
RMS$_RFM	RMS$_RLF	RMS$_RPL
RMS$_RSS	RMS$_RST	RMS$_RUNDOWN
RMS$_SEG	RMS$_SEMANTICS	RMS$_SHR
RMS$_SIZ	RMS$_STR	RMS$_SUC
RMS$_SUP	RMS$_SUPERSEDE	RMS$_SUPPORT
RMS$_SYN	RMS$_SYS	RMS$_UPI
RMS$_VER	RMS$_WLK	RMS$_WPL
RMS$_XAB		

$DELETE

The Delete service removes an existing record from a relative or indexed file. You cannot use this service when processing sequential files.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Delete service on OpenVMS Alpha systems. ♦

Format

SYS$DELETE rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Delete service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

A Delete service always applies to the current record. Therefore, immediately before invoking the Delete service, you must establish the current record by issuing a Find or Get service.

Table RMS–9 lists the control block fields read as input by the Delete service. For additional information on the fields accessed by this service, see Part II.

Table RMS–9 Delete Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Delete service asynchronously.
	RAB$V_FDL	Fast delete (applies to indexed files).

Table RMS–10 lists the control block fields written as output by the Delete service.

Table RMS–10 Delete Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BUG
RMS$_BUG_DAP	RMS$_CDA	RMS$_CHK
RMS$_CUR	RMS$_DME	RMS$_DNR
RMS$_EXT_ERR	RMS$_FAC	RMS$_FTM
RMS$_IAL	RMS$_IBF	RMS$_IOP
RMS$_IRC	RMS$_ISI	RMS$_NET
RMS$_NETFAIL	RMS$_NORMAL	RMS$_OPNOTSUP
RMS$_PENDING	RMS$_RAB	RMS$_RNL
RMS$_RPL	RMS$_RRV	RMS$_RSA
RMS$_STR	RMS$_SUC	RMS$_SUP
RMS$_SUPPORT	RMS$_SYS	RMS$_TRE
RMS$_WLK		

$DISCONNECT

The Disconnect service breaks the connection between a RAB and a FAB, thereby terminating a record stream. All system resources, such as I/O buffers and data structure space, are deallocated.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Disconnect service on OpenVMS Alpha systems. ♦

Format

SYS$DISCONNECT rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Disconnect service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Close service (see $CLOSE) performs an implied disconnect for all record streams connected to the FAB. Thus, you need not explicitly issue a Disconnect service prior to closing the file. However, if more than one RAB is connected to a single FAB, then you must explicitly disconnect the desired RAB in order to terminate a particular record stream and leave the others active.

Table RMS–11 lists the control block fields read as input by the Disconnect service. For additional information on the fields accessed by this service, see Part II.

Table RMS–11 Disconnect Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing option.
	RAB$V_ASY	Asynchronous. Performs Disconnect service asynchronously.

Table RMS–12 lists the control block fields written as output by the Disconnect service.

Table RMS–12 Disconnect Service RAB Output Fields

Field Name	Description
RAB$W_ISI	Internal stream identifier (zeroed).
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BUG_DAP	RMS$_CDA
RMS$_CRC	RMS$_DME	RMS$_DNR
RMS$_EXT_ERR	RMS$_ISI	RMS$_NET
RMS$_NETFAIL	RMS$_NORMAL	RMS$_OPNOTSUP
RMS$_PENDING	RMS$_RAB	RMS$_RSA
RMS$_STR	RMS$_SUC	RMS$_SUP
RMS$_SUPPORT	RMS$_SYS	RMS$_WBE
RMS$_WER	RMS$_WLK	

$DISPLAY

The Display service retrieves file attribute information about a file and places this information in fields in the FAB, in XABs chained to the FAB, and in a NAM block (if one is requested).

Format

SYS$DISPLAY fab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage: fab
type: longword (unsigned)
access: modify
mechanism: by reference

FAB control block whose contents are to be used as indirect arguments for the Display service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

A file must be open for access by a Create or Open service before the Display service can be invoked.

RMS places the file attribute information in the corresponding fields of the FAB and specified XABs. If the FAB$L_NAM field contains a valid NAM block address, certain NAM block fields are filled in, including the resultant string, and the NAM$B_NOP options are examined.

Note that the Open and Create services automatically perform an implicit Display service (see $OPEN and $CREATE).

Table RMS–13 lists the control block fields read as input by the Display service. For additional information on the fields accessed by this service, see Part II.

Table RMS–13 Display Service FAB and NAM Input Fields

Field Name	Option	Description
FAB$W_IFI		Internal file identifier.
FAB$L_NAM[1]		Name block address.
FAB$L_XAB[1]		Extended attribute block address.
NAM$B_NOP		NAM block options.
	NAM$V_PWD	Password: indicates that a password contained in a DECnet for OpenVMS access control string, if present in a file specification, is to be left unaltered in the expanded and resultant strings (instead of being replaced by the word ''password'').
	NAM$V_NOCONCEAL	Do not conceal device name: indicates that when a concealed device logical name is present, the concealed device logical name is to be replaced by the actual physical device name (and directory, if present) in the resultant string.

[1] If you want information about a particular XAB or NAM block, you must pass it to the Display service as input.

Table RMS–14 lists the control block fields written as output by the Display service.

Table RMS–14 Display Service FAB, NAM, and XAB Output Fields

Field Name	XAB Type	Description
FAB$L_ALQ		Allocation quantity in blocks.
FAB$B_BKS		Bucket size.
FAB$W_BLS		Block size.
FAB$W_DEQ		Default file extension quantity.
FAB$L_DEV		Device characteristics.
FAB$B_FAC		File access.
FAB$B_FSZ		Fixed control area size.
FAB$W_GBC		Global buffer count.
FAB$L_MRN		Maximum record number.
FAB$W_MRS		Maximum record size.
FAB$B_ORG		File organization.
FAB$B_RAT		Record attributes.
FAB$B_RFM		Record format.
FAB$B_RTV		Retrieval window size.
FAB$B_SHR		File sharing.
FAB$L_STS		Completion status code (also returned in register 0).
FAB$L_STV		Status value.
FAB$L_XAB		Next XAB address.
	XABALL	Allocation XAB; see Chapter 8.
	XABDAT	Date and time XAB; see Chapter 9.
	XABFHC	File header characteristics XAB; see Chapter 10.
	XABITM	Item list XAB; see Chapter 11.
	XABKEY	Key definition XAB; see Chapter 13.
	XABPRO	Protection XAB; see Chapter 14.
	XABRDT	Revision date and time XAB; see Chapter 15.
	XABSUM	Summary XAB; see Chapter 17.
NAM$W_DID		Directory identification.
NAM$T_DVI		Device identification.
NAM$W_FID		File identification.
NAM$L_FNB		File name status bits.
NAM$B_RSL		Resultant string length: indicates the length of the resultant string that is written into the buffer whose address is contained in the NAM$L_RSA field (if the NAM$L_RSA and NAM$B_RSS fields are nonzero).

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_AID	RMS$_ATR
RMS$_BLN	RMS$_BUG_DAP	RMS$_CDA
RMS$_COD	RMS$_DME	RMS$_DNR
RMS$_ESA	RMS$_ESL	RMS$_ESS
RMS$_FAB	RMS$_IFI	RMS$_IMX
RMS$_NET	RMS$_NETFAIL	RMS$_NORMAL
RMS$_OK_NOP	RMS$_PLG	RMS$_PRV
RMS$_REF	RMS$_RPL	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_XAB		

$ENTER

The Enter service inserts a file name in a directory.

Format

SYS$ENTER fab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage: fab
type: longword (unsigned)
access: modify
mechanism: by reference

FAB control block whose contents are to be used as indirect arguments for the Enter service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Enter service function is performed automatically by the Create service unless you select the FAB$V_TMP option or the FAB$V_TMD option. The Enter service, however, allows you to perform this step separately. Note that the file must be closed before you invoke the Enter service (FAB$W_IFI must be 0).

When you enter a file name in a directory, no file associated with the FAB can be open and no wildcard characters can be used.

The Enter service requires many NAM block fields as input. You normally precede the Enter service with an Open, Create, or Parse service (see $PARSE) and a Search service (see $SEARCH), specifying the same FAB and NAM block for each service.

The optional resultant string is moved to the buffer described by the NAM$L_RSA and NAM$B_RSS fields (only if both these fields are nonzero). If the file version number of the name string described by the expanded string length and address fields of the NAM block is omitted or contains a 0, the Enter service scans the entire directory. It assigns a version number that is one higher than the highest found (or 1 if none is found).

The Enter service is not supported for DECnet for OpenVMS operations on remote files between two OpenVMS systems.

Table RMS–15 lists the control block fields read as input by the Enter service. For additional information on the fields accessed by this service, see Part II.

Table RMS–15 Enter Service FAB and NAM Input Fields

Field Name	Description
FAB$W_IFI	Internal file identifier (must be 0).
FAB$L_NAM	Name block address.
NAM$W_DID[1]	Directory identification: identifies the directory in which the file name is to be entered.
NAM$T_DVI[1]	Device identification: identifies the device containing the directory in which the file name is to be entered.
NAM$L_ESA	Expanded string area address: contains file name, type, and version to be entered.
NAM$B_ESL	Expanded string length.
NAM$W_FID[1]	File identification: identifies the file to be entered into the directory.
NAM$L_RSA	Resultant string area address.
NAM$B_RSS	Resultant string size.

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–16 lists the control block fields written as output by the Enter service.

Table RMS–16 Enter Service FAB and NAM Output Fields

Field Name	Description
FAB$L_STS	Completion status code (also returned in register 0).
FAB$L_STV	Status value.
NAM$B_RSL	Resultant string length.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_BLN	RMS$_CDA	RMS$_CHN
RMS$_DEV	RMS$_DME	RMS$_DNF
RMS$_DNR	RMS$_DVI	RMS$_ENT
RMS$_ESA	RMS$_ESL	RMS$_FAB
RMS$_FNF	RMS$_IFI	RMS$_NAM
RMS$_NORMAL	RMS$_PRV	RMS$_RSL
RMS$_RSS	RMS$_RST	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS	RMS$_WLD	RMS$_WLK

$ERASE

The Erase service deletes a disk file and removes the file's directory entry specified in the path to the file. If additional directory entries have been created for this file by the Enter service, you must use the Remove service to delete them.

Format

SYS$ERASE fab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage:	fab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

FAB control block whose contents are to be used as indirect arguments for the Erase service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

Using the Erase service to delete a file releases the file's allocated space for use by another file. The Erase service does not physically remove the data (as does overwriting or zeroing).

Note that the file must be closed before you invoke the Erase service (FAB$W_IFI must be zero). You can, however, delete a file that is currently open, if you issue a Close service and specify the FAB$L_FOP field FAB$V_DLT option. RMS does not allow you to delete files from magnetic tape volumes; they must be overwritten.

If a search list logical name is specified, the file is deleted only if it is found in the first resulting search list file specification.

Table RMS–17 lists the control block fields read as input by the Erase service. For additional information on the fields accessed by this service, see Part II.

Table RMS–17 Erase Service FAB and NAM Input Fields

Field Name	Option	Description
FAB$L_DNA		Default file specification string address.
FAB$B_DNS		Default file specification string size.
FAB$L_FNA		File specification string address.
FAB$B_FNS		File specification string size.
FAB$L_FOP		File-processing options.
	FAB$V_NAM[1]	NAM block inputs: allows use of the NAMW_DID, NAMT_DVI, and NAM$W_FID fields.
FAB$W_IFI		Internal file identifier (must be 0).
FAB$L_NAM		Name block address.
NAM$W_DID[1]		Directory identification (input only if the FAB$L_FOP field FAB$V_NAM bit is set).
NAM$T_DVI[1]		Device identification (input only if the FAB$L_FOP field FAB$V_NAM bit is set).
NAM$L_ESA		Expanded string area address.
NAM$B_ESS		Expanded string area size.
NAM$W_FID[1]		File identification (input only if the FAB$L_FOP field FAB$V_NAM bit is set).
NAM$L_RLF		Related file NAM block address.
NAM$L_RSA		Related file resultant string address.
NAM$B_RSS		Related file resultant string size.
NAM$L_FNB		Related file filename status bits.

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–18 lists the control block fields written as output by the Erase service. Note that the NAM block fields are used for output only if the name block address field is specified in the FAB.

Table RMS–18 Erase Service FAB and NAM Output Fields

Field Name	Description
FAB$L_STS	Completion status code (also returned in register 0).
FAB$L_STV	Status value.
NAM$W_DID[1]	Directory identification.
NAM$T_DVI[1]	Device identification.
NAM$B_ESL	Expanded string length. If the NAM$L_ESA field and the NAM$B_ESS field are nonzero, and if the FAB$V_NAM bit is clear or the NAM$W_DID field contains a zero, RMS copies the expanded file specification string to the buffer specified by the input NAM$L_ESA field.
NAM$L_FNB	File name status bits.
NAM$B_RSL	Resultant string length (if NAM$L_RSA and NAM$B_RSS are both nonzero on input, the resultant file specification is copied to the buffer specified by NAM$L_RSA).

[1]This field is not supported for DECnet for OpenVMS operations.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACS	RMS$_BLN	RMS$_BUG_DAP
RMS$_BUG_DDI	RMS$_CDA	RMS$_CHN
RMS$_DEV	RMS$_DIR	RMS$_DME
RMS$_DNA	RMS$_DNF	RMS$_DNR
RMS$_DVI	RMS$_ESA	RMS$_ESS
RMS$_FAB	RMS$_FNF	RMS$_FNM
RMS$_IFI	RMS$_IOP	RMS$_LNE
RMS$_MKD	RMS$_NAM	RMS$_NET
RMS$_NETFAIL	RMS$_NOD	RMS$_NORMAL
RMS$_PRV	RMS$_QUO	RMS$_RLF
RMS$_RSS	RMS$_RST	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPERSEDE
RMS$_SUPPORT	RMS$_SYN	RMS$_TYP
RMS$_VER	RMS$_WLK	

$EXTEND

The Extend service increases the amount of space allocated to a disk file. This service is most useful for extending relative files and indexed files when you are doing block I/O transfers using the Write service.

Format

SYS$EXTEND fab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage:	fab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

FAB control block whose contents are to be used as indirect arguments for the Extend service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

When a program adds data to a file using *record* I/O operations, RMS invokes the Extend service to provide additional file space, if needed, regardless of the file organization. But, when data is added using *block* I/O operations, RMS provides additional file space for sequential files *only*. To obtain additional file space when using block I/O, the program must invoke the Extend service.

You might also want to consider using the Extend service for all file organizations when you want to extend a file for performance reasons, such as placing a large file extent (an extended part of a file) contiguous with the file.

In order for a program to invoke the Extend service, it must open the file (FAB$W_IFI must not be 0); otherwise, an error occurs. RMS also requires that the file access field (FAB$B_FAC) specify either put access (FAB$V_PUT) or update access (FAB$V_UPD) before permitting file extensions.

The program uses the appropriate allocation quantity field (FAB$L_ALQ or XAB$L_ALQ) to specify the number of blocks that RMS uses to extend a file.

You can specify other attributes, such as how and where the additional space is allocated. For example, you can specify whether you want the additional space allocated contiguously. If you specify contiguous space and the additional space is not available, the operation fails.

When the program uses an allocation control XAB, the XAB's allocation quantity (XAB$L_ALQ) and allocation options (XABB_AOP, XABV_CBT, and XAB$V_CTG bits) are used instead of the corresponding options specified in the FAB. You can specify multiple XABs to extend separate areas of indexed files.

If no XABALL is present on the extend of an indexed file that is opened for I/O record access, RMS extends Area 0 of the file. If block I/O access is specified, RMS extends the file but does not affect the area descriptor. If no XABALL is present and you have not specified either of the contiguity options (FABV_CBT, FABV_CGT), RMS requests placement control to position the extension as near as possible to the last block allocated to the file.

Table RMS–19 lists the control block fields read as input by the Extend service. For additional information on the fields accessed by this service, see Part II of this manual.

Table RMS–19 Extend Service FAB Input Fields

Field Name	Description
FAB$L_ALQ	Allocation quantity. Ignored if an allocation XAB is present.
FAB$L_FOP	File-processing options. Checked to see whether the FAB$V_CTG or FAB$V_CBT bit is set to indicate contiguous allocation (ignored for allocation XAB).
FAB$W_IFI	Internal file identifier (must not be 0).
FAB$L_XAB	Extended attribute block address. Only an allocation XAB (XABALL) is processed.

Table RMS–20 lists the control block fields written as output by the Extend service.

Table RMS–20 Extend Service FAB Output Fields

Field Name	Description
FAB$L_ALQ	Allocation quantity. Contains the actual extension allocation value if no allocation XAB is present.
FAB$L_STS	Completion status code (also returned in register 0).
FAB$L_STV	Status value. Contains the total number of blocks allocated, totaled across all allocation XABs.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_AID	RMS$_ALN
RMS$_ALQ	RMS$_AOP	RMS$_ATR
RMS$_ATW	RMS$_BLN	RMS$_BUG_DAP
RMS$_CDA	RMS$_COD	RMS$_DME
RMS$_EXT	RMS$_FAB	RMS$_FAC
RMS$_FUL	RMS$_IFI	RMS$_IMX
RMS$_IOP	RMS$_LEX	RMS$_NET
RMS$_NETFAIL	RMS$_NORMAL	RMS$_PLG
RMS$_RPL	RMS$_STR	RMS$_SUC
RMS$_SUP	RMS$_SUPPORT	RMS$_SYS
RMS$_WBE	RMS$_WER	RMS$_WLK
RMS$_WPL	RMS$_XAB	

$FIND

The Find service locates a specified record in a file and returns its record file address in the RAB$W_RFA field of the RAB. The Find service can be used with all file organizations.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Find service on OpenVMS Alpha systems. ♦

Format

SYS$FIND rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Find service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Find service gives you the following functional capabilities:

- You can skip records when you are accessing a file sequentially by making successive invocations of the Find service.
- You can establish the *current record* context prior to invoking an Update, Delete, or Truncate service.
- You can establish a random access starting point in a file for subsequent sequential access operations such as the Get service.

When you follow the Find service with a sequential access operation, such as the Get service, the *current record* context is established by the Find service and the sequential access operation establishes a new *sequential access* context. Conversely, when you follow the Find service with a nonsequential access operation such as a Delete service or an Update service, the sequential access context remains the same as it was prior to the Find service.

RAB Control Block Fields

Table RMS–21 lists the control block fields read as input by the Find service. For additional information on the fields accessed by this service, see Part II.

Table RMS–21 Find Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_KBF		Key buffer address (used only if the RAB$B_RAC field contains RAB$C_KEY or if RAB$B_RAC contains RAB$C_SEQ and the RAB$L_ROP field RAB$V_LIM option is set).
RAB$B_KRF		Key of reference (used only with indexed files and if RAB$B_RAC contains RAB$C_KEY.)
RAB$B_KSZ		Key size (used only if the RAB$B_RAC field contains RAB$C_KEY or if RAB$B_RAC contains RAB$C_SEQ and the RAB$L_ROP field RAB$V_LIM option is set).
RAB$L_PBF[1]		Prompt buffer address (applies to terminal devices only).
RAB$B_PSZ[1]		Prompt buffer size (applies to terminal devices only).
RAB$B_RAC		Record access mode (RABC_SEQ, RABC_KEY, RAB$C_RFA)[2].
RAB$W_RFA		Record file address (used only if the RAB$B_RAC field contains RAB$C_RFA).

[1]This field is not supported for DECnet for OpenVMS operations.

[2]The default for the RAB$B_RAC field is RAB$C_SEQ.

(continued on next page)

Table RMS–21 (Cont.) Find Service RAB Input Fields

Field Name	Option	Description
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Find services asynchronously.
	RAB$V_CVT[1]	Convert: changes characters to uppercase for a Find service to a terminal device.
	RAB$V_KGE[3]	Key is greater than or equal to compared value (applies only to indexed files).
	RAB$V_KGT[4]	Key is greater than compared value (applies only to indexed files). If neither RAB$V_KGE nor RAB$V_KGT is specified, a key equal match is made.
	RAB$V_LIM	Limit: the key value described by the KBF and KSZ fields is compared to the value in the sequentially-accessed record.
	RAB$V_NLK	No lock: specifies that the record accessed through the Find service is not to be locked.
	RAB$V_NXR	Nonexistent record processing: specifies that if the record accessed directly through a Find service does not exist, the service is to be performed anyway.
	RAB$V_PMT[1]	Prompt indicates that the contents of the prompt buffer are to be used as a prompt for a Find service to a terminal device.
	RAB$V_PTA[1]	Purge type-ahead buffer: eliminates any information that may be in the type-ahead buffer for a Find service to a terminal device.
	RAB$V_RAH[1]	Read ahead: used with multiple buffers to indicate read-ahead operations (sequential files only).
	RAB$V_REA	Lock for read: allows other users read access to the record.
	RAB$V_REV	Reverses search direction for random keyed access operations when used with either RAB$V_KGE or RAB$V_KGT (applies only to indexed files).
	RAB$V_RLK	Read of locked record allowed: specifies that a locked record can be read by other users.

[1]This field is not supported for DECnet for OpenVMS operations.

[3]This symbolic offset is logically synonymous with RAB$V_EQNXT.

[4]This symbolic offset is logically synonymous with RAB$V_NXT.

(continued on next page)

Table RMS–21 (Cont.) Find Service RAB Input Fields

Field Name	Option	Description
	RAB$V_RNE[1]	Read no echo: indicates that input data entered on the keyboard is not displayed on the terminal device.
	RAB$V_RNF[1]	Read no filter: indicates that Ctrl/U, Ctrl /R, and DELETE are not to be considered control commands on terminal input, but are to be passed to the application program.
	RAB$V_RRL	Read regardless of lock: read the record even if another stream has locked the record.
	RAB$V_TMO[1]	Timeout: indicates that the contents of the timeout period field (RAB$B_TMO) is to be used on a Find request for a locked record (when the RAB$V_WAT option is also specified) or for a terminal or mailbox device.
	RAB$V_ULK	Manual unlocking: specifies that a record cannot be unlocked automatically.
	RAB$V_WAT	Wait: if record is locked, wait until it is available.
RAB$B_TMO[1]		Timeout period: indicates the maximum number of seconds that RMS can spend to complete a Find request.

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–22 lists the control block fields written as output by the Find service.

Table RMS–22 Find Service RAB Output Fields

Field Name	Description
RAB$L_BKT	Bucket code: set to the relative record number for sequentially-accessed relative files.
RAB$W_RFA	Record file address.
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

The record address (RAB$L_RBF) field and the record size (RAB$W_RSZ) field are undefined after a Find service.

RAB64 Control Block Fields (Alpha Only)

Alpha

Table RMS–23 lists the Alpha-only RAB64 control block fields read as input by the Find service. These fields are comparable to the RAB fields described in Table RMS–21. For additional information on the fields accessed by this service, see Part II.

Table RMS–23 Find Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$B_BLN	This field must be initialized to RAB64$C_BLN64 in order for RAB64 fields to be used.
RAB64$W_ISI	Internal stream identifier (required). Equates to RAB$W_ISI.
RAB64$L_KBF	Key buffer address. This field must contain −1 if you want to use RAB64$PQ_KBF. For 32-bit addressing, this field equates to RAB$L_KBF (see Table RMS–21).
RAB64$PQ_KBF	Key buffer 64-bit address (used if RAB64$L_KBF contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$B_KRF	Key of reference. Equates to RAB$B_KRF (see Table RMS–21).
RAB64$B_KSZ	Key buffer size. Equates to RAB$B_KSZ (see Table RMS–21).
RAB64$L_PBF[1]	Prompt buffer address. Equates to RAB$L_PBF (see Table RMS–21).
RAB64$B_PSZ[1]	Prompt buffer size. Equates to RAB$B_PSZ (see Table RMS–21).
RAB64$B_RAC	Record access mode. Equates to RAB$B_RAC (see Table RMS–21).
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA (see Table RMS–21).
RAB64$L_ROP	Record-processing options. Equates to RAB$L_ROP and options described in Table RMS–21. Options are identical except for the RAB64 prefix; for example, option RAB64$V_ASY equates to RAB$V_ASY.
RAB64$B_TMO[1]	Timeout period. Equates to RAB$B_TMO (see Table RMS–21).

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–24 lists the Alpha-only RAB64 control block fields written as output by the Find service. These fields are comparable to the RAB fields described in Table RMS–22.

Table RMS–24 Find Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$L_BKT	Bucket code. Equates to RAB$L_BKT (see Table RMS–22).
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA.
RAB64$L_STS	Completion status code. Equates to RAB$L_STS (see Table RMS–22).
RAB64$L_STV	Status value. Equates to RAB$L_STV.

The record address (RAB64$L_RBF) field and the record size (RAB64$W_RSZ) field are undefined after a Find service. ♦

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ANI	RMS$_ATR
RMS$_ATW	RMS$_BES	RMS$_BLN
RMS$_BUG	RMS$_BUG_DAP	RMS$_CDA
RMS$_CHK	RMS$_CONTROLC	RMS$_CONTROLY
RMS$_DEADLOCK	RMS$_DEL	RMS$_DME
RMS$_DNR	RMS$_EOF	RMS$_EXENQLM
RMS$_EXT_ERR	RMS$_FAC	RMS$_FTM
RMS$_IBF	RMS$_IDXSEARCH	RMS$_IOP
RMS$_IRC	RMS$_ISI	RMS$_KBF
RMS$_KEY	RMS$_KRF	RMS$_KSZ
RMS$_MRN	RMS$_NET	RMS$_NETFAIL
RMS$_NORMAL	RMS$_OK_ALK	RMS$_OK_DEL
RMS$_OK_LIM	RMS$_OK_RLK	RMS$_OK_RNF
RMS$_OK_RRL	RMS$_OK_WAT	RMS$_OPNOTSUP
RMS$_PBF	RMS$_PENDING	RMS$_PES
RMS$_PLG	RMS$_RAB	RMS$_RAC
RMS$_REF	RMS$_RER	RMS$_RFA
RMS$_RHB	RMS$_RLK	RMS$_RNF
RMS$_ROP	RMS$_RPL	RMS$_RRV
RMS$_RSA	RMS$_SQO	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS	RMS$_TMO	RMS$_TRE
RMS$_WBE	RMS$_WER	RMS$_WLK

$FLUSH

The Flush service writes out all modified I/O buffers and file attributes associated with the file. This ensures that all record activity up to the point at which the Flush service executes is actually reflected in the file.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Flush service on OpenVMS Alpha systems. ♦

Format

SYS$FLUSH   rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Flush service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

Explicit invocation of the Flush service is useful when an application program must be certain that all file changes are represented on disk (as in protecting against a crash).

The Close service includes an implicit Flush service, so an application program need not invoke a Flush service prior to invoking a Close service.

VAX

On VAX systems with recovery unit journaling, no explicit call to the Flush service is required because invoking the Commit service causes an implicit flush.♦

During asynchronous operations, you must wait for any I/O activity to complete before issuing a Flush service. You can also issue a Flush service after receiving notification of completion through an asynchronous system trap (AST).

On a flush, the EOF value written to the file header is only accessible to subsequent accessors. Any process that has the file open at the time of the flush cannot access the new EOF value without some explicit action.

Table RMS–25 lists the control block fields read as input by the Flush service. For additional information on the fields accessed by this service, see Part II.

Table RMS–25 Flush Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing option.
	RAB$V_ASY	Performs Flush services asynchronously.

Table RMS–26 lists the control block fields written as output by the Flush service.

Table RMS–26 Flush Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Additional status information.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BUG_DAP	RMS$_CDA
RMS$_DME	RMS$_DNR	RMS$_EXT_ERR
RMS$_ISI	RMS$_NET	RMS$_NETFAIL
RMS$_NORMAL	RMS$_OPNOTSUP	RMS$_PENDING
RMS$_RAB	RMS$_RSA	RMS$_STR

RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS	RMS$_WBE	RMS$_WER
RMS$_WLK		

$FREE

The Free service unlocks all records that were previously locked for the record stream.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Free service on OpenVMS Alpha systems. ♦

Format

SYS$FREE rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Free service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Free service unlocks all records previously locked for the record stream (see also $RELEASE). If no records are locked for the record stream, RMS returns a status code of RMS$_RNL.

Table RMS–27 lists the control block fields used as input by the Free service. For additional information on the fields accessed by this service, see Part II.

Table RMS–27 Free Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing option.
	RAB$V_ASY	Asynchronous: performs Free services asynchronously.

Table RMS–28 lists the control block fields written as output by the Free service.

Table RMS–28 Free Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Additional status information.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BUG_DAP
RMS$_EXT_ERR	RMS$_ISI	RMS$_NET
RMS$_NETFAIL	RMS$_NORMAL	RMS$_OPNOTSUP
RMS$_PENDING	RMS$_RAB	RMS$_RNL
RMS$_RSA	RMS$_STR	RMS$_SUC
RMS$_SUP	RMS$_SUPPORT	

$GET

The Get service retrieves a record from a file.

Note

When you invoke the Get service, RMS takes control of the record buffer and may modify it. RMS returns the record size and guarantees the contents only from where it accessed the record to the completion of the record.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Get service on OpenVMS Alpha systems. ♦

Format

SYS$GET rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab
OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Get service call. The **rab** argument is the address of the RAB control block.

err
OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Get service uses one of three record access modes, as specified by the record access (RAB$B_RAC) field. The three record access modes are sequential (SEQ), which is the default, random by key (KEY), and random by record file address (RFA).

Relevant Record Access Modes

The sequential access mode is relevant for all file organizations as well as for all devices. It is the only access mode allowed for nondisk devices, such as terminals, mailboxes, and magnetic tape devices. In this mode, records are retrieved from a given file in the same order in which they were written to that file. This is not the case, however, for records retrieved from indexed files. Sequential Get services for indexed files return records by key value in the specified sort order, ascending or descending. The next record's key of reference for sequential access to indexed files is established by one of the following services:

- Connect
- Rewind (see $REWIND)
- Find or Get using random access by key
- Find or Get using random access by RFA

When you use random access by key with any operations related to these services, the key of reference is established by the key of reference field (RAB$B_KRF). When you use random access by RFA in conjunction with a Find or Get service, however, the key of reference is always set to the primary key.

You can use random access by key to retrieve records by key value. For relative files and sequential files having fixed-length records, the key value is the relative record number. For indexed files, the key value depends on the data type of the specified key of reference. The key value is used to search the index of the specified key of reference to locate the desired record. A random access by key also establishes the next record for subsequent sequential retrieval. This type of access may be used in this way to establish a starting point for sequential retrieval of records at other than the beginning of the file.

You can use random access by RFA to retrieve records directly from files residing on disk devices. However, a record's address can be determined only if the record has been accessed previously. The Find, Get, and Put services each return the RFA value as output in the RAB$W_RFA field.

Random access of records in a file is prohibited when you open the file and select the FAB$V_SQO option; that is, if you specify sequential operations only.

Input from Stream Format Files

For stream format files, RMS fills the user buffer with data until a terminator is reached. If the buffer fills before a terminator is encountered, the remainder of the data preceding the terminator is discarded, and an RMS$_RTB error is returned. If the terminator for stream format (FAB$B_RFM contains FAB$C_STM) is not CRLF (carriage return followed by line feed), the terminator is stored in the buffer following the record and included in the size of the record.

Input from Terminal Devices

There are two methods of obtaining input from a terminal using RMS:

- Using the RAB$L_ROP field to define the terminal input operation. Certain options applicable to the RAB$L_ROP field are used for terminal device input, such as whether a prompt is to be displayed and whether a time limit between characters is enforced. These options may require certain information to be placed in other fields of the RAB (see Chapter 6). The maximum buffer size is 512 bytes.
- Using an item list to define the terminal input operation in conjunction with a terminal XAB (XABTRM). The ETO option of the RAB$L_ROP field must be set and the user must provide an item list in the calling program, which RMS passes to the terminal driver using the item list address and length specified in the XABTRM (see Chapter 18). This method allows use of any terminal input option supported by the terminal driver, in contrast to the subset of RAB$L_ROP options available using the other method. The maximum buffer size is 512 bytes.

RMS uses the standard terminator set when performing input operations from terminal devices. The second longword of the I/O status block used is returned in the RAB$L_STV field. The terminating character is returned in the lower word of the status value field (RAB$W_STV0); however, note that with extended terminal operations, the terminating character is in the first byte of RAB$W_STV0, not in the entire RAB$W_STV0 word. More information about the second longword of the I/O status block is available in the *OpenVMS I/O User's Reference Manual*. The RAB$W_STV0 field is device dependent for terminal devices.

The Ctrl/Z character terminates the Get service and acts as an end-of-file marker. If you enter a Ctrl/Z in response to a request for data, RMS returns the completion status code for end-of-file (RMS$_EOF). RMS takes the data you enter before the Ctrl/Z but the next Get service returns a single end-of-file error (RMS$_EOF) without accepting any further input from the device. RMS resumes taking input if you request a subsequent Get service.

RMS also supports the use of escape sequences from terminal devices that are accessed locally and have escape sequences enabled. Escape sequences for a terminal are enabled by the SET TERMINAL command (described in the *OpenVMS DCL Dictionary*). Escape sequences are returned in the record buffer. The record size (RAB$W_RSZ) is the offset within the buffer (RAB$L_RBF) to the beginning of the escape sequence. The high-order word of the status value field (RAB$W_STV2) contains the length of the escape sequence, except for extended terminal operations. In this case, the escape sequence length is returned in the first byte of RAB$W_STV2, not the entire RAB$W_STV2 word, and the terminator position is returned in the second byte of the RAB$W_STV2 word. When a partial escape sequence warning (RMS$_PES) is returned, the remaining characters in the escape sequence are returned by the next read request from the terminal.

Input from Mailbox Devices

Mailboxes may be used to synchronize activity across cooperating processes. Normally, a Get service from a mailbox device is not completed until a record is present in the mailbox. When the Get service is completed, the status value field (RAB$L_STV) contains the process identification (PID) of the process that put the record into the mailbox. However, if the timeout (TMO) record option is specified with a value of 0 in the timeout field and if no messages are present in the mailbox, then the Get service returns an end-of-file error (RMS$_EOF). This technique assures your process of an immediate return, whether or not messages are present in the mailbox.

Using the RAB$L_STV Field

The RAB$L_STV field contains additional status information for a number of situations. When the completion status is a record-too-big warning (RMS$_RTB), RAB$L_STV contains the total record size. For record-oriented devices such as terminals and mailboxes, the second longword of the I/O status block is returned in the RAB$L_STV field, whenever the completion status (RAB$L_STS) is a success code. The alternate field definitions of RAB$W_STV0 and RAB$W_STV2 are provided to reference the respective low- and high-order words of the RAB$L_STV field. The record size field (RAB$W_RSZ) always reports the amount of data returned, regardless of the completion status (RAB$L_STS). The presence of valid data on error conditions may then be detected by checking the record size field.

User Record Area

The Get service always requires the presence of a user record area, as specified by the user record buffer address and user buffer area size fields in the RAB. The traditional fields used are RAB$L_UBF and RAB$W_USZ. However, OpenVMS Alpha users can code -1 in RAB64$L_UBF to direct the Get service to use the alternative fields RAB64$PQ_UBF and RAB64$Q_USZ. RAB64$PQ_UBF can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

For undefined format files, the user buffer area size field (RAB$W_USZ or RAB64$Q_USZ) defines the amount of data to be returned on each Get service.

RAB Control Block Fields

Table RMS–29 lists the RAB control block fields read as input by the Get service. For additional information on the fields accessed by this service, see Part II.

Table RMS–29 Get Service RAB Input Fields

Field Name	Option or XAB Type	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_KBF		Key buffer address: used only if the RAB$B_RAC field contains RAB$C_KEY, or if the RAB$B_RAC field contains RAB$C_SEQ and you select the RAB$V_LIM option.

(continued on next page)

Table RMS–29 (Cont.) Get Service RAB Input Fields

Field Name	Option or XAB Type	Description
RAB$B_KRF		Key of reference: used only with indexed files and only if the RAB$B_RAC field contains RAB$C_KEY.
RAB$B_KSZ		Key buffer size: used only if the RAB$B_RAC field contains RAB$C_KEY, or if the RAB$B_RAC field contains RAB$C_SEQ and you select the RAB$V_LIM option.
RAB$L_PBF[1]		Prompt buffer address (for terminal devices only).
RAB$B_PSZ[1]		Prompt buffer size (for terminal devices only).
RAB$B_RAC		Record access mode (RABC_SEQ, RABC_KEY, RAB$C_RFA).[2]
RAB$W_RFA		Record file address: used only if the RAB$B_RAC field contains RAB$C_RFA.
RAB$L_RHB		Record header buffer: used for the fixed-length control area of VFC records.
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Get services asynchronously.
	RAB$V_CDK	Allows applications to look ahead for indexed file records with keys that duplicate the current key of reference.
	RAB$V_CVT[1]	Convert: changes characters to uppercase for a Get service to a terminal device.
	RAB$V_ETO[1]	Extended terminal operation: specifies that a XABTRM and an item list are used to define the terminal input operation. If this option is specified, no other RAB$L_ROP options applicable to terminal devices can be used.
	RAB$V_KGE[3]	Search for equal key value or next key value according to sort order (for indexed files only).

[1]This field is not supported for DECnet for OpenVMS operations.
[2]The default for the RAB$B_RAC field is RAB$C_SEQ.
[3]This symbolic offset is logically synonymous with RAB$V_EQNXT.

(continued on next page)

Table RMS–29 (Cont.) Get Service RAB Input Fields

Field Name	Option or XAB Type	Description
	RAB$V_KGT[4]	Search for next key value according to sort order; if neither the RAB$V_KGE (RAB$V_EQNXT) option nor the RAB$V_KGT (RAB$V_NXT) option is specified, RMS looks for a key match.
	RAB$V_LIM	Limit: specifies that the key value described by the RAB$L_KBF field and the RAB$B_KSZ field is to be compared with the value in the sequentially-accessed record.
	RAB$V_LOC[1]	Locate mode: specifies that Get service record operations use locate mode.
	RAB$V_NLK	No lock: specifies that the record accessed through the Get service is not to be locked.
	RAB$V_NXR	Nonexistent record processing: specifies that if the record directly accessed through a Get service does not exist, the service is to be performed anyway.
	RAB$V_PMT[1]	Prompt: indicates that the contents of the prompt buffer are to be used as a prompt on a Get service to a terminal device.
	RAB$V_PTA[1]	Purge type-ahead: eliminates any information that may be in the type-ahead buffer on a Get service to a terminal device.
	RAB$V_RAH[1]	Read ahead: used with multiple buffers to indicate read-ahead operations (sequential files only).
	RAB$V_REA	Lock for read: allows other users read access to the record.
	RAB$V_REV	Reverses search direction for random keyed access operations when used with either RAB$V_KGE or RAB$V_KGT (for indexed files only).
	RAB$V_RLK	Read of locked record allowed: specifies that a record locked for modification can be read by other users.

[1]This field is not supported for DECnet for OpenVMS operations.

[4]This symbolic offset is logically synonymous with RAB$V_NXT.

(continued on next page)

Table RMS–29 (Cont.) Get Service RAB Input Fields

Field Name	Option or XAB Type	Description
	RAB$V_RNE[1]	Read no echo indicates that input data entered on the keyboard is not displayed on the terminal device.
	RAB$V_RNF[1]	Read no filter: indicates that Ctrl/U, Ctrl/R, and DELETE are not to be considered control commands on terminal input, but are to be passed to the application program.
	RAB$V_RRL	Read regardless of lock: read the record even if another stream has locked the record.
	RAB$V_TMO[1]	Timeout: indicates that the content of the timeout period field (RAB$B_TMO) is to be used.
	RAB$V_ULK	Manual unlocking: specifies that records cannot be unlocked automatically.
	RAB$V_WAT	Wait: if record is locked, wait until it is available.
RAB$B_TMO[1]		Timeout period: indicates the maximum number of seconds that RMS allows between characters for a Get service to terminal and mailbox devices, or the maximum number of seconds RMS waits for a locked record if you specify the RAB$V_TMO and RAB$V_WAT options in the RAB$L_ROP field.
RAB$L_UBF		User record buffer address (required).
RAB$W_USZ		User record buffer size (required).
RAB$L_XAB	XABTRM[1]	Next XAB address: indicates the address of a XABTRM control block (the RAB$L_ROP field RAB$V_ETO option must be set for an extended terminal operation).

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–30 lists the RAB control block fields written as output by the Get service.

Table RMS–30 Get Service RAB Output Fields

Field Name	Description
RAB$L_BKT	Bucket code: set to the relative record number for relative files when the record access mode is sequential.
RAB$L_RBF	Record buffer address.
RAB$W_RFA	Record file address.
RAB$W_RSZ	Record size.
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value (contains a terminator character for terminal input or the record length if the requested record is too large for the user buffer area).

RAB64 Control Block Fields (Alpha Only)

Alpha

Table RMS–31 lists the Alpha-only RAB64 control block fields read as input by the Get service. These fields are comparable to the RAB fields described in Table RMS–29. For additional information on the fields accessed by this service, see Part II.

Table RMS–31 Get Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$B_BLN	This field must be initialized to RAB64$C_BLN64 in order for RAB64 fields to be used.
RAB64$W_ISI	Internal stream identifier (required). Equates to RAB$W_ISI.
RAB64$L_KBF	Key buffer address. This field must contain −1 if you want to use RAB64$PQ_KBF. For 32-bit addressing, this field equates to RAB$L_KBF (see Table RMS–29).
RAB64$PQ_KBF	Key buffer 64-bit address (used if RAB64$L_KBF contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$B_KRF	Key of reference. Equates to RAB$B_KRF (see Table RMS–29).
RAB64$B_KSZ	Key buffer size. Equates to RAB$B_KSZ (see Table RMS–29).
RAB64$L_PBF[1]	Prompt buffer address. Equates to RAB$L_PBF (see Table RMS–29).
RAB64$B_PSZ[1]	Prompt buffer size. Equates to RAB$B_PSZ (see Table RMS–29).
RAB64$B_RAC	Record access mode. Equates to RAB$B_RAC and constants described in Table RMS–29. The constants are identical except for the RAB64 prefix; for example, RAB64$C_KEY equates to RAB$C_KEY.

[1]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–31 (Cont.) Get Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA (see Table RMS–29).
RAB64$L_RHB	Record header buffer. This field must contain −1 if you want to use RAB64$PQ_RHB. For 32-bit addressing, this field equates to RAB$L_RHB (see Table RMS–29).
RAB64$PQ_RHB	Record header buffer 64-bit address (used if RAB64$L_RHB contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$L_ROP	Record-processing options. Equates to RAB$L_ROP and options described in Table RMS–29. Options are identical except for the RAB64 prefix; for example, option RAB64$V_ASY equates to RAB$V_ASY.
RAB64$B_TMO[1]	Timeout period. Equates to RAB$B_TMO (see Table RMS–29).
RAB64$L_UBF[2]	User record buffer address. This field must contain −1 if you want to use RAB64$PQ_UBF. For 32-bit addressing, this field equates to RAB$L_UBF (see Table RMS–29).
RAB64$PQ_UBF[2]	User record buffer 64-bit address (used if RAB64$L_UBF contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$W_USZ[2]	User record buffer size. This field is ignored in favor of RAB64$Q_USZ if RAB64$L_UBF contains −1. Otherwise, this field equates to RAB$W_USZ (see Table RMS–29).
RAB64$Q_USZ[2]	User record buffer size. This field must be used when RAB64$L_UBF contains −1 and a value is specified in RAB64$PQ_UBF.
RAB64$L_XAB	Next XAB address. Equates to RAB$L_XAB and its XABTRM option (see Table RMS–29).

[1]This field is not supported for DECnet for OpenVMS operations.

[2]One of the UBF fields must contain an address and the USZ field associated with it must contain a size.

Table RMS–32 lists the Alpha-only RAB64 control block fields written as output by the Get service. These fields are comparable to the RAB fields described in Table RMS–30.

Table RMS–32 Get Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$L_BKT	Bucket code. Equates to RAB$L_BKT (see Table RMS–30).
RAB64$L_RBF	Record buffer address is returned to this field if RAB64$L_UBF does not contain −1. Equates to RAB$L_RBF.

(continued on next page)

Table RMS–32 (Cont.) Get Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$PQ_RBF	Record buffer address is returned to this field if RAB64$L_UBF contains −1.
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA.
RAB64$W_RSZ	Record buffer size is returned to this field if RAB64$L_UBF does not contain −1. Equates to RAB$W_RSZ.
RAB64$Q_RSZ	Record buffer size is returned to this field if RAB64$L_UBF contains −1.
RAB64$L_STS	Completion status code. Equates to RAB$L_STS (see Table RMS–30).
RAB64$L_STV	Status value. Equates to RAB$L_STV (see Table RMS–30). ♦

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ANI	RMS$_BES
RMS$_BLN	RMS$_BUG	RMS$_BUG_DAP
RMS$_CDA	RMS$_CHK	RMS$_CONTROLC
RMS$_CONTROLY	RMS$_DEADLOCK	RMS$_DEL
RMS$_DME	RMS$_DNR	RMS$_EOF
RMS$_ENQ	RMS$_EXENQLM	RMS$_EXP
RMS$_EXT_ERR	RMS$_FAC	RMS$_FTM
RMS$_IBF	RMS$_IDXSEARCH	RMS$_IOP
RMS$_IRC	RMS$_ISI	RMS$_KBF
RMS$_KEY	RMS$_KRF	RMS$_KSZ
RMS$_MRN	RMS$_NET	RMS$_NETBTS
RMS$_NETFAIL	RMS$_NORMAL	RMS$_OK_ALK
RMS$_OK_DEL	RMS$_OK_LIM	RMS$_OK_RLK
RMS$_OK_RNF	RMS$_OK_RRL	RMS$_OK_WAT
RMS$_OPNOTSUP	RMS$_PBF	RMS$_PENDING
RMS$_PES	RMS$_PLG	RMS$_RAB
RMS$_RAC	RMS$_RER	RMS$_RFA
RMS$_RHB	RMS$_RLK	RMS$_RNF
RMS$_ROP	RMS$_RPL	RMS$_RRV
RMS$_RSA	RMS$_RTB	RMS$_SQO
RMS$_STR	RMS$_SUC	RMS$_SUP

RMS$_SUPPORT	RMS$_SYS	RMS$_TMO
RMS$_TNS	RMS$_TRE	RMS$_UBF
RMS$_WBE	RMS$_WER	RMS$_WLK
RMS$_XAB		

$NXTVOL

The Next Volume service allows you to process the next tape volume in a multiple volume set. This service applies only to files on magnetic tape volumes.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Next Volume service on OpenVMS Alpha systems. ♦

Format

SYS$NXTVOL rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Next Volume service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

You use the Next Volume service when you want to proceed to the next volume in the set before the end of the current volume (EOV label) is reached on input, or before the end-of-tape (EOT) mark is reached on output. RMS positions your process to the first file section on the next volume. File sections occur when a file is written on more than one volume, the portion of the file on each of the volumes constituting a file section.

When you perform a Next Volume service for input files, RMS responds as follows:

- If the current volume is the last volume of the set, RMS reports end-of-file information.
- If another file section exists, the next volume is mounted. When necessary, the current volume is rewound and a request to mount the next volume is issued to the operator.
- The header label (HDR1) of the file section on the newly mounted volume is read. If this is not the volume being sought, the operator is requested to mount the correct volume.

When you perform a Next Volume service for output files, the following sequence occurs:

1. The file section on the current volume is closed with the appropriate end-of-volume labels, and the volume is rewound.
2. The next volume is mounted.
3. A file with the same file name and the next higher file section number is opened for output, and processing continues.

If your program is operating asynchronously, it must wait for any I/O activity on this volume to complete before issuing a Next Volume service.

The Next Volume service performs a Flush service for write-accessed volumes (see $FLUSH), thus writing the I/O buffers on the current volume before creating the next file section. If this is an input-only file, then all records currently contained in the I/O buffers are lost, and the next Get service returns the first record on the next volume.

The Next Volume service is not supported for DECnet for OpenVMS operations on files between two OpenVMS systems.

Table RMS–33 lists the control block fields read as input and written as output by the Next Volume service. For additional information on the fields accessed by this service, see Part II.

Table RMS–33 Next Volume Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Next Volume service asynchronously.

Table RMS–34 lists the control block fields written as output by the Next Volume service.

Table RMS–34 Delete Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_CDA
RMS$_DME	RMS$_DNR	RMS$_DPE
RMS$_EXT_ERR	RMS$_IOP	RMS$_ISI
RMS$_NORMAL	RMS$_OPNOTSUP	RMS$_PENDING
RMS$_RAB	RMS$_RSA	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS		

$OPEN

The Open service makes an existing file available for processing by your program. The Open service specifies the type of record access to be used and determines whether the file can be shared. The Open service also performs an implicit Display service.

Format

SYS$OPEN fab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage:	fab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

FAB control block whose contents are to be used as indirect arguments for the Open service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

You must open a file to perform any record operations and most file operations. If any XABs are chained to the FAB, RMS places the attribute values in the fields of the appropriate XAB. If you specify a NAM block in the FAB, the contents of the device, directory, and file identification fields can be used with the open-by-NAM-block option to open the file. The NAM block fields are filled in with auxiliary file specification information.

Table RMS–35 lists the control block fields read as input by the Open service. For additional information on the fields accessed by this service, see Part II.

Table RMS–35 Open Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
FAB$B_ACMODES		File access modes.
	FAB$V_CHAN_MODE[3]	Assigns the channel access mode by setting either the FAB$V_UFO or the FAB$V_NFS bit in the FAB$L_FOP field (see Section 4.17). If neither bit is set, this field can be used to override the access mode protection for a specified I/O operation. See Section 4.8.
	FAB$V_LNM_MODE[3]	Specifies the logical name translation access mode.
FAB$W_DEQ		Default file extension quantity: if a nonzero value is present in this field, it applies only to this open of the file.
FAB$L_DNA		Default file specification string address.
FAB$B_DNS		Default file specification string size.
FAB$B_FAC		File access field.
	FAB$V_BIO	Block I/O access.
	FAB$V_BRO	Block or record I/O.
	FAB$V_DEL	Delete access.
	FAB$V_GET[1]	Read access.
	FAB$V_PUT	Write access.
	FAB$V_TRN	Truncate access.
	FAB$V_UPD	Update access.
FAB$L_FNA[2]		File specification string address.

[1]This is the default value supplied by RMS.

[2]These fields must be specified by the user.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–35 (Cont.) Open Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
FAB$B_FNS[2]		File specification string size.
FAB$L_FOP		File-processing options.
	FAB$V_DFW[3]	Deferred write: indicates that writing back to the file from the modified buffer is deferred (applies to relative and indexed files only).
	FAB$V_DLT	Delete: indicates the file is to be deleted when it is closed.
	FAB$V_NAM[3]	Name block inputs: indicates that the NAMW_FID, NAMW_DID, and NAM$T_DVI fields in the specified NAM block are to be used to describe the file.
	FAB$V_NFS[3]	Non-file-structured: indicates that the accessed volume is to be processed in a non-file-structured manner.
	FAB$V_OFP	Output file parse: specifies that the related file resultant file specification string, if used, is to provide file name and file type defaults only.
	FAB$V_RCK	Read-check: indicates that transfers from disk are to be checked by a followup, read-compare operation.
	FAB$V_RWC	Rewind on close (applies to magnetic tapes only).
	FAB$V_RWO	Rewind on open (applies to magnetic tapes only).
	FAB$V_SCF[3]	Submit command file: indicates that the file is to be submitted as a batch-command file to the process default batch queue (SYS$BATCH) when the file is closed (applies to sequential files only).

[2]These fields must be specified by the user.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–35 (Cont.) Open Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
	FAB$V_SPL[3]	Spool: indicates that the file is to be spooled to the process default print queue (SYS$PRINT) when the file is closed (applies to sequential files only).
	FAB$V_SQO	Sequential only: indicates that the file can be processed in a sequential manner only.
	FAB$V_TEF	Truncate at end of file: indicates that unused space allocated to a file is to be deallocated when that file is closed (applies to sequential files only).
	FAB$V_UFO[3]	User file open: indicates the file is to be opened only (no further processing of that file is allowed).
	FAB$V_WCK	Write-check: indicates that transfers to disk are to be followed by a read-compare operation.
FAB$B_FSZ		Fixed control area size: unit record devices only.
FAB$W_IFI		Internal file identifier (must be 0).
FAB$L_NAM		Name block address.
FAB$B_RAT		Record attributes; only for process permanent files with print file format.
FAB$B_RFM		Record format; unit record devices only.
FAB$B_RTV[3]		Retrieval window size.
FAB$B_SHR		File-sharing field.
	FAB$V_SHRDEL	Shared delete access.
	FAB$V_SHRGET	Shared read access.
	FAB$V_MSE[3]	Multistream access; also used with the FAB$V_MSE and FAB$V_GET bits to specify a read-only global buffer cache when global buffering is enabled.
	FAB$V_NIL	No shared access.
	FAB$V_SHRPUT	Shared write access.

[3]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–35 (Cont.) Open Service FAB and XAB Input Fields

Field Name	Option or XAB Type	Description
	FAB$V_SHRUPD	Shared update access.
	FAB$V_UPI	Shared write access for block I/O (applies to sequential files only).
FAB$L_XAB[4]		Extended attribute block address.
	XABITM	Item list XAB; see Chapter 11.

[4]The appropriate XAB must be specified as input if you desire information about that particular XAB on output from the Open service.

Table RMS–36 lists the control block fields written as output by the Open service.

Table RMS–36 Open Service FAB and XAB Output Fields

Field Name	Option or XAB Type	Description
FAB$L_ALQ		Allocation quantity: contains the highest numbered block allocated to the file.
FAB$B_BKS		Bucket size (does not apply to sequential files).
FAB$W_BLS		Device block size (applies only to sequential files).
FAB$W_DEQ		Default file extension quantity.
FAB$L_DEV		Device characteristics.
FAB$B_FAC		File access.
FAB$L_FOP		File-processing options.
	FAB$V_ASY	Asynchronous; indicates that the specified task is to be done asynchronously.
	FAB$V_CBT	Contiguous best try: indicates that the file is allocated contiguously on a ‘‘best effort’’ basis.
	FAB$V_CTG	Contiguous: indicates that space for the file is allocated contiguously.
	FAB$V_RCK	Read-check: transfers are followed up by a read-compare operation.
	FAB$V_WCK	Write-check: transfers are followed up by a read-compare operation.
FAB$B_FSZ		Fixed-length control area size (applies only to VFC records).
FAB$W_GBC		Global buffer count.
FAB$W_IFI		Internal file identifier.

(continued on next page)

Table RMS–36 (Cont.) Open Service FAB and XAB Output Fields

Field Name	Option or XAB Type	Description
FAB$L_MRN		Maximum record number (for relative files only).
FAB$W_MRS		Maximum record size.
FAB$B_ORG		File organization.
FAB$B_RAT		Record attributes; used as output field except for process-permanent files with print file format.
FAB$B_RFM		Record format.
FAB$L_SDC		Spooling device characteristics.
FAB$B_SHR		File sharing.
FAB$L_STS		Completion status code (also returned in register 0).
FAB$L_STV		Status value (contains the I/O channel number if the operation is successful).
FAB$L_XAB		Next XAB address.
	XABALL	Allocation XAB; see Chapter 8.
	XABDAT	Date and time XAB; see Chapter 9.
	XABFHC	File header characteristics XAB; see Chapter 10.
	XABITM	Item list XAB; see Chapter 11.
	XABKEY	Key definition XAB; see Chapter 13.
	XABPRO	Protection XAB; see Chapter 14.
	XABRDT	Revision date and time XAB; see Chapter 15.
	XABSUM	Summary XAB; see Chapter 17.

Using the Name Block for Opening Files

Table RMS–37 and Table RMS–38 list the NAM block fields (further described in Chapter 5) used as input and output for the Open service (provided that the NAM block address field is specified in the FAB).

Table RMS–37 Open Service NAM Block Input Fields

Field Name	Option	Description
NAM$W_DID[1]		Directory identification (input only if the FAB$L_FOP field FAB$V_NAM option is set).
NAM$T_DVI[1]		Device identification (input only if the FAB$L_FOP field FAB$V_NAM option is set).

[1]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–37 (Cont.) Open Service NAM Block Input Fields

Field Name	Option	Description
NAM$L_ESA		Expanded string area address.
NAM$B_ESS		Expanded string area size.
NAM$W_FID[1]		File identification (input only if the FAB$L_FOP field FAB$V_NAM option is set).
NAM$B_NOP		NAM block options.
	NAM$V_PWD	Password: indicates that a password contained in a DECnet for OpenVMS access control string, if present in a file specification, is to be left unaltered in the expanded and resultant strings (instead of being replaced by the word ''password'').
	NAM$V_NOCONCEAL	Do not conceal device name: indicates that when a concealed device logical name is present, the concealed device logical name is to be replaced by the actual physical device name in the resultant string.
NAM$L_RLF		Related file NAM block address.
NAM$B_RSL		Related file NAM block resultant string length.
NAM$L_RSA		Related file NAM block resultant string address.
NAM$L_FNB		Related file NAM block file name status bits.
NAM$B_RSS		Resultant string area size.

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–38 Open Service NAM Block Output Fields

Field Name	Description
NAM$W_DID[1]	Directory identification.
NAM$T_DVI[1]	Device identification.
NAM$B_ESL	Expanded string length. If the NAM$L_ESA and NAM$B_ESS fields are nonzero, and if the FAB$L_FOP field FAB$V_NAM option is clear or the NAM$W_DID and NAM$W_FID fields are 0 on input, the expanded file specification string is copied to the buffer specified by the NAM$L_ESA field.

[1]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–38 (Cont.) Open Service NAM Block Output Fields

Field Name	Description
NAM$W_FID[1]	File identification.
NAM$L_FNB	File name status bits.
NAM$B_RSL	Resultant string length. If the NAM$L_RSA field and the NAM$B_RSS field are nonzero, and if the FAB$V_NAM bit is clear or the NAM$W_FID field is zero when you invoke the Open service, the resultant file specification is copied to the buffer specified by the NAM$L_RSA field.
NAM$L_DEV	Address of file specification device string.
NAM$B_DEV	Size of file specification device string.
NAM$L_DIR	Address of file specification directory string.
NAM$B_DIR	Size of file specification directory string.
NAM$L_NAME	Address of file specification name string.
NAM$B_NAME	Size of file specification name string.
NAM$L_NODE	Address of file specification node string.
NAM$B_NODE	Size of file specification node string.
NAM$L_TYPE	Address of file specification type string.
NAM$B_TYPE	Size of file specification type string.
NAM$L_VER	Address of file specification version string.
NAM$B_VER	Size of file specification version string.

[1]This field is not supported for DECnet for OpenVMS operations.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACC	RMS$_ACS	RMS$_ACT
RMS$_AID	RMS$_ATR	RMS$_BLN
RMS$_BUG_DAP	RMS$_BUG_DDI	RMS$_CHN
RMS$_COD	RMS$_CRMP	RMS$_DEV
RMS$_DIR	RMS$_DME	RMS$_DNA
RMS$_DNF	RMS$_DNR	RMS$_DVI
RMS$_ENQ	RMS$_ENV	RMS$_ESA
RMS$_ESS	RMS$_EXP	RMS$_EXTNOTFOU
RMS$_FAB	RMS$_FLK	RMS$_FNA
RMS$_FNF	RMS$_FNM	RMS$_FOP
RMS$_IFA	RMS$_IFI	RMS$_IMX
RMS$_IRC	RMS$_KNM	RMS$_KSI
RMS$_LNE	RMS$_NAM	RMS$_NET
RMS$_NETFAIL	RMS$_NOD	RMS$_NORMAL

RMS$_OK_NOP	RMS$_ORG	RMS$_PLG
RMS$_PLV	RMS$_PRV	RMS$_QUO
RMS$_RAT	RMS$_REF	RMS$_RLF
RMS$_RPL	RMS$_RSS	RMS$_RST
RMS$_RUNDOWN	RMS$_SHR	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPERSEDE
RMS$_SUPPORT	RMS$_SYN	RMS$_SYS
RMS$_TYP	RMS$_UPI	RMS$_VER
RMS$_WLK	RMS$_XAB	

$PARSE

The Parse service analyzes the file specification string and fills in various NAM block fields.

Format

SYS$PARSE fab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage:	fab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

FAB control block whose contents are to be used as indirect arguments for the Parse service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The functions of the Parse service are performed automatically as part of the Open, Create, and Erase services. One special purpose of the Parse service is to prepare the FAB and NAM blocks for wildcard character processing to be used in the Search service. If wildcard characters, search list logical names, or a node name are present in the file specification, RMS allocates internal data structures (including a device channel) to store the context for subsequent searches. This space is released when the Search service encounters a no-more-files condition (in which case an RMS$_NMF error status is returned) or when another Parse service is performed using the same FAB and NAM blocks. To release this space, use a Parse service that specifies the NAM$B_NOP field NAM$V_SYNCHK option and sets the FAB$B_DNS and NAM$L_RLF fields to zero.

Note that the file must be closed before you invoke the Parse service (FAB$W_IFI must be 0).

By default, the Parse service assigns a channel to the device and does a lookup of the directory in addition to analyzing the file specification and filling in the NAM block fields. To request a Parse service without I/O, specify the NAM$B_NOP field NAM$V_SYNCHK option. The result of a Parse service without I/O cannot be used as input to subsequent Search services.

Table RMS–39 and Table RMS–40 list the fields in both the FAB and NAM block that the Parse service uses as input and output. In addition, RMS fills in the string component descriptors from the expanded string (see Chapter 5).

The expanded file specification string is moved to the buffer described by the expanded string area address (NAM$L_ESA) and size (NAM$B_ESS) fields of the NAM block (only if both fields are nonzero). The NAM$L_ESA and NAM$B_ESS fields must be specified (nonzero) for wildcard character processing.

Table RMS–39 lists the control block fields read as input by the Parse service. For additional information on the fields accessed by this service, see Part II.

Table RMS–39 Parse Service FAB and NAM Block Input Fields

Field Name	Option	Description
FAB$B_ ACMODES		File access modes.
	FAB$V_CHAN_ MODE[1]	This field can be used to override the access mode protection for a specified I/O operation (see Section 4.8).
FAB$L_DNA		Default file specification string.
FAB$B_DNS		Default file specification string size.
FAB$L_FNA		File specification string address.
FAB$B_FNS		File specification string size.

[1]Not supported for DECnet for OpenVMS operations

(continued on next page)

Table RMS–39 (Cont.) Parse Service FAB and NAM Block Input Fields

Field Name	Option	Description
FAB$L_FOP		File-processing option, FAB$V_OFP. Output file parse: indicates that RMS uses only the file name and file type fields of a related file resultant string whose address is contained in the NAM$L_RSA field.
FAB$W_IFI		Internal file identifier (must be zero).
FAB$L_NAM		Name block address.
NAM$L_ESA		Expanded string area address.
NAM$B_ESS		Expanded string area size.
NAM$B_NOP		NAM block options.
	NAM$V_ NOCONCEAL	Do not conceal device name: indicates that when a concealed device logical name is present, the concealed device logical name is to be replaced by the actual physical device name in the expanded string.
	NAM$V_PWD	Password: indicates that a password contained in a DECnet for OpenVMS access control string, if present in a file specification, is to be left unaltered in the expanded and resultant strings (instead of being replaced by the word ''password'').
	NAM$V_SYNCHK	Performs Parse service with no I/O.
NAM$L_RLF		Related file NAM block address.
NAM$L_RSA		Related file NAM block resultant string area address.
NAM$B_RSL		Related file NAM block resultant string length.
NAM$L_FNB		Related file NAM block file name status bits.

Table RMS–40 lists the control block fields written as output by the Parse service.

Table RMS–40 Parse Service FAB and NAM Block Output Fields

Field Name	Description
FAB$L_DEV	Device characteristics (unless you select the NAM$V_SYNCHK option).
FAB$L_SDC	Secondary device characteristics (unless you select the NAM$V_SYNCHK option).
FAB$L_STS	Completion status code (also returned in register 0).
FAB$L_STV	Status value.
NAM$L_DEV	Address of file specification device string.
NAM$B_DEV	Size of file specification device string.
NAM$W_DID[1]	Directory identification (unless you select the NAM$V_SYNCHK option).
NAM$L_DIR	Address of file specification directory string.
NAM$B_DIR	Size of file specification directory string.
NAM$T_DVI[1]	Device identification (unless you select the NAM$V_SYNCHK option).
NAM$B_ESL	Expanded string length.
NAM$W_FID[1]	File identification (zeroed).
NAM$L_FNB	File name status bits: contains information about the parse results.
NAM$L_NAME	Address of file specification name string.
NAM$B_NAME	Size of file specification name string.
NAM$L_NODE	Address of file specification node string.
NAM$B_NODE	Size of file specification node string.
NAM$B_RSL	Resultant string length (zeroed).
NAM$L_TYPE	Address of file specification type string.
NAM$B_TYPE	Size of file specification type string.
NAM$L_VER	Address of file specification version string.
NAM$B_VER	Size of file specification version string.
NAM$L_WCC	Wildcard context.

[1]This field is not supported for DECnet for OpenVMS operations.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACS	RMS$_BLN	RMS$_BUG_DDI
RMS$_CDA	RMS$_CHN	RMS$_DEV
RMS$_DIR	RMS$_DME	RMS$_DNA
RMS$_DNF	RMS$_DNR	RMS$_ESA
RMS$_ESS	RMS$_FAB	RMS$_FNA
RMS$_FNM	RMS$_IFI	RMS$_LNE

RMS$_NAM	RMS$_NOD	RMS$_NORMAL
RMS$_QUO	RMS$_RLF	RMS$_RUNDOWN
RMS$_STR	RMS$_SUC	RMS$_SYN
RMS$_TYP	RMS$_VER	RMS$_WCC

$PUT

The Put service inserts a record into a file.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Put service on OpenVMS Alpha systems. ♦

Format

SYS$PUT rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Put service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Put service usually adds records to the logical end of a sequential file. For relative files, it may add records to the logical end of the file or it may insert new records in cells formerly occupied by deleted records. RMS directs the Put service where to insert the record using the contents of the record's primary key field.

Inserting Records into Sequential Files

When using sequential record access mode to process sequential files, you usually insert records at the end of the file only. The records to be inserted cannot be larger than the maximum length that was specified when the file was created.

You can use random access by relative record number mode and the update-if record-processing option (RAB$V_UIF) to insert fixed-length records into a sequential file residing on a disk device.

RMS also provides for establishing the logical end of the file when two or more processes are doing shared write operations. For example, assume that processes A and B are sharing a sequential file and each process is putting data into the file. Process A puts a record at the end of the file and intends to put another record at the new end-of-file location. However, before process A can put the next record in the file, process B gains access to the file and puts a record at the end of the file. In order to ensure that the next record from process A does not overwrite the record just inserted by process B, RMS updates process A's write pointer to the new end-of-file position; that is, the location immediately following the location of process B's record.

The truncate-on-put option (RAB$V_TPT) can be used with sequential files. This option lets you add records at locations other than the logical end of the file. When you add a record using the truncate-on-put option, the file is automatically truncated, effectively deleting all data between the new record (logical end of the file) and the physical end of the file. If you try to use this option without having truncate access, RMS rejects the operation and issues a file access error (RMS$_FAC).

For stream format files, RMS writes the contents of the user's buffer into the file beginning at the current entry position. If the last byte in the buffer is not a terminator, RMS adds the appropriate terminator. For stream format, the terminator is CRLF (carriage return character followed immediately by a line feed character).

Mailboxes may be used to synchronize activity between processes. Usually, a Put service to a mailbox does not conclude until another accessor reads the record. If you select the timeout option (RAB$V_TMO) and specify a timeout period of 0, the Put service does not wait for another accessor to read the record.

At the conclusion of the Put service, the RAB$L_STV field contains the process identification (PID) of the process that read the record.

Inserting Records into Relative Files

When processing relative files, you can use either sequential or random access by key mode. Records cannot be larger than the size specified at file creation time, and the record's relative record number must not exceed the maximum record number established for the file. Usually, if the target record cell for a Put service contains a record, a record-already-exists error (RMS$_REX) is returned as the completion status (RAB$L_STS). If you specify the update-if (RAB$V_UIF) record option, RMS overwrites the existing record instead of returning an error message.

If you try to use the update-if option but do not have update access, RMS rejects the operation and issues a file access error (RMS$_FAC).

Inserting Records into Indexed Files

In an indexed file, you can use sequential access or random access by key mode. When sequential access is used to insert records, the primary key value of the record to be inserted must be consistent with the specified sort order of the file. That is, the key must be greater than or equal to the primary value of the previous record if ascending sort order is specified. If descending sort order is specified, the key must be less than or equal to the primary key value of the previous record.

The records cannot be larger than the size established when the file was created if a maximum length was specified. Each record written must contain a primary key, but the records do not have to contain alternate keys. If alternate keys are partially or completely missing because of the record length limitation, RMS does not make an entry for the record in the associated alternate index. Put services to an indexed file do not require a separate key value or key of reference. By examining the contents of the primary key in the record, RMS determines where to insert the record.

When inserting a record into an indexed file, RMS compares the key values in the record with the key values of records previously inserted into the file to determine whether the new record's key value duplicates any existing key values. If the record duplicates a key value in an index where duplication is not allowed, RMS rejects the operation with an RMS$_DUP error code. Where duplicate keys are allowed, RMS inserts the record.

Records with duplicate keys are inserted in chronological order; that is, RMS inserts each record having duplicate keys at the end of a ''chain'' of identically keyed records so that newer records are stored closer to the end of the file regardless of sort order.

If you specify the update-if (RAB$V_UIF) option when duplicates are not allowed on the primary key, RMS overwrites the existing record with the same primary key value, rather than returning a duplicate record error (RMS$_DUP). This gives the appearance of an Update service being performed on the existing record. Alternate key values are modified to reflect the newly inserted record.

To use the RAB$V_UIF option, you must have update access to the file. If update access to the file is not permitted, the Put service (which becomes an Update service when this option is selected) fails, and RMS returns a file access error (RMS$_FAC).

Be careful when invoking the Put service with the RAB$V_UIF option and automatic record locking for a shared file. The Put service, unlike the Update service, momentarily releases record locks previously applied by a Get or Find service, until the Put service is converted into an Update service. This could allow another record stream to delete or update the record between the invocation of the Put service and the conversion to an Update service. To avoid this complication, you should use the Update service instead of the Put service with the update-if option to update an existing record in a file-sharing situation.

The record address field and the record size field are required inputs to the Put service. Some Put service options may require additional fields. The traditional address and size fields are RAB$L_RBF and RAB$W_RSZ. However, OpenVMS Alpha users have the option to code −1 in the RAB64$L_RBF field to direct the Put service to use the values in the alternative fields, RAB64$PQ_RBF and

RAB64$Q_RSZ. The RAB64$PQ_RBF field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

A successful Put service returns the record file address (RFA) in the RAB$W_RFA field.

RAB Control Block Fields

Table RMS–41 lists the control block fields read as input by the Put service. For additional information on the fields accessed by this service, see Part II.

Table RMS–41 Put Service RAB Input Fields

Field Name	Option or XAB Type	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_KBF		Key buffer address (used as input only with random access by relative record number mode).
RAB$B_KSZ		Key size (used only if RAB$B_RAC is KEY and the file is a relative file).
RAB$B_RAC		Record access mode (SEQ, KEY)[1].
RAB$L_RBF		Record buffer address.
RAB$L_RHB		Record header buffer (for variable with fixed control records only).
RAB$W_RSZ		Record size.
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Put services asynchronously.
	RAB$V_CCO[2]	Cancel Ctrl/O: guarantees that terminal output is not discarded if the operator enters Ctrl/O.
	RAB$V_LOA	Load: specifies that buckets are to be loaded according to the fill size established at file creation time.
	RAB$V_REA[3]	Lock for read: allows other users read access to the record. This is not valid for relative files.
	RAB$V_RLK[3]	Read of locked record allowed: specifies that a record locked for modification can be read by other users.
	RAB$V_TMO[2]	Timeout: indicates that the content of the timeout period field (RAB$B_TMO) is to be used.

[1]The default for the RAB$B_RAC field is RAB$C_SEQ.

[2]This field is not supported for DECnet for OpenVMS operations.

[3]This option is meaningless unless you specify manual unlocking.

(continued on next page)

Table RMS–41 (Cont.) Put Service RAB Input Fields

Field Name	Option or XAB Type	Description
	RAB$V_TPT	Truncate-on-put: specifies that a Put service with a sequentially-accessed record can occur at any point in the file, truncating the file at that point.
	RAB$V_UIF	Update-if: converts a Put service to a record that already exists to an Update service.
	RAB$V_ULK	Manual unlocking: specifies that records cannot be unlocked automatically.
	RAB$V_WBH	Write behind: two buffers are allocated to allow multibuffering.
	RAB$V_WAT	Wait: if the record is locked, wait until it is available (for relative files only).
RAB$B_TMO[2]		Timeout period: a value of 0 indicates that RMS should not wait to complete a Put service (for mailbox devices only).

[2]This field is not supported for DECnet for OpenVMS operations.

Table RMS–42 lists the control block fields written as output by the Put service.

Table RMS–42 Put Service RAB Output Fields

Field Name	Description
RAB$L_BKT	Bucket code: set to the relative record number for sequential access to relative files.
RAB$W_RFA	Record file address.
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value[1].

[1]On the successful completion of a Put service to a record-oriented device, the RAB$L_STV field contains the second longword of the I/O status block. See the *OpenVMS I/O User's Reference Manual* for details on specific devices.

RAB64 Control Block Fields (Alpha Only)

Alpha

Table RMS–43 lists the Alpha-only RAB64 control block fields read as input by the Put service. These fields are comparable to the RAB fields described in Table RMS–41. For additional information on the fields accessed by this service, see Part II.

Table RMS–43 Put Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$B_BLN	This field must be initialized to RAB64$C_BLN64 in order for RAB64 fields to be used.
RAB64$W_ISI	Internal stream identifier (required). Equates to RAB$W_ISI.
RAB64$L_KBF	Key buffer address. This field must contain −1 if you want to use RAB64$PQ_KBF. For 32-bit addressing, this field equates to RAB$L_KBF (see Table RMS–41).
RAB64$PQ_KBF	Key buffer 64-bit address (used if RAB64$L_KBF contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$B_KSZ	Key buffer size. Equates to RAB$B_KSZ (see Table RMS–41).
RAB64$B_RAC	Record access mode. Equates to RAB$B_RAC (see Table RMS–41).
RAB64$L_RBF[1]	Record buffer address. This field must contain −1 if you want to use RAB64$PQ_RBF. For 32-bit addressing, this field equates to RAB$L_RBF.
RAB64$PQ_RBF[1]	Record buffer 64-bit address (used if RAB64$L_RBF contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$L_RHB	Record header buffer. This field must contain −1 if you want to use RAB64$PQ_RHB. For 32-bit addressing, this field equates to RAB$L_RHB (see Table RMS–41).
RAB64$PQ_RHB	Record header buffer 64-bit address (used if RAB64$L_RHB contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$W_RSZ[1]	Record buffer size. This field is ignored in favor of RAB64$Q_RSZ if RAB64$L_RBF contains −1. Otherwise, this field equates to RAB$W_RSZ.
RAB64$Q_RSZ[1]	Record buffer size. This field must be used when RAB64$L_RBF contains −1 and a value is specified in RAB64$PQ_RBF.
RAB64$L_ROP	Record-processing options. Equates to RAB$L_ROP and options described in Table RMS–41. Options are identical except for the RAB64 prefix; for example, option RAB64$V_ASY equates to RAB$V_ASY.
RAB64$B_TMO[2]	Timeout period. Equates to RAB$B_TMO (see Table RMS–41).

[1]One of the RBF fields must contain an address and the RSZ field associated with it must contain a size.

[2]This field is not supported for DECnet for OpenVMS operations.

Table RMS–44 lists the Alpha-only RAB64 control block fields written as output by the Put service. These fields are comparable to the RAB fields described in Table RMS–42.

Table RMS–44 Put Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$L_BKT	Bucket code. Equates to RAB$L_BKT (see Table RMS–42).
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA.
RAB64$L_STS	Completion status code. Equates to RAB$L_STS (see Table RMS–42).
RAB64$L_STV	Status value. Equates to RAB$L_STV (see Table RMS–42). ♦

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BUG
RMS$_BUG_DAP	RMS$_CDA	RMS$_CHK
RMS$_CONTROLC	RMS$_CONTROLO	RMS$_CONTROLY
RMS$_DME	RMS$_DNR	RMS$_DUP
RMS$_ENQ	RMS$_EXT	RMS$_EXT_ERR
RMS$_FAC	RMS$_FTM	RMS$_FUL
RMS$_IBF	RMS$_IDX	RMS$_IOP
RMS$_IRC	RMS$_ISI	RMS$_KBF
RMS$_KEY	RMS$_KSZ	RMS$_MRN
RMS$_NEF	RMS$_NET	RMS$NETBTS
RMS$_NETFAIL	RMS$_NORMAL	RMS$_OK_ALK
RMS$_OK_DUP	RMS$_OK_IDX	RMS$_OPNOTSUP
RMS$_PENDING	RMS$_PLG	RMS$_RAB
RMS$_RAC	RMS$_RBF	RMS$_RER
RMS$_REX	RMS$_RHB	RMS$_RLK
RMS$_RPL	RMS$_RRV	RMS$_RSA
RMS$_RSZ	RMS$_RVU	RMS$_SEQ
RMS$_SQO	RMS$_STR	RMS$_SUC
RMS$_SUP	RMS$_SUPPORT	RMS$_SYS
RMS$_TRE	RMS$_WBE	RMS$_WER
RMS$_WLK	RMS$_WPL	

$READ

The Read service retrieves a specified number of bytes from a file (beginning on a block boundary) and transfers them to memory. A Read service using block I/O can be performed on any file organization.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Read service on OpenVMS Alpha systems. ♦

Format

SYS$READ rab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Read service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

To use the Read service, you must do the following:

1. Supply a buffer area for transferring data (user record area address field) and specify the buffer size:
   - To supply a 32-bit buffer address and a buffer size no greater than 65,535 bytes, use these fields:

User Buffer Address Field	User Buffer Size Field
RAB$L_UBF	RAB$W_USZ

Alpha

   - On OpenVMS Alpha systems, you can supply a 64-bit buffer address (or a 32-bit address sign-extended to 64 bits) and a buffer size up to $2**31-1$ bytes. To do so, code -1 in RAB64$L_UBF and use these fields:

User Buffer Address Field	User Buffer Size Field
RAB64$PQ_UBF	RAB64$Q_USZ ♦

2. Indicate the first virtual block number (VBN) for the transfer (bucket number field). This field is RAB$L_BKT or RAB64$L_BKT (available only on Alpha to accommodate 64-bit addressing). If the value for the VBN is 0, the transfer starts with the block indicated by the next block pointer (NBP).

RAB Control Block Fields

Table RMS–45 lists the control block fields read as input by the Read service. For additional information on the fields accessed by this service, see Part II.

Table RMS–45 Read Service RAB Input Fields

Field Name	Option	Description
RAB$L_BKT		Bucket number: must contain the virtual block number of the first block to read. When this field has a value of 0, then the next block is read.
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing option.
	RAB$V_ASY	Asynchronous: performs Delete service asynchronously.
RAB$L_UBF		User record buffer address. For block I/O, alignment of the user's record buffer on a page or at least a quadword boundary may improve performance.
RAB$W_USZ		User record area size: indicates the length of the transfer, in bytes[1].

[1]Certain devices require that an even number of bytes be transferred. For further details, see the *OpenVMS I/O User's Reference Manual*.

Table RMS–46 lists the control block fields written as output by the Read service.

Table RMS–46 Read Service RAB Output Fields

Field Name	Description
RAB$L_RBF	Record address.
RAB$W_RFA	Record file address.
RAB$W_RSZ	Record size: indicates the actual number of bytes transferred.
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

RAB64 Control Block Fields (Alpha Only)

Alpha

Table RMS–47 lists the Alpha-only RAB64 control block fields read as input by the Read service. These fields are comparable to the RAB fields described in Table RMS–45. For additional information on the fields accessed by this service, see Part II.

Table RMS–47 Read Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$B_BLN	This field must be initialized to RAB64$C_BLN64 in order for RAB64 fields to be used.
RAB64$L_BKT	Bucket number. Equates to RAB$L_BKT (see Table RMS–45).
RAB64$W_ISI	Internal stream identifier (required). Equates to RAB$W_ISI.
RAB64$L_ROP	Record-processing option. Equates to RAB$L_ROP (see Table RMS–45). The RAB64$V_ASY option is identical to RAB$V_ASY.
RAB64$L_UBF[1]	User record buffer address. This field must contain −1 if you want to use RAB64$PQ_UBF. For 32-bit addressing, this field equates to RAB$L_UBF (see Table RMS–45).
RAB64$PQ_UBF[1]	User record buffer 64-bit address (used if RAB64$L_UBF contains −1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$W_USZ[1]	User record buffer size. This field is ignored in favor of RAB64$Q_USZ if RAB64$L_UBF contains −1. Otherwise, this field equates to RAB$W_USZ (see Table RMS–45).
RAB64$Q_USZ[1]	User record buffer size. This field must be used when RAB64$L_UBF contains −1 and a value is specified in RAB64$PQ_UBF. (See Section 7.8 for more information.)

[1]One of the UBF fields must contain an address and the USZ field associated with it must contain a size.

Table RMS–48 lists the Alpha-only RAB64 control block fields written as output by the Read service. These fields are comparable to the RAB fields described in Table RMS–46.

Table RMS–48 Read Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$L_RBF	Record buffer address is returned to this field if RAB64$L_UBF does not contain −1. Equates to RAB$L_RBF.
RAB64$PQ_RBF	Record buffer address is returned to this field if RAB64$L_UBF contains −1.
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA.
RAB64$W_RSZ	Record buffer size is returned to this field if RAB64$L_UBF does not contain −1. Equates to RAB$W_RSZ (see Table RMS–46).
RAB64$Q_RSZ	Record buffer size is returned to this field if RAB64$L_UBF contains −1.
RAB64$L_STS	Completion status code. Equates to RAB$L_STS (see Table RMS–46).
RAB64$L_STV	Status value. Equates to RAB$L_STV. ♦

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BUG_DAP
RMS$_CDA	RMS$_CONTROLC	RMS$_CONTROLY
RMS$_DME	RMS$_DNR	RMS$_EOF
RMS$_FAC	RMS$_FTM	RMS$_IOP
RMS$_ISI	RMS$_NET	RMS$_NETFAIL
RMS$_NORMAL	RMS$_PBF	RMS$_PENDING
RMS$_RAB	RMS$_RER	RMS$_RSA
RMS$_STR	RMS$_SUC	RMS$_SUP
RMS$_SUPPORT	RMS$_SYS	RMS$_TMO
RMS$_UBF	RMS$_USZ	RMS$_WBE

$RELEASE

The Release service unlocks the record specified by the contents of the record file address (RAB$W_RFA) field of the RAB.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Release service on OpenVMS Alpha systems. ♦

Format

SYS$RELEASE rab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Release service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Release service unlocks a specified record (see also the discussion of the Free service). If the named record is not locked, RMS returns a status code of RMS$_RNL.

Table RMS–49 lists the control block fields used as input by the Release service. For additional information on the fields accessed by this service, see Part II.

Table RMS–49 Release Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Release service asynchronously.

Table RMS–50 lists the control block fields written as output by the Release service.

Table RMS–50 Release Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Additional status information.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BUG_DAP
RMS$_CDA	RMS$_EXT_ERR	RMS$_ISI
RMS$_NET	RMS$_NETFAIL	RMS$_NORMAL
RMS$_OPNOTSUP	RMS$_PENDING	RMS$_RAB
RMS$_RNL	RMS$_RSA	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT

$REMOVE

The Remove service deletes a file name from a directory. It is the reverse of the Enter service.

Format

SYS$REMOVE fab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage: fab
type: longword (unsigned)
access: modify
mechanism: by reference

FAB control block whose contents are to be used as indirect arguments for the Remove service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Remove service searches for the first file name that matches the expanded name string and directory ID in the user's NAM block, and then it deletes the file name without deleting the actual file. The Remove service is similar to the Erase service, except that the Erase service also deletes the file after performing an implicit Remove. Note that you must close the file before invoking the Remove service (that is, the value of FAB$W_IFI must be 0).

The Remove service accepts wildcard characters and search lists, and it is usually preceded by a call to the Parse service in order to fill in the appropriate fields of the NAM block. Because the Remove service returns the wildcard context field of the NAM block (NAM$L_WCC), the Remove service can be used on multiple successive calls in order to remove successive file names that match a wildcard file specification.

Be careful when you mix calls to the Search and Remove services. For example, assume you invoke the following service sequence:

1. PARSE
2. SEARCH
3. REMOVE
4. SEARCH
5. REMOVE

RMS responds by doing the following:

1. Searches for the first file specification that matches the expanded name string
2. Searches for and removes the second file specification
3. Searches for the third file specification
4. Searches for and removes the fourth file specification

If you want to remove the directory entry of a file and you have that file's ID, then you can improve the speed of the Remove service by specifying the NAM bit in the FAB$L_FOP field. In order to do this properly, you must first parse the name of the file specification (to clear the NAM$W_FID field), place the correct FID in the NAM block, and then perform the Remove service.

The Remove service is not supported for DECnet for OpenVMS operations on files at remote OpenVMS systems.

Table RMS–51 lists the control block fields read as input by the Remove service. For additional information on the fields accessed by this service, see Part II.

Table RMS–51 Remove Service FAB and NAM Block Input Fields

Field Name	Description
FAB$L_FOP[1]	File-processing option, FAB$V_NAM only. NAM block inputs: indicates that the NAM$W_FID field is used as input.
FAB$W_IFI	Internal file identifier (must be zero).

[1]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–51 (Cont.) Remove Service FAB and NAM Block Input Fields

Field Name	Description
FAB$L_NAM	Name block address.
NAM$W_DID[1]	Directory identification; identifies the directory from which the file is to be removed.
NAM$T_DVI[1]	Device identification; identifies the device containing the directory from which the file is to be removed.
NAM$L_ESA	Expanded string area address specifying the name, type, and version of the file to be removed.
NAM$B_ESL	Expanded string length.
NAM$W_FID[1]	File identification: if nonzero and FAB$L_FOP field FAB$V_NAM bit is set in the input FAB, the first file in the directory with this file identification is removed.
NAM$L_FNB	File name status bits (wildcard character bits only).
NAM$L_RSA	Resultant string area address: specifies the name, type, and version number of the last file removed (required for wildcard character processing).
NAM$B_RSL	Resultant string length.
NAM$B_RSS	Resultant string area size.
NAM$L_WCC	Wildcard character context value.

[1]This field is not supported for DECnet for OpenVMS operations.

Table RMS–52 lists the control block fields written as output by the Remove service.

Table RMS–52 Remove Service FAB and NAM Block Output Fields

Field Name	Description
FAB$L_STS	Completion status code (also returned in register 0).
FAB$L_STV	Status value.
NAM$B_RSL	Resultant string length.
NAM$L_WCC	Wildcard context value.

The resultant string is moved to the buffer described by the NAM$L_RSA and NAM$B_RSS fields (only if both fields are nonzero on input).

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_BLN	RMS$_CDA	RMS$_CHN
RMS$_DEV	RMS$_DME	RMS$_DNF
RMS$_DNR	RMS$_DVI	RMS$_ESA
RMS$_ESL	RMS$_FAB	RMS$_FNF

RMS$_IFI	RMS$_NAM	RMS$_NMF
RMS$_NORMAL	RMS$_PRV	RMS$_RMV
RMS$_RSL	RMS$_RSS	RMS$_RST
RMS$_STR	RMS$_SUC	RMS$_SUP
RMS$_SUPPORT	RMS$_SYS	RMS$_WCC
RMS$_WLD	RMS$_WLK	

$RENAME

You can use this service to change the name, type, or version of a file, or to move a file to another directory by changing its directory specification. However, note that you *cannot* use this service to move a file to another device.

Format

SYS$RENAME old-fab ,[err] ,[suc] ,new-fab

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

old-fab

OpenVMS usage:	fab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

FAB control block whose contents are to be used as indirect arguments for the Rename service call. The **old-fab** argument is the address of the FAB control block that specifies the old file name.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

new-fab

OpenVMS usage: fab
type: longword (unsigned)
access: modify
mechanism: by reference

The **new-fab** argument is the address of the FAB control block that specifies the new file name.

Note

If you invoke the Rename service using the $RENAME macro and if you do not specify arguments, you must construct an additional field within your argument list to contain the address of the FAB that specifies the new file name. This additional field is placed in the argument list following the field for the success completion routine (see Part I), and the argument count is set to 4.

Description

The Rename service performs the equivalent of two Parse services (old and new name), a Search service for the old directory, an Enter service to insert the new file name into the new directory, and a Remove service to delete the old file name from the old directory.

No wildcard character specifications are allowed. You can move a file from one directory to another using this service, but both directories must be on the same disk device.

If the Rename service is successful, the new directory entry is created and the old entry is deleted. If the service fails, the old entry remains and the new entry is deleted.

Table RMS–53 lists the fields in two FABs and two NAM blocks that the Rename service uses as input and output. In the table, these blocks are called FAB1 and NAM1 for the old entry, and FAB2 and NAM2 for the new entry. For output, FAB2 is not used, although it must be in writable memory. To check or signal the completion codes in FAB$L_STS and FAB$L_STV, use the first FAB (FAB1).

The resultant file specification string for each of the names (old and new) is placed in the buffer described by the NAM$L_RSA and NAM$B_RSS fields of the separate NAM blocks (only if both fields are nonzero).

Table RMS–53 lists the control block fields read as input by the Rename service. For additional information on the fields accessed by this service, see Part II.

Table RMS–53 Rename Service FAB and NAM Block Input Fields

Control Block	Field Name	Description
FAB1 and FAB2	FAB$L_DNA	Default file specification string address.
	FAB$B_DNS	Default file specification string size.
	FAB$L_FNA	File specification string address.
	FAB$B_FNS	File specification string size.
	FAB$W_IFI	Internal file identifier (must be zero).
	FAB$L_NAM	Name block address.
NAM1 and NAM2	NAM$L_ESA	Expanded string area address (must be nonzero).
	NAM$B_ESS	Expanded string area size (must be nonzero).
	NAM$L_RLF	Related file NAM block address.
	NAM$L_RSA	Resultant string area address.
	NAM$B_RSS	Resultant string area size.
Related file NAM blocks	NAM$L_RSA	Related file resultant string area address.
	NAM$B_RSL	Related file resultant string length.
	NAM$L_FNB	Related file name status bits.

Table RMS–54 lists the control block fields written as output by the Rename service.

Table RMS–54 Rename Service FAB and NAM Block Output Fields

Control Block	Field Name	Description
FAB1	FAB$L_STS	Completion status code (also returned in register 0).
	FAB$L_STV	Status value.
NAM1 and NAM2	NAM$W_DID[1]	Directory identification.
	NAM$T_DVI[1]	Device identification.
	NAM$B_ESL	Expanded string length.
	NAM$W_FID[1]	File identification.
	NAM$L_FNB	File name status bits.
	NAM$B_RSL	Resultant string length.
	NAM$L_WCC	Wildcard context.

[1]This field is not supported for DECnet for OpenVMS operations.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACC	RMS$_BLN	RMS$_BUG_DDI
RMS$_CDA	RMS$_CHN	RMS$_DEV
RMS$_DIR	RMS$_DME	RMS$_DNA
RMS$_DNF	RMS$_DNR	RMS$_DVI
RMS$_ENT	RMS$_ESA	RMS$_ESS
RMS$_FAB	RMS$FNA	RMS$_FNM
RMS$_IDR	RMS$_IFI	RMS$_LNE
RMS$_NAM	RMS$_NET	RMS$_NETFAIL
RMS$_NMF	RMS$_NORMAL	RMS$_PRV
RMS$_QUO	RMS$_REENT	RMS$_RLF
RMS$_RMV	RMS$_RSS	RMS$_RST
RMS$_RUNDOWN	RMS$_STR	RMS$_SUC
RMS$_SUPPORT	RMS$_SYN	RMS$_SYS
RMS$_TYP	RMS$_VER	RMS$_WLD

$REWIND

The Rewind service sets the context of a record stream to the first record in the file. RMS alters the context of the next record to indicate the first record as being the next record.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Rewind service on OpenVMS Alpha systems. ♦

Format

SYS$REWIND rab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Rewind service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Rewind service implicitly performs the Flush and Free services, writing out all I/O buffers and releasing all locked records. This service is valid for all file organizations on disk volumes and for sequential files on tape volumes. For indexed files, the key of reference field establishes the index to be used for subsequent sequential accesses. You cannot rewind a unit record device (such as a card reader), indirectly accessed process-permanent files, or a file that was opened with the FAB$V_SQO option set.

Table RMS–55 lists the control block fields read as input by the Rewind service. For additional information on the fields accessed by this service, see Part II.

Table RMS–55 Rewind Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$B_KRF		Key of reference (used only with indexed files).
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Rewind service asynchronously.

Table RMS–56 lists the control block fields written as output by the Rewind service.

Table RMS–56 Rewind Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BOF	RMS$_BUG_DAP
RMS$_CDA	RMS$_DME	RMS$_DNR
RMS$_DPE	RMS$_EXT_ERR	RMS$_IOP
RMS$_ISI	RMS$_KRF	RMS$_NET
RMS$_NETFAIL	RMS$_NORMAL	RMS$_OPNOTSUP
RMS$_PENDING	RMS$_QUO	RMS$_RAB
RMS$_RSA	RMS$_STR	RMS$_SUC
RMS$_SUP	RMS$_SUPPORT	RMS$_SYS
RMS$_WBE	RMS$_WER	RMS$_WLK

$SEARCH

The Search service scans a directory file and fills in various NAM block fields. This service should be preceded by the Parse service, in order to initialize the NAM block appropriately.

Format

SYS$SEARCH fab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset FAB$L_STS. Symbolic offset FAB$L_STV may contain additional status information.

Arguments

fab

OpenVMS usage: fab
type: longword (unsigned)
access: modify
mechanism: by reference

FAB control block whose contents are to be used as indirect arguments for the Search service call. The **fab** argument is the address of the FAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The basic functions of the Search service and the Parse service are performed automatically as part of the Open, Create, and Erase services. Note that you must close the file before invoking the Search service (FAB$W_IFI must be 0).

When called, the Search service scans the directory file specified by the directory identification (NAM$W_DID) field of the NAM block. It looks for an entry that matches the file name, type, and version number specified by the expanded string area address and expanded string length fields. Upon finding a match, RMS returns the file name, type, and version number in the buffer described by the resultant string area address and size fields. RMS also fills in the file identification field to enable a subsequent open-by-NAM-block operation. You can also use the Search service to obtain a series of file specifications whose names match a file specification that contains wildcard characters or search lists.

The resultant file specification string is placed in the buffer described by the resultant string area address (NAM$L_RSA) and size (NAM$B_RSS) fields of the NAM block (only if both fields are nonzero). The NAM$L_RSA and NAM$B_RSS fields must be specified (nonzero) for wildcard character processing.

Table RMS–57 lists the control block fields read as input by the Search service. For additional information on the fields accessed by this service, see Part II.

Table RMS–57 Search Service FAB and NAM Block Input Fields

Field Name	Option	Description
FAB$B_ACMODES		File access modes.
	FAB$V_CHAN_MODE[1]	This field can be used to override the access mode protection for a specified I/O operation (see Section 4.8).
FAB$W_IFI		Internal file identifier (must be zero).
FAB$L_NAM		Name block address.
NAM$W_DID[1]		Directory identification of the directory to be searched.
NAM$T_DVI[1]		Device identification of device containing directory to be searched.
NAM$L_ESA		Expanded string area address: specifies file name, type, and version of file.
NAM$B_ESL		Expanded string length.
NAM$L_FNB		File name status bits (wildcard character options only).
NAM$B_NOP		NAM block options.

[1]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–57 (Cont.) Search Service FAB and NAM Block Input Fields

Field Name	Option	Description
	NAM$V_PWD	Password: indicates that a password contained in a DECnet for OpenVMS access control string, if present in a file specification, is to be left unaltered in the expanded and resultant strings (instead of being replaced by the word ''password'').
	NAM$V_NOCONCEAL	Do not conceal device name: indicates that when a concealed device logical name is present, the concealed device logical name is to be replaced by the actual physical device name in the resultant string.
	NAM$V_SRCHXABS	Performs Display service on remote files (for output fields, see description of Display service).
NAM$L_RSA		Resultant NAM block string area address: specifies name, type, and version of last file found (required for wildcard character processing).
NAM$B_RSS		Resultant NAM block string area size.
NAM$L_WCC		Wildcard character context value.

Table RMS–58 lists the control block fields written as output by the Search service.

Table RMS–58 Search Service FAB and NAM Block Output Fields

Field Name	Description
FAB$L_STS	Completion status code (also returned in register 0).
FAB$L_STV	Status value.
NAM$L_DEV	Address of file specification device string.
NAM$B_DEV	Size of file specification device string.
NAM$L_DIR	Address of file specification directory string.
NAM$B_DIR	Size of file specification directory string.
NAM$W_FID[1]	File identification.
NAM$L_FNB	File name status bits (wildcard status bits only).
NAM$L_NAME	Address of file specification name string.
NAM$B_NAME	Size of file specification name string.
NAM$L_NODE	Address of file specification node string.

[1]This field is not supported for DECnet for OpenVMS operations.

(continued on next page)

Table RMS–58 (Cont.) Search Service FAB and NAM Block Output Fields

Field Name	Description
NAM$B_NODE	Size of file specification node string.
NAM$B_RSL	Resultant string length.
NAM$L_TYPE	Address of file specification type string.
NAM$B_TYPE	Size of file specification type string.
NAM$L_VER	Address of file specification version string.
NAM$B_VER	Size of file specification version string.
NAM$L_WCC	Wildcard character context value.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACS	RMS$_BLN	RMS$_CHN
RMS$_DEV	RMS$_DME	RMS$_DNF
RMS$_DNR	RMS$_DVI	RMS$_ESA
RMS$_ESL	RMS$_FAB	RMS$_FND
RMS$_FNF	RMS$_IFI	RMS$_NAM
RMS$_NET	RMS$_NETFAIL	RMS$_NMF
RMS$_NORMAL	RMS$_NOVALPRS	RMS$_PRV
RMS$_RSL	RMS$_RSS	RMS$_RST
RMS$_STR	RMS$_SUC	RMS$_SUP
RMS$_SUPPORT	RMS$_SYS	RMS$_WCC

$SPACE

The Space service lets you space (skip) a tape file forward or backward a specified number of blocks.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Space service on OpenVMS Alpha systems. ♦

Format

SYS$SPACE rab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Space service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Space service is intended primarily for use with magnetic tape files; the tape is skipped forward or backward the number of blocks specified in the bucket number field. (The size of each block on any tape is specific to that tape and is defined on the tape itself, not by OpenVMS or RMS.) If the value in this field is positive, the tape skips forward; if the value is negative, the tape skips backward. For disk files, the next block pointer (NBP) is updated to reflect the new sequential operation position.

Table RMS–59 lists the control block fields read as input by the Space service. For additional information on the fields accessed by this service, see Part II.

Table RMS–59 Space Service RAB Input Fields

Field Name	Option	Description
RAB$L_BKT		Bucket code. Indicates the number of blocks to space forward (positive value) or backward (negative value).
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous. Performs Space service asynchronously.

Table RMS–60 lists the control block fields written as output by the Space service.

Table RMS–60 Space Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value (the absolute number of blocks actually skipped; the value is always positive).

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_BLN	RMS$_BOF
RMS$_BUG_DAP	RMS$_CDA	RMS$_DME
RMS$_DNR	RMS$_DPE	RMS$_EOF
RMS$_IOP	RMS$_ISI	RMS$_NET
RMS$_NETFAIL	RMS$_NORMAL	RMS$_PENDING
RMS$_RAB	RMS$_RSA	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS	RMS$_WBE	

$TRUNCATE

The Truncate service shortens a sequential file.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Truncate service on OpenVMS Alpha systems. ♦

Format

SYS$TRUNCATE rab [,[err] [,suc]]

Returns

OpenVMS usage: cond_value
type: longword
access: write only
mechanism: by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage: rab
type: longword (unsigned)
access: modify
mechanism: by reference

RAB control block whose contents are to be used as indirect arguments for the Truncate service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage: ast_procedure
type: procedure value
access: call without stack unwinding
mechanism: by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The Truncate service shortens a sequential file by resetting the logical end-of-file position to the beginning of the current record. You can then append records to the file by issuing successive Put services.

The record access mode determines the current record position for the Truncate service as follows:

- In sequential record access mode, you can only use this service immediately after setting the context of the current record by successfully executing a Get or Find service.
- In random-access-by-key mode, RMS establishes the current record position as defined by the key of reference or the relative record number (RRN), as applicable.
- In random-access-by-RFA (record file address) mode, RMS establishes the current record position as defined by the RFA.

The Truncate service does not deallocate space between the end-of-file position and the physical end of the file, nor does it overwrite the records between the end-of-file position and the physical end of the file with an erase pattern. You can read records from the end-of-file position to the physical end of the file by setting the end-of-file position to the highest block allocated using the DCL command SET FILE/END_OF_FILE. If you want to erase the data between the logical end of the file and the physical end of the file, either you must delete the file and write a new one, or your application program must overwrite the records you want to delete.

The Truncate service requires that the file access (FAB$B_FAC) field specify a truncate access (FAB$V_TRN). Truncating a file opened for shared access may have unexpected results if other accessors are currently accessing the file beyond the end-of-file position.

Table RMS–61 lists the control block fields read as input by the Truncate service. For additional information on the fields accessed by this service, see Part II.

Table RMS–61 Truncate Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Truncate service asynchronously.

Table RMS–62 lists the control block fields written as output by the Truncate service.

Table RMS–62 Truncate Service RAB Output Fields

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BUG_DAP	RMS$_CDA
RMS$_CUR	RMS$_DEADLOCK	RMS$_DME
RMS$_DNR	RMS$_DPE	RMS$_EXT_ERR
RMS$_FAC	RMS$_IOP	RMS$_ISI
RMS$_NET	RMS$_NETFAIL	RMS$_NORMAL
RMS$_OPNOTSUP	RMS$_PENDING	RMS$_RAB
RMS$_RER	RMS$_RSA	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_SYS	RMS$_WBE	RMS$_WER
RMS$_WLK		

$UPDATE

The Update service allows you to modify the contents of an existing record in a file residing on a disk device.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Update service on OpenVMS Alpha systems. ♦

Format

SYS$UPDATE rab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Update service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

The record to be updated by the Update service must first be locked by this stream, using either a Find or Get service. When updating a record, you must use move mode (not locate mode); that is, you must supply a buffer.

The record length for sequential files cannot change. For relative files with variable-length or variable with fixed-length control records, the length of the replacement record can be different from the length of the original record, but cannot be larger than the maximum size you set when you created the file.

For stream format files, the Update service functions in the same manner as the Put service, with one exception: when using the Update service, you do not have to set the RAB$L_ROP field RAB$V_TPT option to update data in the middle of a file.

For indexed files, the length of the replacement record written by the Update service may be different from the original record, but RMS does not permit you to change the primary key. Each replacement record must be large enough to contain a complete primary key, but it does not have to contain all alternate keys.

If an alternate key is partially or completely missing in the replacement record, the key must have the characteristic that the values can change; this is also true if the replacement record contains a key that was not present in the original record.

Update operations to an indexed file do not require a key value or key of reference. Before writing the record, RMS compares the key values (primary and alternate) in the replacement record with the key values of the original record already existing in the file. This comparison takes the defined characteristics of each key into account. For example, if a particular key is not allowed to change, RMS rejects the operation with an RMS$_CHG error code if the replacement record contains an altered value in the associated key. Similarly, this comparison determines whether the replacement record would result in the presence of duplicate key values among records of the file. If duplicates would occur, RMS verifies the defined characteristics for the keys being duplicated. If duplicates are not allowed for a particular key, RMS rejects the operation with an RMS$_DUP error code. If duplicates are allowed, RMS performs the operation.

Subsequent sequential operations on a given index retrieve records with identical key values in the order in which the records were written.

RAB Control Block Fields

Table RMS–63 lists the control block fields read as input by the Update service. For additional information on the fields accessed by this service, see Part II.

Table RMS–63 Update Service RAB Input Fields

Field Name	Option	Description
RAB$W_ISI		Internal stream identifier (required).
RAB$L_RBF		Record buffer address.
RAB$L_RHB		Record header buffer (for variables with fixed control records only).

(continued on next page)

Table RMS–63 (Cont.) Update Service RAB Input Fields

Field Name	Option	Description
RAB$L_ROP		Record-processing options.
	RAB$V_ASY	Asynchronous: performs Update services asynchronously.
	RAB$V_WBH	Write-locked: two buffers are allocated to allow multibuffering.
RAB$W_RSZ		Record size (required).

Table RMS–64 lists the control block fields written as output by the Update service.

Table RMS–64 Update Service RAB Output Fields

Field Name	Description
RAB$W_RFA	Record file address.
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value.

RAB64 Control Block Fields (Alpha Only)

Alpha

Table RMS–65 lists the Alpha-only RAB64 control block fields read as input by the Update service. These fields are comparable to the RAB fields described in Table RMS–63. For additional information on the fields accessed by this service, see Part II.

Table RMS–65 Update Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$B_BLN	This field must be initialized to RAB64$C_BLN64 in order for RAB64 fields to be used.
RAB64$W_ISI	Internal stream identifier (required). Equates to RAB$W_ ISI.
RAB64$L_RBF[1]	Record buffer address. This field must contain -1 if you want to use RAB64$PQ_RBF. For 32-bit addressing, this field equates to RAB$L_RBF.
RAB64$PQ_RBF[1]	Record buffer 64-bit address (used if RAB64$L_RBF contains -1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.
RAB64$L_RHB	Record header buffer. This field must contain -1 if you want to use RAB64$PQ_RHB. For 32-bit addressing, this field equates to RAB$L_RHB (see Table RMS–63).
RAB64$PQ_RHB	Record header buffer 64-bit address (used if RAB64$L_RHB contains -1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

[1]One of the RBF fields must contain an address and the RSZ field associated with it must contain a size.

(continued on next page)

Table RMS–65 (Cont.) Update Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$L_ROP	Record-processing options. Equates to RAB$L_ROP and options described in Table RMS–63. Options are identical except for the RAB64 prefix; for example, option RAB64$V_ASY equates to RAB$V_ASY.
RAB64$W_RSZ[1]	Record buffer size. This field is ignored in favor of RAB64$Q_RSZ if RAB64$L_RBF contains -1. Otherwise, this field equates to RAB$W_RSZ.
RAB64$Q_RSZ[1]	Record buffer size. This field must be used when RAB64$L_RBF contains -1 and a value is specified in RAB64$PQ_RBF.

[1]One of the RBF fields must contain an address and the RSZ field associated with it must contain a size.

Table RMS–66 lists the Alpha-only RAB64 control block fields written as output by the Update service. These fields are comparable to the RAB fields described in Table RMS–64.

Table RMS–66 Update Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA.
RAB64$L_STS	Completion status code. Equates to RAB$L_STS (see Table RMS–64).
RAB64$L_STV	Status value. Equates to RAB$L_STV. ♦

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BUG	RMS$_BUG_DAP
RMS$_CDA	RMS$_CHG	RMS$_CHK
RMS$_CUR	RMS$_DME	RMS$_DNR
RMS$_DUP	RMS$_ENQ	RMS$_EXP
RMS$_EXT_ERR	RMS$_FAC	RMS$_FTM
RMS$_IBF	RMS$_IDX	RMS$_IOP
RMS$_NETBTS	RMS$_NETFAIL	RMS$_NORMAL
RMS$_OK_DUP	RMS$_OK_IDX	RMS$_OPNOTSUP
RMS$_PENDING	RMS$_PLG	RMS$_RAB
RMS$_RBF	RMS$_RER	RMS$_RHB
RMS$_RNL	RMS$_RPL	RMS$_RRV
RMS$_RSA	RMS$_RSZ	RMS$_RVU

RMS$_STR	RMS$_SUC	RMS$_SUP
RMS$_SUPPORT	RMS$_SYS	RMS$_TRE
RMS$_WBE	RMS$_WER	RMS$_WLK
RMS$_WPL		

$WAIT

The Wait service suspends image execution until an asynchronous record service completes. Upon completion of the service, RMS returns control to your program at the point following the Wait service call.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Wait service on OpenVMS Alpha systems. ♦

Format

SYS$WAIT rab

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Wait service call. The **rab** argument is the address of the RAB control block. It can be either the RAB whose I/O request is in progress or some other RAB.

Description

The Wait service takes no arguments to define entry points for user-written completion routines; the completion routines are specified by the service being awaited.

Any completion routines specified on the operation being awaited are declared as ASTs before RMS returns control. They are executed before the Wait service completes unless ASTs are disabled. Completion routines are always executed as asynchronous system traps (ASTs).

Table RMS–67 lists the control block fields used as input by the Wait service. For additional information on the fields accessed by this service, see Part II.

Table RMS–67 Wait Service RAB Input Fields

Field Name	Description
RAB$W_ISI	Internal stream identifier (required).
RAB$L_STS	Status completion code.

Table RMS–68 lists the control block field written as output by the Wait service.

Table RMS–68 Wait Service RAB Output Field

Field Name	Description
RAB$L_STS	Completion status code (also returned in register 0).

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_BLN	RMS$_CDA	RMS$_EXT_ERR
RMS$_ISI	RMS$_NORMAL	RMS$_OPNOTSUP
RMS$_RAB	RMS$_STR	RMS$_SUC

The RMS completion status codes for the Wait service are determined by the service being awaited, unless the address of the RAB specified for the wait is different from that specified for the awaited operation. In this case, RMS$_NORMAL is returned.

$WRITE

The Write service transfers a user-specified number of bytes (beginning on a block boundary) to an RMS file of any file organization.

Alpha

RAB64 Users

RAB64 can replace the RAB or RAB prefix wherever it is used with the Write service on OpenVMS Alpha systems. ♦

Format

SYS$WRITE rab [,[err] [,suc]]

Returns

OpenVMS usage:	cond_value
type:	longword
access:	write only
mechanism:	by value

The value is returned in symbolic offset RAB$L_STS. Symbolic offset RAB$L_STV may contain additional status information.

Arguments

rab

OpenVMS usage:	rab
type:	longword (unsigned)
access:	modify
mechanism:	by reference

RAB control block whose contents are to be used as indirect arguments for the Write service call. The **rab** argument is the address of the RAB control block.

err

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level error completion routine that the service invokes if the operation is unsuccessful. The **err** argument is the address of the entry mask of this user-written completion routine.

suc

OpenVMS usage:	ast_procedure
type:	procedure value
access:	call without stack unwinding
mechanism:	by reference

AST-level success completion routine that the service invokes if the operation is successful. The **suc** argument is the address of the entry mask of this user-written completion routine.

Description

To use the Write service, you must do the following:

1. Supply a buffer area and specify the buffer size:
   - To supply a 32-bit buffer address and a buffer size no greater than 65,535 bytes, use these fields:

User Buffer Address Field	User Buffer Size Field
RAB$L_RBF	RAB$W_RSZ

Alpha

   - On OpenVMS Alpha systems, you can supply a 64-bit buffer address (or a 32-bit address sign-extended to 64 bits) and a buffer size up to $2**31-1$ bytes. To do so, put a -1 in RAB64$L_RBF and use these fields:

User Buffer Address Field	User Buffer Size Field
RAB64$PQ_RBF	RAB64$Q_RSZ ♦

2. Indicate the virtual block number (VBN) of the first block to be written in the bucket number field. This field is RAB$L_BKT or RAB64$L_BKT (available only on Alpha to accommodate 64-bit addressing). If the value for the VBN is 0, the transfer starts with the block indicated by the next block pointer (NBP).

A sequential file is automatically extended if you write a block past the end of the currently allocated space when using block I/O (or record I/O). For sequential files, RMS maintains a logical end of file to correspond to the last block and highest byte written within the block. For relative and indexed files, you must use the Extend service when using block I/O.

RAB Control Block Fields

Table RMS–69 lists the control block fields read as input by the Write service. For additional information on the fields accessed by this service, see Part II.

Table RMS–69 Write Service RAB Input Fields

Field Name	Option	Description
RAB$L_BKT		Bucket number: must contain the virtual block number of the first block to be written.
RAB$W_ISI		Internal stream identifier.
RAB$L_RBF		Record buffer address. For block I/O, alignment of the user's record buffer on a page or at least a quadword boundary may improve performance.
RAB$L_ROP		Record-processing options.

(continued on next page)

Table RMS–69 (Cont.) Write Service RAB Input Fields

Field Name	Option	Description
	RAB$V_ASY	Asynchronous: performs Write services asynchronously.
	RAB$V_TPT	Truncate on Put: specifies that a Write service truncate the file after the transferred data.
RAB$W_RSZ		Record size: indicates the transfer length, in bytes.[1]

[1]Certain devices require that an even number of bytes be transferred. For further details, see the *OpenVMS I/O User's Reference Manual*.

Table RMS–70 lists the control block fields written as output by the Write service.

Table RMS–70 Write Service RAB Output Fields

Field Name	Description
RAB$W_RFA	Record file address.
RAB$L_STS	Completion status code (also returned in register 0).
RAB$L_STV	Status value: contains the actual number of bytes transferred if an end-of-file error occurs.

RAB64 Control Block Fields (Alpha Only)

Alpha

Table RMS–71 lists the Alpha-only RAB64 control block fields read as input by the Write service. These fields are comparable to the RAB fields described in Table RMS–69. For additional information on the fields accessed by this service, see Part II.

Table RMS–71 Write Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$B_BLN	This field must be initialized to RAB64$C_BLN64 in order for RAB64 fields to be used.
RAB64$L_BKT	Bucket number. Equates to RAB$L_BKT (see Table RMS–69).
RAB64$W_ISI	Internal stream identifier. Equates to RAB$W_ISI.
RAB64$L_RBF[1]	Record buffer address. This field must contain -1 if you want to use RAB64$PQ_RBF. For 32-bit addressing, this field equates to RAB$L_RBF (see Table RMS–69).
RAB64$PQ_RBF[1]	Record buffer 64-bit address (used if RAB64$L_RBF contains -1). This field can hold either a 64-bit address or a 32-bit address sign-extended to 64 bits.

[1]One of the RBF fields must contain an address and the RSZ field associated with it must contain a size.

(continued on next page)

Table RMS–71 (Cont.) Write Service RAB64 Input Fields (Alpha Only)

Field Name	Description
RAB64$L_ROP	Record-processing options. Equates to RAB$L_ROP and options described in Table RMS–69. Options are identical except for the RAB64 prefix; for example, option RAB64$V_ASY equates to RAB$V_ASY.
RAB64$W_RSZ[1]	Record buffer size. This field is ignored in favor of RAB64$Q_RSZ if RAB64$L_RBF contains −1. Otherwise, this field equates to RAB$W_RSZ (see Table RMS–69).
RAB64$Q_RSZ[1]	Record buffer size. This field must be used when RAB64$L_RBF contains −1 and a value is specified in RAB64$PQ_RBF. See Section 7.6 for more information.

[1]One of the RBF fields must contain an address and the RSZ field associated with it must contain a size.

Table RMS–72 lists the Alpha-only RAB64 control block fields written as output by the Write service. These fields are comparable to the RAB fields described in Table RMS–70.

Table RMS–72 Write Service RAB64 Output Fields (Alpha Only)

Field Name	Description
RAB64$W_RFA	Record file address. Equates to RAB$W_RFA.
RAB64$L_STS	Completion status code. Equates to RAB$L_STS (see Table RMS–70).
RAB64$L_STV	Status value. Equates to RAB$L_STV (see Table RMS–70). ♦

Condition Values Returned

The following condition values can be returned. Use the Help Message utility to access online message descriptions. For more information about interpreting condition values, see Section 2.4.

RMS$_ACT	RMS$_ATR	RMS$_ATW
RMS$_BLN	RMS$_BUG_DAP	RMS$_CDA
RMS$_CONTROLC	RMS$_CONTROLO	RMS$_CONTROLY
RMS$_DME	RMS$_DNR	RMS$_EOF
RMS$_EXT	RMS$_FAC	RMS$_FTM
RMS$_FUL	RMS$_IOP	RMS$_ISI
RMS$_NET	RMS$_NETFAIL	RMS$_NORMAL
RMS$_PENDING	RMS$_RAB	RMS$_RBF
RMS$_RSA	RMS$_RSZ	RMS$_STR
RMS$_SUC	RMS$_SUP	RMS$_SUPPORT
RMS$_WBE	RMS$_WER	RMS$_WLK

A

RMS Control Block Macros

This appendix lists the format of each RMS control block macro and includes special syntax notes that differ from the rules provided in Part I. Note that in this appendix the use of the term ‘‘macro’’ refers to a VAX MACRO macro.

$FAB

The $FAB macro allocates storage for a FAB and initializes certain FAB fields with defaults and user-specified values. No value is returned for this assembly-time operation.

Format

```
$FAB  ALQ=allocation-quantity,
      BKS=bucket-size,
      BLS=block-size,
      CHAN_MODE=channel-access-mode
      CTX=user-context-value,
      DEQ=extension-quantity,
      DNA=default-filespec-address,
      DNM=<filespec>,
      DNS=default-filespec-string-size,
      FAC=<BIO BRO DEL GET PUT TRN UPD>,
      FNA=filespec-string-address,
      FNM=<filespec>,
      FNS=filespec-string-size,
                <CBT CIF CTG DFW DLT
                 MXV NAM NEF NFS OFP
      FOP=       POS RCK RWC RWO SCF
                 SPL SQO SUP TEF TMD
                 TMP UFO WCK>,
      FSZ=header-size,
      GBC=global-buffer-count,
      LNM_MODE=  logical-name-translation-
                 access-mode,
      MRN=maximum-record-number,
      MRS=maximum-record-size,
      NAM=nam-address,
           { IDX }
      ORG= { REL } ,
           { SEQ }
           { CR         }
      RAT= { <BLK FTN>  } ,
           { PRN        }
           { FIX    }
           { STM    }
           { STMCR  }
      RFM= { STMLF  } ,
           { UDF    }
           { VAR    }
           { VFC    }
      RTV=window-size,
      SHR=<DEL GET MSE NIL PUT UPD UPI>,
      XAB=xab-address
```

Arguments

For a description of the control block fields that correspond to the $FAB macro arguments, see Chapter 4. In some cases, specific default values are assigned automatically when you omit an argument. If there is no specific default, RMS uses a default value of 0.

Arguments fall into three categories: values, addresses, and keywords. Rules applicable to these argument categories are described in Appendix B.

Note that multiple arguments can be specified for the FAC, FOP, RAT, and SHR keywords, but the arguments must be enclosed within left angle (<) and right angle (>) brackets. The DNM and FNM arguments must also be delimited by these signs.

The DNM and FNM arguments contain ASCII characters and have no corresponding field in the FAB. If the DNM argument is present, RMS places its appropriate address and size in the FAB$L_DNA and FAB$B_DNS fields. Similarly, if the FNM argument is present, RMS places its appropriate address and size in the FAB$L_FNA and FAB$B_FNS fields.

$FAB_STORE

The $FAB_STORE macro moves user-specified values into fields of the specified FAB. The expanded $FAB_STORE code executes at run time on a previously initialized (allocated) FAB, in contrast to the $FAB macro, which initializes the FAB at assembly time. The $FAB_STORE macro must reside in a code program section.

Format

```
$FAB_STORE   fab=fab-address,
             ALQ=#allocation-quantity,
             BKS=#bucket-size,
             BLS=#block-size,
             CHAN_MODE=#channel-access-mode
             CTX=user-context-value,
             DEQ=#extension-quantity,
             DNA=default-filespec-address,
             DNS=#default-filespec-string-size,
             FAC=<BIO BRO DEL GET PUT TRN UPD>,
             FNA=filespec-string-address,
             FNS=#filespec-string-size,
                   <CBT CIF CTG DFW DLT
                   MXV NAM NEF NFS OFP
             FOP=  POS RCK RWC RWO SCF
                   SPL SQO SUP TEF TMD
                   TMP UFO WCK>,
             FSZ=#header-size,
             GBC=#global-buffer-count,
             LNM_MODE=#logical-name-translation-access-mode,
             MRN=#maximum-record-number,
             MRS=#maximum-record-size,
             NAM=nam-address,
                   { IDX }
             ORG=  { REL } ,
                   { SEQ }
                   { CR        }
             RAT=  { <BLK FTN> } ,
                   { PRN       }
                   { FIX    }
                   { STM    }
                   { STMCR  }
             RFM=  { STMLF  } ,
                   { UDF    }
                   { VAR    }
                   { VFC    }
             RTV=#window-size,
             SHR=<DEL GET MSE NIL PUT UPD UPI>,
             XAB=xab-address
```

Arguments

For a description of the control block fields that correspond to the $FAB_STORE macro arguments, see Chapter 4.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The FAB argument **fab-address** is required for the $FAB_STORE macro and is not present for the $FAB macro. Conversely, the DNM argument **filespec** and FNM argument **default-filespec** are not available for the $FAB_STORE macro, although you can use the DNA/DNS and FNA/FNS arguments to specify file specifications at run time.

Note that R0 is usually used by the $FAB_STORE macro; thus, R0 is not preserved and does not contain a return status.

$NAM

The $NAM macro allocates storage for a NAM block and initializes certain NAM fields with default values and user-specified values. No value is returned for this assembly-time operation.

Format

```
$NAM   ESA=expanded-string-address,
       ESS=expanded-string-size,
       NOP= <NOCONCEAL PWD
             SRCHXABS SYNCHK>,
       RLF=related-file-nam-block-address,
       RSA=resultant-string-address,
       RSS=resultant-string-size
```

Arguments

For a description of the control block fields that correspond to the $NAM macro arguments, see Chapter 5.

Arguments fall into three categories: values, addresses, and keywords. Rules applicable to these argument categories are described in Appendix B.

Note that multiple arguments can be specified for the NOP keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets.

$NAM_STORE

The $NAM_STORE macro moves user-specified values into fields of the specified NAM block. The expanded $NAM_STORE code executes at run time on a previously initialized (allocated) NAM block, in contrast to the $NAM macro, which initializes a NAM block at assembly time. The $NAM_STORE macro must reside in a code program section.

Format

```
$NAM_STORE   NAM=nam-address,
             DID=#directory-identification,
             DVI=#device-identification,
             ESA=expanded-string-address,
             ESS=#expanded-string-size,
             FID=#file-identification,
             NOP= <NOCONCEAL PWD
                  SRCHXABS SYNCHK>,
             RLF=related-file-nam-block-address,
             RSA=resultant-string-address,
             RSS=#resultant-string-size
```

Arguments

For a description of the control block fields that correspond to the $NAM_STORE macro arguments, see Chapter 5.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The NAM argument **nam-address** is required for the $NAM_STORE macro and is not present for the $NAM macro. Also, the following $NAM_STORE argument fields are not available for the $NAM macro:

- The DID argument **directory-identification** sets the NAM$W_DID field, which is a 3-word field used when the FAB$L_FOP field FAB$V_NAM option is set. This argument is usually specified by its symbolic address. If a register is used to contain a value for the NAM$W_DID field, do not use R12, because two contiguous registers must be used to contain the value of this 3-word field. Note that you cannot use the byte, word, or longword displacements for an offset, or for indexed or deferred addressing.
- The DVI argument **device-identification** sets the NAM$T_DVI field, which is a 16-byte field used when the FAB$L_FOP field FAB$V_NAM option is set. This argument must be passed by its symbolic address. A register must not be specified to contain a value for this argument.
- The FID argument **file-identification** sets the NAM$W_FID field, which is a 3-word field used when the FAB$L_FOP field FAB$V_NAM option is set. This argument is specified by its symbolic address. If a register is used to contain a value for the NAM$W_FID field, do not use R12, because two contiguous registers must be used to contain the value of this 3-word field. Note that you cannot use the byte, word, or longword displacements for an offset, or for indexed or deferred addressing.

Note that R0 is usually used by the $NAM_STORE macro; thus, R0 is not preserved and does not contain a return status.

$RAB

The $RAB macro allocates storage for a RAB and initializes certain RAB fields with defaults and user-specified values. You cannot use this macro within a sequence of executable instructions. No value is returned for this assembly-time operation.

Format

```
$RAB   BKT=bucket-code-number,
       CTX=user-context-value,
       FAB=fab-address,
       KBF=key-buffer-address,
       KRF=key-of-reference-number,
       KSZ=key-size,
       MBC=multiblock-count-number,
       MBF=multibuffer-count-number,
       PBF=prompt-buffer-address,
       PSZ=prompt-buffer-size,
            { KEY }
       RAC= { RFA } ,
            { SEQ }
       RBF=record-buffer-address,
       RHB=record-header-buffer-address,
             <ASY BIO CCO CDK CVT EOF
             EQNXT ETO FDL KGE KGT LIM
       ROP=  LOA LOC NLK NXR NXT PMT
             PTA RAH REA REV RLK RNE
             RNF RRL TMO TPT UIF ULK
             WAT WBH>,
       RSZ=record-size,
       TMO=time-out-number-of-seconds,
       UBF=user-record-buffer-address,
       USZ=user-record-buffer-size,
       XAB=xab-address
```

Arguments

For a description of the control block fields that correspond to the $RAB macro arguments, see Chapter 6. In some cases, specific default values are assigned automatically when you omit an argument. These specific defaults are noted in the text that explains each field in Chapter 6. If there is no specific default, RMS uses a default value of 0.

Arguments fall into three categories: values, addresses, and keywords. Rules applicable to these argument categories are described in Appendix B.

Note that multiple arguments can be specified for the ROP keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets. Also note that the KGE and EQNXT arguments are logically synonymous, as are the KGT and NXT arguments.

$RAB_STORE

The $RAB_STORE macro moves user-specified values into fields of the specified RAB. The expanded $RAB_STORE code executes at run time on a previously initialized (allocated) RAB, in contrast to the $RAB macro, which allocates and initializes the RAB at assembly time. The $RAB_STORE macro must reside in a code program section.

Format

```
$RAB_STORE  RAB=rab-address,
            BKT=#bucket-code-number,
            CTX=user-context-value,
            FAB=fab-address,
            KBF=key-buffer-address,
            KRF=#key-of-reference-number,
            KSZ=#key-size,
            MBC=#multiblock-count-number,
            MBF=#multibuffer-count-number,
            PBF=prompt-buffer-address,
            PSZ=#prompt-buffer-size,
                  { KEY }
            RAC=  { RFA } ,
                  { SEQ }
            RBF=record-buffer-address,
            RFA=#record-file-address,
            RHB=record-header-buffer-address,
                   <ASY BIO CCO CDK
                   CVT EOF EQNXT ETO
                   FDL KGE KGT LIM
            ROP=   LOA LOC NLK NXR
                   NXT PMT PTA RAH
                   REA REV RLK RNE
                   RNF RRL TMO TPT
                   UIF ULK WAT WBH>,
            RSZ=#record-size,
            TMO=   #time-out-number-of-
                   seconds,
            UBF=user-record-buffer-address,
            USZ=#user-record-buffer-size,
            XAB=xab-address
```

Arguments

For a description of the control block fields that correspond to the $RAB_STORE macro arguments, see Chapter 6.

Arguments fall into many categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The RAB argument **rab-address** is required for the $RAB_STORE macro and is not present for the $RAB macro.

The RFA argument **record-file-address** is a value (not an address), and it is not available for the $RAB macro. The value for the 3-word RAB$W_RFA field must be set before each RFA record access. This argument is specified by its symbolic address. If a register is used to contain a value for the RAB$W_RFA field, do not use R12, because two contiguous registers must be used to contain the value of this 3-word field. Note that you cannot use the byte, word, or longword displacements for an offset, or for indexed or deferred addressing.

Note that multiple arguments can be specified for the ROP keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets. Also note that the KGE and EQNXT arguments are logically synonymous, as are the KGT and NXT arguments.

Note that R0 is usually used by the $RAB_STORE macro; thus, R0 is not preserved and does not contain a return status.

$RAB64 (Alpha Only)

Alpha

The $RAB64 macro allocates storage for a RAB64 and initializes certain RAB64 fields with defaults and user-specified values. You cannot use this macro within a sequence of executable instructions. No value is returned for this assembly-time operation.

The defaults unique to $RAB64 are as follows:

- RAB64$B_BLN is initialized to RAB64$C_BLN64.
- The original longword I/O buffers (KBF, RHB, RBF, and UBF) are initialized to −1; USZ and RSZ word sizes are initialized to 0.

User-specified values using the CTX, KBF, RHB, RBF, RSZ, UBF, or USZ keywords are moved into the quadword fields for these keywords. In contrast, the $RAB macro moves them into the longword (or word) fields for these keywords.

Format

```
$RAB64  BKT=bucket-code-number,
        CTX=user-context-value,
        FAB=fab-address,
        KBF=key-buffer-address,
        KRF=key-of-reference-number,
        KSZ=key-size,
        MBC=multiblock-count-number,
        MBF=multibuffer-count-number,
        PBF=prompt-buffer-address,
        PSZ=prompt-buffer-size,
                ⎧ KEY ⎫
        RAC=    ⎨ RFA ⎬ ,
                ⎩ SEQ ⎭
        RBF=record-buffer-address,
        RHB=record-header-buffer-address,
                <ASY BIO CCO CDK CVT EOF
                EQNXT ETO FDL KGE KGT LIM
        ROP=    LOA LOC NLK NXR NXT PMT
                PTA RAH REA REV RLK RNE
                RNF RRL TMO TPT UIF ULK
                WAT WBH>,
        RSZ=record-size,
        TMO=time-out-number-of-seconds,
        UBF=user-record-buffer-address,
        USZ=user-record-buffer-size,
        XAB=xab-address
```

Arguments

For a description of the control block fields that correspond to the $RAB64 macro arguments, see Chapter 7. In some cases, specific default values are assigned automatically when you omit an argument. These specific defaults are described above or noted in the text that explains each field in Chapter 7. If there is no specific default, RMS uses a default value of 0.

Arguments fall into three categories: values, addresses, and keywords. Rules applicable to these argument categories are described in Appendix B.

Note that multiple arguments can be specified for the ROP keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets. Also note that the KGE and EQNXT arguments are logically synonymous, as are the KGT and NXT arguments. ♦

$RAB64_STORE (Alpha Only)

Alpha

The $RAB64_STORE macro moves user-specified values into fields of the specified RAB64. The expanded $RAB64_STORE code executes at run time on a previously initialized (allocated) RAB64. In contrast, the $RAB64 macro allocates and initializes the RAB64 at assembly time. The $RAB64_STORE macro must reside in a code program section.

User-specified values that use the CTX, KBF, RHB, RBF, RSZ, UBF, or USZ keywords are moved into the quadword fields for these keywords. In contrast, the $RAB_STORE macro moves them into the longword (or word) fields for these keywords.

Format

```
$RAB64_STORE   RAB=rab64-address,
               BKT=#bucket-code-number,
               CTX=user-context-value,
               FAB=fab-address,
               KBF=key-buffer-address,
               KRF=#key-of-reference-number,
               KSZ=#key-size,
               MBC=#multiblock-count-number,
               MBF=#multibuffer-count-number,
               PBF=prompt-buffer-address,
               PSZ=#prompt-buffer-size,
                    { KEY }
               RAC= { RFA } ,
                    { SEQ }
               RBF=record-buffer-address,
               RFA=#record-file-address,
               RHB=record-header-buffer-address,
                     <ASY BIO CCO CDK
                     CVT EOF EQNXT ETO
                     FDL KGE KGT LIM
               ROP=  LOA LOC NLK NXR
                     NXT PMT PTA RAH
                     REA REV RLK RNE
                     RNF RRL TMO TPT
                     UIF ULK WAT WBH>,
               RSZ=#record-size,
               TMO= #time-out-number-of-
                    seconds,
               UBF=user-record-buffer-address,
               USZ=#user-record-buffer-size,
               XAB=xab-address
```

Arguments

For a description of the control block fields that correspond to the $RAB64_STORE macro arguments, see Chapter 7.

Arguments fall into many categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The RAB argument **rab64-address** is required for the $RAB64_STORE macro and is not present for the $RAB64 macro.

The RFA argument **record-file-address** is a value (not an address), and it is not available for the $RAB64 macro. The value for the 3-word RAB64$W_RFA field must be set before each RFA record access. This argument is specified by its symbolic address. If a register is used to contain a value for the RAB64$W_RFA field, do not use R12, because two contiguous registers must be used to contain the value of this 3-word field. Note that you cannot use the byte, word, or longword displacements for an offset, or for indexed or deferred addressing.

Note that multiple arguments can be specified for the ROP keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets. Also note that the KGE and EQNXT arguments are logically synonymous, as are the KGT and NXT arguments.

Note that R0 is usually used by the $RAB64_STORE macro; thus, R0 is not preserved and does not contain a return status. ♦

$XABALL

The $XABALL macro allocates and initializes a XABALL, which allows extended control of file disk space allocation, both for initial allocation and later extension. No value is returned for this assembly-time operation.

Format

```
$XABALL   AID=area-identification-number,
                ( ANY )
                | CYL |
          ALN=  { LBN } ,
                | RFI |
                ( VBN )
          ALQ=allocation-quantity,
          AOP=  <CBT CTG
                HRD ONC>,
          BKZ=bucket-size,
          DEQ=extension-quantity,
          LOC=location-number,
          NXT=next-xab-address,
          RFI=<f(1), f(2), f(3)>,
          VOL=volume-number
```

Arguments

For a description of the control block fields that correspond to the $XABALL macro arguments, see Chapter 8.

Arguments fall into three categories: values, addresses, and keywords. Rules applicable to these argument categories are described in Appendix B.

Note that multiple arguments can be specified for the AOP keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets.

$XABALL_STORE

The $XABALL_STORE macro moves user-specified values into fields of the specified XABALL. The expanded $XABALL_STORE code executes at run time on a previously initialized (allocated) XABALL, in contrast to the $XABALL macro, which initializes a XABALL at assembly time. The $XABALL_STORE macro must reside in a code program section.

Format

```
$XABALL_STORE  XAB=xaball-address,
               AID=#area-identification-number,
                    { ANY }
                    { CYL }
               ALN= { LBN } ,
                    { RFI }
                    { VBN }
               ALQ=#allocation-quantity,
               AOP= <CBT CTG
                    HRD ONC>,
               BKZ=#bucket-size,
               DEQ=#extension-quantity,
               LOC=#location-number,
               NXT=next-xab-address,
               RFI=#related-file-identification,
               VOL=#volume-number
```

Arguments

For a description of the control block fields that correspond to the $XABALL_STORE macro arguments, see Chapter 8.

Arguments fall into several categories: value, address, keyword, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xaball-address** is required for the $XABALL_STORE macro and is not present for the $XABALL macro. Also, the RFI argument **related file identification** sets the XAB$W_RFI field, which is a 3-word field used when the XAB$B_ALN field XAB$V_RFI option is set. This argument is usually specified by its symbolic address. If a register is used to contain a value for the XAB$W_RFI field, do not use R12, because two contiguous registers must be used to contain the value of this 3-word field. Note that you cannot use the byte, word, or longword displacements for an offset, or for indexed or deferred addressing.

Note that R0 is usually used by the $XABALL_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABDAT

The $XABDAT macro allocates and initializes a XABDAT. No value is returned for this assembly-time operation.

Format

```
$XABDAT   EDT=date-time,
          NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABDAT macro arguments, see Chapter 8.

Rules applicable to arguments are described in Appendix B.

$XABDAT_STORE

The $XABDAT_STORE macro moves user-specified values into fields of the specified XABDAT. The expanded $XABDAT_STORE code executes at run time on a previously initialized (allocated) XABDAT, in contrast to the $XABDAT macro, which initializes a XABDAT at assembly time. The $XABDAT_STORE macro must reside in a code program section.

Format

```
$XABDAT_STORE   XAB=xabdat-address,
                CDT=#creation-date-time,
                EDT=#expiration-date-time,
                RDT=#revision-date-time,
                RVN=#revision-number,
                NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABDAT_ STORE macro arguments, see Chapter 8.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabdat-address** is required for the $XABDAT_STORE macro and is not present for the $XABDAT macro. Also, the arguments differ from the general rules.

- The CDT argument **creation-date-time** sets the XAB$Q_CDT field, which is a quadword field. However, if a register is used to contain a literal value for the XAB$Q_CDT field, do not use R12, because two contiguous registers must be used to contain the value of this quadword field.
- The EDT argument **expiration-date-time** sets the XAB$Q_EDT field, which is a quadword field. The rules for the other time fields also apply to this one.
- The RDT argument **revision-date-time** sets the XAB$Q_CDT field, which is a quadword field. The rules for the other time fields also apply to this one.

Note that R0 is usually used by the $XABDAT_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABFHC

The $XABFHC macro allocates and initializes a XABFHC. No value is returned for this assembly-time operation.

Format

$XABFHC NXT=next-xab-address

Arguments

For a description of the control block fields that correspond to the $XABFHC macro arguments, see Chapter 10.

Rules applicable to arguments are described in Appendix B.

$XABFHC_STORE

The $XABFHC_STORE macro moves user-specified values into fields of the specified XABFHC. The expanded $XABFHC_STORE code executes at run time on a previously initialized (allocated) XABFHC, in contrast to the $XABFHC macro, which initializes a XABFHC at assembly time. The $XABFHC_STORE macro must reside in a code program section.

Format

```
$XABFHC_STORE   XAB=xabfhc-address,
                NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABFHC_STORE macro arguments, see Chapter 10.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabfhc-address** is required for the $XABFHC_STORE macro and is not present for the $XABFHC macro.

Note that R0 may be used by the $XABFHC_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABITM

The $XABITM macro allocates and initializes a XABITM. No value is returned for this assembly-time operation.

Format

```
$XABITM   ITEMLIST=item-list-address,
          MODE= { sensemode
                  setmode   } ,
          NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABITM macro arguments, see Chapter 11.

Rules applicable to arguments are described in Appendix B.

ITEMLIST defaults to 0 but a valid pointer must be specified when you use a $XABITM macro. MODE defaults to sensemode.

$XABKEY

The $XABKEY macro allocates and initializes a XABKEY. No value is returned for this assembly-time operation.

Format

```
$XABKEY  COLTBL=collating-table-address,
         DAN=data-bucket-area-number,
         DFL=data-bucket-fill-size,
              { BN2   }
              { DBN2  }
              { BN4   }
              { DBN4  }
              { BN8   }
              { DBN8  }
              { IN2   }
              { DIN2  }
         DTP= { IN4   } ,
              { DIN4  }
              { IN8   }
              { DIN8  }
              { COL   }
              { DCOL  }
              { PAC   }
              { DPAC  }
              { STG   }
              { DSTG  }
              <CHG DAT_NCMPR DUP
         FLG= IDX_NCMPR
              KEY_NCMPR NUL>,
         IAN=index-bucket-area-number,
         IFL=index-bucket-file-size,
         KNM=key-name-buffer-address,
         LAN=lowest-level-index-area-number,
         NUL=null-key-value,
         NXT=next-xab-address,
         POS=<position,...>,
         PROLOG=prolog-level,
         REF=key-of-reference-value,
         SIZ=<size,...>
```

Arguments

For a description of the control block fields that correspond to the $XABKEY macro arguments, see Chapter 13.

Arguments fall into three categories: values, addresses, and keywords. Rules applicable to these argument categories are described in Appendix B.

Multiple arguments can be specified for the FLG keyword, but the arguments must be enclosed within left angle (<) and right angle (>) brackets. Defaults are applied to the XAB$B_FLG field only if no FLG argument is specified. Consider the following:

```
KEY_1:  $XABKEY REF=1, POS=0, SIZ=10
```

This line specifies the key for alternate index 1. Therefore the macro defaults the XAB$B_FLG field to allow duplicates and changes (the default for alternate keys). However, if an FLG argument is explicitly specified, the results are different, as shown in the following code example.

```
KEY_2:  $XABKEY REF=1, POS=0, SIZ=10, FLG=CHG
```

In this case, the CHG bit is set in the XAB$B_FLG field and the DUP bit remains clear, to disallow duplicates on this key.

Depending on whether the key being defined is simple or segmented, you would use one of the following two formats for the POS and SIZ arguments:

```
POS=position
   .
   .
   .
SIZ=size
```

or

```
POS=<position0,...,position7>
   .
   .
   .
SIZ=<size0,...,size7>
```

You must include the angle brackets for multiple argument key positions and sizes.

$XABKEY_STORE

The $XABKEY_STORE macro moves user-specified values into fields of the specified XABKEY. The expanded $XABKEY_STORE code executes at run time on a previously initialized (allocated) XABKEY, in contrast to the $XABKEY macro, which initializes the XABKEY at assembly time. The $XABKEY_STORE macro must reside in a code program section.

Format

```
$XABKEY_STORE  XAB=xabkey-address,
               COLTBL=#collating-table-address,
               DAN=#data-bucket-area-number,
               DFL=#data-bucket-fill-size,
                     BN2
                     DBN2
                     BN4
                     DBN4
                     BN8
                     DBN8
                     IN2
                     DIN2
                     IN4
               DTP= { DIN4 } ,
                     IN8
                     DIN8
                     COL
                     DCOL
                     PAC
                     DPAC
                     STG
                     DSTG
                     <CHG DAT_NCMPR
               FLG=  DUP IDX_NCMPR
                     KEY_NCMPR NUL>,
                     #index-bucket-
               IAN=  area-number,
               IFL=#index-bucket-fill-size,
                     key-name-buffer-
               KNM=  address,
                     #lowest-level-index-
               LAN=  area-number,
               NUL=#null-key-value,
               NXT=next-xab-address,
               POS=<position,...>,
               PROLOG=#prolog-level,
               REF=#key-of-reference-value,
               SIZ=<size,...>
```

Arguments

For a description of the control block fields that correspond to the $XABKEY_STORE macro arguments, see Chapter 13.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabkey-address** is required for the $XABKEY_STORE macro and is not present for the $XABKEY macro. The POS and SIZ arguments can be either symbolic addresses or a list of up to eight values, where each value must be preceded by a number sign (#), and the entire list must be enclosed within left angle and right angle brackets (<#value,...,#value>). The number of POS and SIZ values must be equal. Alternatively, each POS and SIZ value can be specified as an argument, using the following form:

POS0 = #value, POS1 = #value, ..., POS7 = #value

SIZ0 = #value, SIZ1 = #value, ..., SIZ7 = #value

Note that R0 is usually used by the $XABKEY_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABPRO

The $XABPRO macro allocates and initializes a XABPRO. No value is returned for this assembly-time operation.

Format

```
$XABPRO  ACLBUF=ACL-buffer-address,
         ACLCTX=<ACL-context>,
         ACLSIZ=ACL-buffer-size,
         MTACC=magnetic-tape-accessibility,
         NXT=next-xab-address,
         PRO=<system, owner, group, world>,
         PROT_OPT=<PROPAGATE>,
         UIC=<group, member>
```

Arguments

For a description of the control block fields that correspond to the $XABPRO macro arguments, refer to Chapter 14.

Rules applicable to arguments are described in Appendix B.

For the MTACC (**magnetic tape accessibility**) argument, an ASCII radix indicator is required. For example, the letter Z is entered as the accessibility character with the following macro expression:

```
$XABPRO  MTACC=^A/Z/
```

In this example, the circumflex (^) followed by an uppercase A indicates that ASCII text follows. The two slashes (//) delimit the ASCII text. RMS converts all lowercase characters to uppercase. No other modification is made.

For the PRO argument, the angle brackets are required syntax, and each user class must be separated from the others by a comma. When you omit a class to use the default protection, you must retain the comma to indicate the omission, unless no other class follows.

To allow all system users read and write access, use the default file protection for the file owner (by omission), allow group users read access, and use the default for world users, you would specify <RW,,R>. You may specify all, some, or none of the access characters, and place multiple characters in any order, for each user class.

Here is a listing of the user classes together with the letters used to represent them:

- R—read access
- W—write access
- E—execute access
- D—delete access

The absence of a code specifies that the access associated with the code is denied to the user.

Users are granted the maximum number of access rights for each of the classes to which they belong.

For the UIC argument, the value for the group item must be in the range of 0 to 37777; the value for the member item must from 0 to 177777. Note that the maximum values (37777 and 177777) are reserved for Digital use only. The group number and member number must be enclosed within angle brackets, placed in the order <group,member>, and be separated by a comma. Each number is interpreted as an octal number.

$XABPRO_STORE

The $XABPRO_STORE macro moves user-specified values into fields of the specified XABPRO. The expanded $XABPRO_STORE code executes at run time on a previously initialized (allocated) XABPRO, in contrast to the $XABPRO macro, which initializes a XABPRO at assembly time. The $XABPRO_STORE macro must reside in a code program section.

Format

```
$XABPRO_STORE   XAB=xabpro-address,
                ACLBUF=ACL-buffer-address,
                ACLCTX=#<ACL-context>,
                ACLSIZ=#ACL-buffer-size,
                MTACC= #magnetic-tape-
                       accessibility,
                NXT=next-xab-address,
                PRO= <system, owner,
                     group, world>,
                PROT_OPT=<PROPAGATE>,
                UIC=#uic-value
```

Arguments

For a description of the control block fields that correspond to the $XABPRO_ STORE macro arguments, see Chapter 14.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabpro-address** is required for the $XABPRO_STORE macro and is not present for the $XABPRO macro. Also, the following arguments do not comply with the general rules:

- The PRO argument (file protection) can be either a symbolic address or a list of keyword values. If you specify a list of keywords, it must be enclosed within left angle (<) and right angle (>) brackets and the number sign (#) must be omitted; for example, PRO=<RWED,RWED,R,R>.
- The UIC argument (group,member) can be either a symbolic address or a list of two data values. If the data values are constants, they must be specified with an octal radix without a preceding number sign (#). This argument can be passed by its symbolic address or by using a VAX MACRO expression.

Note that R0 is usually used by the $XABPRO_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABRDT

The $XABRDT macro allocates and initializes a XABRDT. No value is returned for this assembly-time operation.

Format

$XABRDT   NXT=next-xab-address

Arguments

For a description of the control block fields that correspond to the $XABRDT macro argument, see Chapter 15.

Rules applicable to arguments are described in Appendix B.

$XABRDT_STORE

The $XABRDT_STORE macro moves user-specified values into fields of the specified XABRDT. The expanded $XABRDT_STORE code executes at run time on a previously initialized (allocated) XABRDT, in contrast to the $XABRDT macro, which initializes the XABRDT at assembly time. The $XABRDT_STORE macro must reside in a code program section.

Format

```
$XABRDT_STORE   XAB=xabrdt-address,
                RDT=#revision-date-time,
                RVN=#revision-number,
                NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABRDT_STORE macro arguments, see Chapter 15.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabrdt-address** is required for the $XABRDT_STORE macro and is not present for the $XABRDT macro. Also, the RDT argument **revision-date-time** and RVN argument **revision-number** are not present in the $XABRDT macro. The RDT argument **revision-date-time** is usually passed by its symbolic address. However, if a register is used to contain a value for the XAB$Q_RDT field, do not use R12, because two contiguous registers must be used to contain the value of this quadword field.

Note that R0 is usually used by the $XABRDT_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABSUM

The $XABSUM macro allocates and initializes a XABSUM. No value is returned for this assembly-time operation.

Format

$XABSUM   NXT=next-xab-address

Arguments

For a description of the control block fields that correspond to the $XABSUM macro argument, see Chapter 17.

Rules applicable to arguments are described in Appendix B.

$XABSUM_STORE

The $XABSUM_STORE macro moves user-specified values into fields of the specified XABSUM. The expanded $XABSUM_STORE code executes at run time on a previously initialized (allocated) XABSUM, in contrast to the $XABSUM macro, which initializes the XABSUM at assembly time. The $XABSUM_STORE macro must reside in a code program section.

Format

```
$XABSUM_STORE   XAB=xabsum-address,
                NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABSUM_STORE macro arguments, see Chapter 17.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabsum-address** is required for the $XABSUM_STORE macro and is not present for the $XABSUM macro.

Note that R0 may be used by the $XABSUM_STORE macro; thus, R0 is not preserved and does not contain a return status.

$XABTRM

The $XABTRM macro allocates and initializes a XABTRM. No value is returned for this assembly-time operation.

Format

```
$XABTRM   ITMLST=item-list-address,
          ITMLST_LEN=item-list-length,
          NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABTRM macro arguments, see Chapter 18.

Rules applicable to arguments are described in Appendix B.

$XABTRM_STORE

The $XABTRM_STORE macro moves user-specified values into fields of the specified XABTRM. The expanded $XABTRM_STORE code executes at run time on a previously initialized (allocated) XABTRM, in contrast to the $XABTRM macro, which initializes a XABTRM at assembly time. The $XABTRM_STORE macro must reside in a code program section.

Format

```
$XABTRM_STORE   XAB=xabtrm-address,
                ITMLST=item-list-address,
                ITMLST_LEN=#item-list-length,
                NXT=next-xab-address
```

Arguments

For a description of the control block fields that correspond to the $XABTRM_STORE macro arguments, see Chapter 18.

Arguments fall into several categories: values, addresses, keywords, and the address of the control block to receive the specified arguments. Rules applicable to these argument categories for the control block store macros are described in Appendix B.

The XAB argument **xabtrm-address** is required for the $XABTRM_STORE macro and is not present for the $XABTRM macro.

Note that R0 is usually used by the $XABTRM_STORE macro; thus, R0 is not preserved and does not contain a return status.

B
VAX MACRO Programming Information and Examples

This appendix provides VAX MACRO programmers with information about the four types of RMS macros used in VAX MACRO programming. It describes each of the four types of macros, the macro naming conventions, and macro syntax rules. It then shows you how to use the macros and includes examples for each of the four types. Note that the term ‘‘macro’’ refers to a program macro written in the VAX MACRO language.

B.1 RMS Macros

RMS provides four types of macros used by VAX MACRO programs implementing RMS features. The functions these macros provide are described in the following list.

- Control block initialization macros initialize a control block at assembly time. This type of macro performs five separate actions:
  - Allocates space within the program image for the specified control block
  - Defines the symbolic names associated with a control block
  - Initializes certain control block fields with internally used values
  - Initializes specified control block fields with user-specified values
  - Initializes certain fields with system-supplied defaults (does not apply to all control block macros)

  As an alternative to using this type of macro, an application program would have to allocate each control block needed with its correct length, initialize the internally used fields with the correct values, and initialize or set user-specified values in the appropriate fields. It is strongly recommended that you use these macros for VAX MACRO programs.
- Control block symbol definition macros define control block symbolic names at assembly time without allocating and initializing the control block. These macros are needed only when the corresponding initialization macro is not used and the symbols are not defined.
- Control block store macros store user-specified field values in control blocks at run time. Alternatively, you can store values in control block fields at run time using VAX MACRO instructions, such as the MOVx and MOVAx instructions. RMS accesses control block fields using the symbolic names that represent field offsets from the start of the control block.
- Service macros invoke record management services at run time. When a service is invoked, one or more control blocks are examined for required field values. Values are also returned in one or more control blocks, including condition codes. Record management services conform to the OpenVMS

calling standard and can be invoked directly, if needed, without the calling program having to use the service macro. However, the appropriate control block must be present with the appropriate field values set for the requested operation.

RMS stores its macros for use by VAX MACRO programs in SYS$LIBRARY:STARLET.MLB.

B.1.1 Conventions for Naming RMS Macros

The names for the macros that initialize control blocks at assembly time consist of a dollar sign ($) followed by the name of the control block. For example, the name of the macro that initializes a FAB is $FAB; the name of the macro that initializes a XABALL is $XABALL.

The names for the macros that define symbolic offsets without performing control block initialization contain the suffix DEF following the corresponding initialization macro name; for example, $FABDEF and $XABALLDEF.

The names for the macros that set control block field values at run time contain the suffix ''_STORE'' following the corresponding initialization macro name; for example, $FAB_STORE and $XABALL_STORE.

Table B–1 summarizes the control blocks, their assembly time macro names, and their functions.

Table B–1 User Control Blocks

Control Block	Macro Names	Function
FAB		Describes a file and contains file-related information
	$FAB	Allocates storage for a FAB and initializes certain FAB fields; also defines symbolic offsets for a FAB
	$FAB_STORE	Moves specified values into a previously allocated and initialized FAB
NAM		Contains file specification information beyond that in the file access block
	$NAM	Allocates storage for a NAM and initializes certain NAM fields; also defines symbolic offsets for a NAM
	$NAM_STORE	Moves specified values into a previously specified and allocated NAM
RAB		Describes a record stream and contains record-related information
	$RAB	Allocates storage for a RAB and initializes certain RAB fields; also defines symbolic offsets for a RAB
	$RAB_STORE	Moves specified values into a previously specified and allocated RAB
	$RAB64	Allocates storage for a RAB64 and initializes certain RAB64 fields; also defines symbolic offsets for a RAB64

(continued on next page)

Table B–1 (Cont.) User Control Blocks

Control Block	Macro Names	Function
	$RAB64_STORE	Moves specified values into a previously specified and allocated RAB64
XAB		Supplements file attributes in the file access block; for XABTRM and XABRU, supplements information in the record access block
	$XABxxx[1]	Allocates and initializes a XAB
	$XABxxx_STORE	Moves specified values into a previously specified and allocated XABxxx

[1]The variable *xxx* represents a 3-character mnemonic.

Record management services can be called using the OpenVMS calling standard. These services are system services identified by the entry point prefix ‘‘SYS$’’ followed by three or more characters; the ‘‘SYS’’ prefix is omitted in the corresponding VAX MACRO macro name. For example, the Create service has an entry point of SYS$CREATE and a VAX MACRO macro name of $CREATE. A complete description of each service is provided in Part III.

Table B–2 describes the functions of each service, including the service entry point name and its corresponding VAX MACRO macro name.

Table B–2 Record Management Services

Service Name	Macro Name	Description
File-Processing and File-Naming Macros		
SYS$CLOSE	$CLOSE	Terminates file processing and disconnects all record streams previously associated with the file
SYS$CREATE	$CREATE	Creates and opens a new file
SYS$DISPLAY	$DISPLAY	Returns the opened file’s attributes to the application program
SYS$ENTER[1]	$ENTER	Enters a file name into a directory
SYS$ERASE	$ERASE	Deletes a file and removes its directory entry
SYS$EXTEND	$EXTEND	Allocates additional space to a file
SYS$OPEN	$OPEN	Opens an existing file and initiates file processing
SYS$PARSE	$PARSE	Parses a file specification
SYS$REMOVE[1]	$REMOVE	Removes a file name from a directory but does not actually delete the file data; compare this with the $ERASE macro
SYS$RENAME	$RENAME	Assigns a new name to (renames) a file

[1]This service is not supported for DECnet for OpenVMS operations involving remote file access between two OpenVMS systems.

(continued on next page)

Table B–2 (Cont.) Record Management Services

Service Name	Macro Name	Description
File-Processing and File-Naming Macros		
SYS$SEARCH	$SEARCH	Searches a directory, or possibly multiple directories, for a file name
Record-Processing Macros		
SYS$CONNECT	$CONNECT	Establishes a record stream by associating a RAB with an open file
SYS$DELETE	$DELETE	Deletes a record from a relative or indexed file
SYS$DISCONNECT	$DISCONNECT	Terminates a record stream by disconnecting a RAB from an open file
SYS$FIND	$FIND	Locates the specified record, establishes it as the current record, and returns the record’s RFA to the application program
SYS$FLUSH	$FLUSH	Writes (flushes) modified I/O buffers and file attributes to the disk before closing a file
SYS$FREE	$FREE	Unlocks all records previously locked by the record stream
SYS$GET	$GET	Retrieves a record from a file
SYS$NXTVOL[1]	$NXTVOL	Causes processing of a magnetic tape file to continue to the next volume of a volume set
SYS$PUT	$PUT	Writes a new record to a file
SYS$RELEASE	$RELEASE	Unlocks a record pointed to by the contents of the RAB$W_RFA field
SYS$REWIND	$REWIND	Establishes the first file record as the current record
SYS$TRUNCATE	$TRUNCATE	Truncates a sequential file
SYS$UPDATE	$UPDATE	Deletes and rewrites (updates) an existing file record
SYS$WAIT	$WAIT	Awaits the completion of an asynchronous record operation
Block I/O-Processing Macros		
SYS$READ	$READ	Retrieves a specified number of bytes from a file, beginning on block boundaries
SYS$SPACE	$SPACE	Positions forward or backward in a file to a block boundary
SYS$WRITE	$WRITE	Writes a specified number of bytes to a file, beginning on block boundaries

[1]This service is not supported for DECnet for OpenVMS operations involving remote file access between two OpenVMS systems.

B.1.2 Applicable VAX MACRO Syntax Rules

One of the conventions used with RMS control block macros is to identify a field by a mnemonic; for example, you can specify the FAB$L_ALQ field using its mnemonic, ALQ. Using a mnemonic ensures the accuracy of argument value placement regardless of how you code the related argument. For example, the mnemonic for the FAB field that specifies the allocation quantity is ALQ. Thus, when using the $FAB macro to initialize the allocation quantity field, you might use the following macro expression:

```
INFAB:  $FAB  ALQ=500
```

This macro statement defines the start of the FAB at label (symbolic address) INFAB and initializes the allocation field to provide 500 blocks of space to the specified file.

In this instance, if you want to change the allocation value to 250 blocks at run time, you could use the following macro expression:

```
MOVL   #250, INFAB+FAB$L_ALQ    ; Set allocation quantity
```

In fields that contain binary options or keyword values, you should use the appropriate keyword or symbolic binary option value. For example, the FAB uses the ORG field to specify a file organization. Three keywords are defined for this field: SEQ (sequential file), REL (relative file), and IDX (indexed file). To specify an indexed file organization, you should use the following macro expression:

```
OUTFAB: $FAB  ORG=IDX
```

To specify an indexed file organization at run time, you must move the value into the field using appropriate symbols:

```
MOVAL OUTFAB, R5                  ; Move address into R5
MOVB  #FAB$C_IDX, FAB$B_ORG(R5)   ; Store constant value
```

In control block macros, arguments for bit fields that can contain multiple values are usually enclosed within angle brackets <value_1,value_2,value_n>. Consider the file access (FAC) field (FAB$B_FAC) in the FAB, a bit field that can contain multiple values. To permit a process to do Get and Put operations, the following macro expression could be used:

```
INFAB:  $FAB  FAC=<GET,PUT>   ; Specify Put and Get operations
```

Control block macro arguments that are interpreted as ASCII characters (such as a file specification) must also be enclosed within angle brackets. The use of the left angle (<) and right angle (>) delimiters is noted in the format and argument descriptions of the control block macros in Part II.

At run time, you could use the following code sequence to make the file accessible to a Get operation:

```
MOVAL OUTFAB, R6                    ; Move FAB addr into R6
BBS   #FAB$V_GET, FAB$B_FAC(R6), OK    ; Go to OK if GET bit set
BISB  #FAB$M_GET, FAB$B_FAC(R6)        ; Else set GET bit
OK:
```

When you use RMS macros, follow the coding rules used by the VAX MACRO assembler as described in the following list:

- Comments must be separated from the rest of the code line by a semicolon (;). For example:

  ```
  $FAB   BKS=4              ; Bucket size
  ```

- All the arguments for a macro must be coded in a single statement. If the arguments do not fit on a line or if you want to use multiple lines, type the continuation character, a hyphen (-), as the last character on the line; then continue typing arguments on the next line. Comments can follow the hyphen. For example:

```
$FAB  FNA=FLNAM,-            ; Filename address
      ALQ=100,-              ; Allocation quantity
      BKS=4                  ; Bucket size
```

- Arguments and subarguments can be separated from each other by one of the following:
  - A comma, with or without spaces or tabs:

    ```
    FNA=FLNAM,ALQ=100
    ```

  - A space:

    ```
    FNA=FLNAM ALQ=100
    ```

  - Multiple spaces or tabs:

    ```
    FNA=FLNAM        ALQ=100
    ```

  The comma without a space or tab is preferred. All coding examples in this manual use a comma to separate arguments.

B.2 Using the RMS Macros

This section provides examples of how to use the four types of RMS macros.

B.2.1 Control Block Initialization Macros

A major advantage to using the control block initialization macros is that they direct the initialization values to the correct field locations in the control block. Returned status values do not apply here because RMS evaluates this type of macro at assembly time, not at run time.

Control block initialization macros are located at the beginning of their associated control block. The initialization macro should have a related label because the address of the control block is a required argument for most services and for some control block macros. As shown in the following example, the label provides a programming convenience for symbolically addressing the control block:

```
MYFAB:  $FAB
```

Arguments usually require an address, such as a symbolic address, or a value. RMS initializes the appropriate field by simply taking the supplied argument and placing it after the appropriate macro data declaration directive, as shown in the following code example:

```
.ADDRESS address
.BYTE value
```

Arguments that specify field values must be enclosed within angle brackets under the following conditions:

- When the argument is a file specification (ASCII character string)
- When more than one argument is supplied for a binary options field, where each bit option is identified by a 3-letter mnemonic

- Where otherwise indicated in the format of the macro, such as for UICs and file identifiers in which multiple values separated by commas constitute the argument

Here are several examples:

```
MYFAB:  $FAB     FAC=<GET, PUT>,-         ; Multioption field
                 FNM=<DEGREE_DAY.DAT>,-   ; File specification
   .
   .
   .
                 NXT=MYXPRO,-             ; XAB address
                 ORG=SEQ                  ; Single-option field
;
MYXPRO: $XABPRO  PRO=<RWED,RWED,R,R>,-    ; File protection
                 UIC=<377,377>            ; UIC
```

The complete format of each control block initialization macro is provided in Appendix A.

Do not position the macro name in a read-only program section because control block fields may receive values during the execution of a service. For efficiency, align the control blocks on a longword boundary. The initialization macros display an informational message in the listing file if the control block is not longword aligned.

In summary, initialization macros must be placed in a writable data program section in which the data has been aligned on a longword boundary.

B.2.2 Control Block Symbol Definition Macros

A control block symbol definition macro includes the macro name only, has no arguments, and can be placed in any program section. The macro name is made up of the associated initialization macro and the suffix DEF.

RMS does not provide return values for control block symbol definition macros.

B.2.3 Control Block Store Macros

A control block store macro consists of executable run-time code, so it must be placed within an executable code program section. R0 cannot be used to return condition codes because control block store macros may use R0 to move arguments. The only detectable errors are assembly-time errors.

The calling format of each control block store macro resembles the calling format of the corresponding initialization macro except that a control block store macro can take a run-time value as an argument. Run-time values include date and time values, file identifier values, device identifier values, directory identifier values, and record file address values.

The following list describes other differences between the format of a control block store macro and its corresponding control block initialization macro:

- An argument specifying the address of the control block is required unless the calling program provides the control block address in R0. If this argument is not a register value in the form R*n*, RMS loads the control block address into R0.
- For each argument that requires an address, the store macro uses the VAX MACRO instruction MOVAx (usually MOVAL) to move the address into the appropriate control block field. Thus, VAX MACRO expressions can be used. For instance, you can use a symbolic address to specify the control

block address argument directly (for example, FAB=MYFAB, NAM=MYNAM, RAB=MYRAB, or XAB=MYXAB).

You may also specify a register that contains the address using the form Rn, where *n* is a decimal number from 0 through 12.

- For each argument that requires a nonkeyword data value, the store macro uses the VAX MACRO instruction MOVx to move data into the appropriate control block field. Thus, VAX MACRO expressions can be used. Note that a number sign (#) must precede a literal nonkeyword value, except when a literal value is enclosed within left angle (<) and right angle (>) brackets. However, if you specify the address where the argument value resides, a number sign must not precede the symbolic address nor the register expression that contains the address.
- For binary option or keyword value fields, use the supplied keyword without a number sign and do not use a VAX MACRO expression. Multiple keyword arguments must be enclosed within left angle (<) and right angle (>) brackets.

In some cases, arguments are specified as run-time values using one of the following forms:

- A VAX MACRO expression
- The symbolic address of the argument value

The format for each store macro is provided in Appendix A, which includes exceptions to the general rules previously described.

Example B–1 illustrates the use of the $XABDAT_STORE macro to set the creation date of the file to the beginning of a fiscal quarter, thereby establishing a valid starting date for the file data.

This short program creates a file with a creation date of midnight, August 9, 1994. The FAB at symbolic address MYFAB defines a sequential file with variable-length records up to 52 bytes in length and specifies an allocation size of 500 blocks using the contiguous-best-try file processing option. It also specifies the file specification. The .ASCID assembler directive defines the constant date-time character string at symbolic address ATIM.

The SYS$BINTIM system service is invoked to convert the constant ASCII time at symbolic address ATIM to binary format in the quadword at BTIM. The BTIM value is moved into the XAB$Q_CDT field of the XABDAT control block at symbolic address MYXDAT using the following XABDAT_STORE macro:

```
$XABDAT_STORE XAB=MYXDAT, CDT=BTIM
```

Because the creation date in field XAB$Q_CDT is input to the Create service ($CREATE macro), the value must be stored before the program invokes the Create service. The file is created, then closed. Note that Create service errors are signaled using the FAB$L_STS and FAB$L_STV fields, not R0.

Example B–1 Use of the $XABDAT and $XABDAT_STORE Macros

```
.TITLE  CREAT  -  SET CREATION DATE
;       Program that uses XABDAT and XABDAT_STORE
;
        .PSECT LONG  WRT,NOEXE
;
MYFAB: $FAB  ALQ=500,FOP=CBT,FAC=PUT, -
             FNM=<DISK$:[PROGRAM]SAMPLE_FILE.DAT>,-
             ORG=SEQ,RAT=CR,RFM=VAR,SHR=NIL,MRS=52,XAB=MYXDAT
;
MYXDAT: $XABDAT
;
ATIM:   .ASCID  /9-AUG-1994 00:00:00.00/
BTIM:   .BLKQ   1
;
        .PSECT  CODE NOWRT,EXE
        .ENTRY  CREAT, ^M<>
START:  $BINTIM_S TIMBUF=ATIM, TIMADR=BTIM   ; Convert ASCII to binary time
        BLBC    R0,SS_ERR                    ; Branch on error
        $XABDAT_STORE XAB=MYXDAT, CDT=BTIM   ; Move time into XAB$Q_CDT
        $CREATE FAB=MYFAB                    ; Create file; populate
        BLBC    R0,ERROR                     ; file later
CLOSE:  $CLOSE  FAB=MYFAB                    ; Close file
        BRB     FINI                         ; and exit
ERROR:  PUSHL   FAB$L_STV+MYFAB              ; Push FAB STS and
        PUSHL   FAB$L_STS+MYFAB              ; STV on stack
        CALLS   #2, G^LIB$SIGNAL             ; Signal error
        BRB     FINI
SS_ERR: PUSHL   R0                           ; Push R0
        CALLS   #1, G^LIB$SIGNAL             ; Signal error
FINI:   RET
        .END CREAT
```

B.2.4 Service Macros

This section describes the general macro format of record management service macros. Part III describes each service in detail, including the calling format.

Note that the general information applicable to invoking record management services in Chapter 2 also applies to programs written in VAX MACRO.

The service macros use two general formats:

```
label: macro-name
```

```
label: macro-name   RAB=rab-address,-
                        ERR=entry,-
                        SUC=entry
```

The first format takes no arguments. You supply the argument list within your program, and the argument pointer register (AP) is assumed to contain the address of the argument list. An example of this format follows.

```
ARG_LOC: .BLKL 2
   .
   .
   .
MOVL   #1,ARG_LOC            ; Move number of args to ARG_LOC
MOVAL  INFAB, ARG_LOC+4      ; Move FAB address to ARG_LOC+4
MOVAL  ARG_LOC, AP           ; Move ARG_LOC address to AP
$OPEN                        ; Open file
```

In this form, the $OPEN macro expands to the following VAX MACRO code:

```
CALLG  (AP), G^SYS$OPEN
```

In the second format, you supply arguments that automatically generate an argument list on the stack according to the values you supplied. You specify these arguments using keywords, which can be in any order. You must separate keywords using a comma, a blank space, or tabs. The only argument required when using the second format is the control block address (FAB=fab-address or RAB=rab-address). This argument must be either a general register (R0 through R11) containing the control block address, or a suitable address for a PUSHAL instruction. If you omit this argument, no other arguments are allowed; in other words, you must use the first format.

The ERR=entry and SUC=entry arguments are optional and, if used, provide the addresses of completion routine entry points. Completion routines are always executed as ASTs. RMS places the values you supply in the argument list on the stack during execution of the expanded macro. These values must be addresses that can be used by a PUSHAL instruction.

Here is an example of the second format:

```
$OPEN FAB=INFAB
```

This macro line expands to the following VAX MACRO code:

```
PUSHAL  INFAB
CALLS   #01, G^SYS$OPEN
```

When the argument list contains a completion routine argument, an AST is queued. When the AST routine executes, the following conditions hold:

- General registers R0 through R11 are undefined. The AP contains the address of the AST argument list; the AST argument value in the AST argument list specifies the address of the associated control block (FAB or RAB). The status must be retrieved from the completion status code field (STS) of the associated control block.
- Any general registers saved by an entry mask can be modified, in addition to R0 and R1.
- Additional calls to record management services can be made within the completion routines.
- To exit from a completion routine, you must perform any necessary cleanup operations and execute a RET instruction.

The calling format of each service is listed alphabetically in Part III. The format for the Close service is shown in the following code example:

```
SYS$CLOSE   fab  [,[err]  [,suc]]
```

When you use a macro to call a service, remember to omit the SYS prefix. For example, use $CLOSE instead of SYS$CLOSE.

All file-processing macros require the FAB address as an argument and optionally allow you to specify the entry points for error or success condition handlers, as shown in the following format illustration:

```
$macro  FAB=fab-addr  [,ERR=error-entry]  [,SUC=success-entry]
```

For example, to invoke the $OPEN macro and pass it the FAB address of INFAB and the error entry point of OPEN_ERR, you could use the following macro:

```
$OPEN   FAB=INFAB, ERR=OPEN_ERR
```

Note that the $RENAME macro has a different format, as noted in Table B–3. This file processing macro has the following format:

```
$RENAME  OLDFAB=old-fab-addr [,ERR=error-entry]
         [,SUC=success-entry] ,NEWFAB=new-fab-addr
```

The format for record processing macros and block I/O macros requires the RAB address as an argument and optionally allows you to specify the entry points for error or success condition handlers, as shown in the following format illustration:

```
$macro  RAB=rab-addr  [,ERR=error-entry]  [,SUC=success-entry]
```

Note that the $WAIT macro has a different format, in that it does not use the error and success arguments:

```
$WAIT   RAB=rab-addr
```

Table B–3 lists each service macro according to its macro type.

Table B–3 File, Record, and Block I/O Processing Macros

File Processing	Record Processing	Block I/O
$CLOSE	$CONNECT	$READ
$CREATE	$DELETE	$SPACE
$DISPLAY	$DISCONNECT	$WRITE
$ENTER	$FIND	
$ERASE	$FLUSH	
$EXTEND	$FREE	
$NXTVOL	$GET	
$OPEN	$PUT	
$PARSE	$RELEASE	
$REMOVE	$REWIND	
$RENAME[1]	$TRUNCATE	
$SEARCH	$UPDATE	
	$WAIT	

[1]Denotes macro with nonstandard format (see text).

After calling a service, you should check the status code returned in R0 (and the STS field of the appropriate control block). The recommended way to signal errors is to provide both the STS and STV fields of the FAB or RAB as arguments to the appropriate run-time library routine. The following VAX MACRO instructions invoke the LIB$SIGNAL routine for a file-related (FAB) error using the CALLS (stack) form of calling a routine, where the FAB is located at symbolic address MYFAB (not shown):

```
PUSHL  MYFAB+FAB$L_STV         ; Push fields on stack
PUSHL  MYFAB+FAB$L_STS         ; in reverse order
CALLS  #2, G^LIB$SIGNAL        ; Invoke signal routine
```

B.3 VAX MACRO Example Programs

This section includes examples illustrating the implementation of RMS at the VAX MACRO programming level. See the *Guide to OpenVMS File Applications* for RMS examples using the Edit/FDL utility.

Using RMS macros, you can create new files, process existing files, extend and delete files, and read, write, update, and delete records within files.

To create and process RMS files, your program must contain calls to appropriate services. Generally, you make these calls by using the service macros for run-time processing. When encountered at run time, the expanded code of these macros generates a call to the corresponding service. Each macro and its resultant call represent a program request for a file or record service, or a block I/O transfer operation.

B.3.1 Creating, Accessing, and Deaccessing a File

The Create service constructs a new file according to the attributes you specify in the FAB for the file, whereas the Open service makes an existing file available for processing by your program. Both of these services, invoked by the $CREATE and $OPEN macros respectively, allocate resources within the system to establish access (a path) to a file. You must open or create a file to perform most file operations and any record operations on that file. Applications designed for shared access must declare the type of sharing at this time. The user specifies the various types of shared access by setting bits in the file access control (FAB$B_FAC) and share (FAB$B_SHR) fields in the appropriate FAB.

RMS provides several file-processing options for the Create service. The create-if option (FAB$V_CIF option in the FAB$L_FOP field) requests that the file be created only if it does not exist. If the file does exist in the specified directory, the file is opened, not created. The Open and Create services both establish access to the desired file, but the Create service additionally allocates disk space and performs functions related to allocation.

When you are finished processing a file, you invoke the Close service ($CLOSE macro) to close the file, disconnect all record streams associated with the file, and free all resources allocated to the file. If you do not explicitly invoke the Close service when the program image exits, RMS attempts an implicit close. All resources associated with open files are returned when the files are deaccessed at image rundown time. However, process permanent files are not implicitly closed when an image exits. These are special files that the current CLI opens outside the context of a normal image.

B.3.2 Example of Opening and Creating Files

Example B–2 illustrates the use of the Open, Create, Connect, Get, Put, and Close services to access and copy records from one file to another. Note that the arguments to the $FAB and $RAB macros are listed vertically on separate lines for ease in reading them. However, the argument list must be contiguous and a common programming error is omission of required delimiters and continuation characters when the arguments are listed in this manner.

Example B–2 Use of the Create, Open, and Close Services

```
        .TITLE  COPYFILE
;
; This program copies the input file to the output file.
;
        .PSECT  DATA,WRT,NOEXE
INFAB:  $FAB    FNM = <INFILE:>,-       ; Primary input file name
                DNM = <.INV>            ; Default input file type
INRAB:  $RAB    FAB = INFAB,-           ; Pointer to FAB
                ROP = RAH,-             ; Read-ahead option
                UBF = REC_BUFF,-        ; Record buffer
                USZ = REC_SIZE          ; and size
OUTFAB: $FAB    FNM = <OUTFILE:>,-      ; Primary output file name
                DNM = <.INV>,-          ; Default output file name
                FOP = CTG,-             ; Make contiguous file
                FAC = <PUT>,-           ; Open for PUT operations
                SHR = <NIL>,-           ; Exclusive file access
                MRS = REC_SIZE,-        ; Maximum record size
                RAT = CR                ; Implied carriage control

OUTRAB: $RAB    FAB = OUTFAB,-          ; Pointer to FAB
                ROP = WBH,-             ; Write-behind option
                RBF = REC_BUFF          ; Output uses same buffer
                                        ; as input

;
REC_SIZE = 132                          ; Maximum record size
REC_BUFF:
        .BLKB   REC_SIZE                ; Record buffer

        .PSECT  CODE,NOWRT,EXE
;
; Initialization - Open input and output files and connect streams
;
        .ENTRY  COPYFILE,^M<R6>         ; Save R6
        $OPEN   FAB=INFAB               ; Open input file
        BLBC    R0,EXIT1                ; Quit on error
        $CONNECT   RAB=INRAB            ; Connect to input
        BLBC    R0,EXIT2                ; Quit on error
        MOVL    INFAB+FAB$L_ALQ,-       ; Set proper size for output
                OUTFAB+FAB$L_ALQ
        $CREATE FAB=OUTFAB              ; Create output file
        BLBC    R0,EXIT3                ; Quit on error
        $CONNECT        RAB=OUTRAB      ; Connect to output
        BLBS    R0,READ                 ; Branch to READ loop
        BRB     EXIT4                   ; Trap error
EXIT1:  MOVAL   INFAB,R6                ; Error: Keep FAB address
        BRB     F_ERR                   ; Signal file error
EXIT2:  MOVAL   INRAB,R6                ; Keep RAB address
        BRB     R_ERR                   ; Signal record error
EXIT3:  MOVAL   OUTFAB,R6               ; Keep FAB address
        BRB     F_ERR                   ; Signal record error
EXIT4:  MOVAL   OUTRAB,R6               ; If error, retain RAB addr.
        BRB     R_ERR                   ; Signal record error
;
; Copy records loop
;
READ:   $GET    RAB=INRAB               ; Get a record
        BLBS    R0,WRITE                ; Write the record
        CMPL    R0,#RMS$_EOF            ; Was error end-of-file?
        BEQL    DONE                    ; Successful completion
        BRB     EXIT2                   ; Error otherwise
```

(continued on next page)

Example B–2 (Cont.) Use of the Create, Open, and Close Services

```
WRITE:  MOVW    INRAB+RAB$W_RSZ, -      ; Input RAB sets record
                OUTRAB+RAB$W_RSZ        ; size for output RAB
        $PUT    RAB=OUTRAB              ; Write the record
        BLBC    R0,EXIT4                ; Quit on error
        BRB     READ                    ; Go back for more
;
; Close files, signal any errors, and exit
;
F_ERR:  PUSHL   FAB$L_STV(R6)           ; Push STV and STS of FAB
        PUSHL   FAB$L_STS(R6)           ; on the stack
        CALLS   #2, G^LIB$SIGNAL        ; Signal error
        BRB     EXIT
R_ERR:  PUSHL   RAB$L_STV(R6)           ; Push STV and STS of RAB
        PUSHL   RAB$L_STS(R6)           ; on the stack
        CALLS   #2, G^LIB$SIGNAL        ; Signal error
DONE:   $CLOSE  FAB=INFAB               ; Close input
        $CLOSE  FAB=OUTFAB              ; and output
EXIT:   RET                             ; Return with status in R0
        .END    COPYFILE
```

This example illustrates how you can use the sequential file organization to create a new file by copying records from an existing file. The newly created file and the source file have variable-length records.

This example assumes that an external program has identified the input file as a search list logical name using this statement:

```
$ ASSIGN [INV]30JUN85,[INV.OLD]30JUN85 INFILE:
```

This directs RMS to look for the input file in directory [INV] first, and if it does not find the file, it should look in directory [INV.OLD].

The program also specifies the default file type .INV for the input file using this statement:

```
DNM=<.INV>              ; Default input file name
```

Next the program configures the RAB used for the input file (labeled INRAB). The first argument links the RAB to the associated FAB (INFAB) and this is the only required argument to a RAB. The rest of the arguments specify the read-ahead option (described in later text) and the record buffer for the input file. The Get service uses the user record buffer address (UBF) field and the user record buffer size (USZ) field as inputs to specify the record buffer and the record size, respectively.

Note

When you invoke the Get service, RMS takes control of the record buffer and may modify it. RMS returns the record size and only guarantees the contents from where it accessed the record to the completion of the record.

The program then configures the FAB for the output file. The first argument uses the FNM field to equate the file name to the externally defined logical name OUTFILE. After the program specifies the default file specification extension for the output file, it specifies three additional FAB fields.

First it directs RMS to allocate contiguous space for the output file by setting the CTG bit in the FAB$L_FOP field of the FAB.

Next the program uses a program-defined variable to store the value 132 in the MRS field:

```
MRS=REC_SIZE
REC_SIZE= 132
```

The program then specifies that each record is to be preceded by a line feed and followed by a carriage return whenever the record is output to a line printer or terminal:

```
RAT=CR
```

Because the program alternately reads and then writes each record, the input file and the output file may share the same buffer. However, because the Put service does not have access to the UBF and UBZ fields, the output RAB defines the buffer using the RBF and the RSZ fields.

Note that the UBF, USZ, and RBF values are set prior to run time, but that the RSZ value is set at run time, just prior to invocation of the Put service. This is done because the input file contains variable-length records and the Put service relies on the Get service to supply each record's size by way of the RSZ field, an INRAB output field.

The following statement from the sample program illustrates this feature:

```
WRITE:  MOVW    INRAB+RAB$W_RSZ, -      ; Input RAB sets record
                OUTRAB+RAB$W_RSZ        ; size for output RAB
```

The run-time processing macros for the input file consist of a $OPEN, a $CONNECT, a $GET, and a $CLOSE macro. Because the input file already exists, the program accesses it with a $OPEN macro. The sole argument to this macro identifies the FAB to the Open service:

```
$OPEN  FAB=INFAB
```

Next, the program connects a record stream to the input file by calling the Connect service and specifying INRAB as the appropriate RAB:

```
$CONNECT  RAB=INRAB
```

Note that upon completion of each service call, the program tests the condition value in R0 returned by the service before proceeding to the next call. If the call fails, the program exits with the appropriate control block address in R6.

After creating the output file and establishing its record stream, the program begins a processing loop in which the Get service reads a record from the input file and the Put service writes the record to the output file. When all file records are copied, as indicated by the detection of the end of the file, the program exits to label DONE, which closes both files.

The Close service disconnects the record stream for all RABs connected to the specified FAB. In a multistream environment (more than one RAB can be connected to a single FAB), a program may disconnect individual record streams using the Disconnect service.

B.3.3 Example of Creating a Multiple-Key Indexed File

Example B–3 creates an indexed file on a remote node from a sequential file on the local node. The indexed file contains three keys: a segmented primary key and two simple alternate keys. The segmented primary key includes the customer's last name, the first letter of the customer's first name, and the customer's middle initial.

Example B–3 Use of the Create Service for an Indexed File

```
        .TITLE  CREATEIDX - CREATE INDEXED FILE
        .IDENT  /V001/
;
; This program creates an indexed file with three keys from a
; sequential file containing a name and address list. The record
; format of the input file is shown below:
;
;       First Name      Column  00-10
;       Middle Initial  Column  11-11
;       Last Name       Column  12-26
;       Street          Column  27-46
;       City            Column  47-58
;       State           Column  59-60
;       Zip Code        Column  61-65
;       Reserved for
;         new data      Column  66-end of record
;
; The input and output files are specified by the logical names SRC
; and DST, respectively. For example:
;
;       $ DEFINE SRC DBB1:[TEST]INPUT.DAT
;       $ DEFINE DST TRNTO::DRA4:[RMS.FILES]OUTPUT.DAT
;       $ RUN CREATEIDX
;
;*********************************************************************
        .SBTTL  Control block and buffer storage
        .PSECT  DATA            NOEXE,LONG
;
; Define the source file FAB and RAB control blocks.
;
SRC_FAB:
        $FAB    FAC=<GET>,-             ; File access for GET only
                FOP=<SQO>,-             ; DAP file transfer mode

                FNM=<SRC:>              ; Name of input file
SRC_RAB:
        $RAB    FAB=SRC_FAB,-           ; Address of associated FAB
                RAC=SEQ,-               ; Sequential record access
                UBF=BUFFER,-            ; Buffer address
                USZ=BUFFER_SIZE         ; Buffer size
```

(continued on next page)

Example B–3 (Cont.) Use of the Create Service for an Indexed File

```
;
; Define the destination file FAB and RAB control blocks.
;
DST_FAB:
        $FAB    FAC=<PUT>,-             ; File access for PUT only
                FOP=CTG,-               ; Allocate contiguous
                SHR = <NIL>,-           ; Exclusive file access
                FNM=<DST:>,-            ; Name of output file
                MRS=128,-               ; Maximum record size
                RFM=VAR,-               ; Variable length records
                RAT=<CR>,-              ; Implied carriage control
                ORG=IDX,-               ; Indexed file organization
                XAB=DST_KEY0            ; Address of start of XAB chain
DST_RAB:
        $RAB    FAB=DST_FAB,-           ; Address of associated FAB
                MBF=3,-                 ; Use 3 buffers
                RAC=KEY,-               ; Random record writes
                RBF=BUFFER,-            ; Buffer address
                ROP=LOA,-               ; Specify initial fill size
                RSZ=BUFFER_SIZE         ; Buffer size
;
; Define a key definition XAB for the primary key.
;
DST_KEY0:                               ; Primary key is Name
        $XABKEY REF=0,-                 ; Key reference number
                DAN=0,-                 ; Define data XABALL
                DFL=1536,-              ; Define data fill of 75%
                FLG=<DUP>,-             ; Allow duplicate keys
                DTP=DSTG,-              ; Decending sort order
                IAN=1,-                 ; Define index XABALL
                IFL=1536,-              ; Initial index fill 75%
                PROLOG=3,-              ; Request prolog 3
                POS=<12,0,11>,-         ; Key segment positions
                SIZ=<15,1,1>,-          ; Key segment lengths
                NXT=DST_KEY1            ; Address of next XAB in chain
;
; Define key definition XABs for the alternate keys.
;
DST_KEY1:                               ; 1st alternate key is City
        $XABKEY REF=1,-                 ; Key reference number
                DAN=2,-                 ; Data level (SIDR) XABALL
                IAN=2,-                 ; Index XABALL
                IFL=768,-               ; Initial index fill 75%
                POS=47,-                ; Starting key position
                SIZ=12,-                ; Key size
                FLG=<CHG,DUP>,-         ; Duplicates and changes
                NXT=DST_KEY2            ; Address of next XAB in chain
DST_KEY2:                               ; 2nd alternate key is State
        $XABKEY REF=2,-                 ; Key reference number
                DAN=2,-                 ; Data level (SIDR) XABALL
                IAN=2,-                 ; Index XABALL
                IFL=768,-               ; Initial index fill 75%
                POS=59,-                ; Starting key position
                FLG=<CHG,DUP>,-         ; Duplicates and changes
                SIZ=2,-                 ; Key size
                NXT=DST_ALL0            ; Designate next XAB
```

(continued on next page)

Example B–3 (Cont.) Use of the Create Service for an Indexed File

```
;
;  Define allocation control XABs to define multiple areas
;
DST_ALL0:
        $XABALL AID=0,-                 ; Data area definition
                ALQ=328,-               ; Allocation quantity and
                AOP=<CBT>,-             ; contiguous best try
                BKZ=4,-                 ; Bucket size of 4 blocks
                DEQ=112,-               ; Default extension quantity
                NXT=DST_ALL1            ; Designate next XAB
DST_ALL1:
        $XABALL AID=1,-                 ; Primary key index area
                ALQ=8,-                 ; Allocation quantity and
                AOP=<CBT>,-             ; contiguous best try
                BKZ=4,-                 ; Bucket size of 4 blocks
                DEQ=4,-                 ; Default extension quantity
                NXT=DST_ALL2            ; Designate next XAB
DST_ALL2:
        $XABALL AID=2,-                 ; Alternate key data area
                ALQ=112,-               ; Allocation quantity and
                AOP=<CBT>,-             ; contiguous best try
                BKZ=2,-                 ; Bucket size of 2 blocks
                DEQ=38,-                ; Default extension quantity
                NXT=0                   ; No more XABs
;
; Allocate buffer to the size of the largest record being read.
;
BUFFER: .BLKB   66                      ; Buffer for input and output
        BUFFER_SIZE=.-BUFFER            ; Buffer size

;**********************************************************************

        .SBTTL  Mainline
        .PSECT  CODE            NOWRT,BYTE
;
; Start of program
;
        .ENTRY  CREATEIDX,^M<R6>        ; Entry point
;
; Open the source and destination files.
;
        $OPEN    FAB=SRC_FAB            ; Open input file
        BLBC     R0,EXIT1               ; Branch on failure
        $CONNECT RAB=SRC_RAB            ; Connect input record stream
        BLBC     R0,EXIT2               ; Branch on failure
        $CREATE  FAB=DST_FAB            ; Create output file
        BLBC     R0,EXIT3               ; Branch on failure
        $CONNECT RAB=DST_RAB            ; Connect output record stream
        BLBC     R0,EXIT4               ; Branch on failure
        BRB      LOOP                   ; Bypass signaling code
EXIT1:  MOVAL    SRC_FAB,R6             ; Keep FAB address
        BRB      F_ERR                  ; Signal error
EXIT2:  MOVAL    SRC_RAB,R6             ; Keep RAB address
        BRB      R_ERR                  ; Signal error
EXIT3:  MOVAL    DST_FAB,R6             ; Keep FAB address
        BRB      F_ERR                  ; Signal error
EXIT4:  MOVAL    DST_RAB,R6             ; Keep RAB address
        BRB      R_ERR                  ; Signal error
```

(continued on next page)

Example B–3 (Cont.) Use of the Create Service for an Indexed File

```
;
; Transfer records until end-of-file is reached.
;
LOOP:   $GET    RAB=SRC_RAB             ; Read next rec from input file
        BLBS    R0,PUT                  ; Branch on success
        CMPL    R0,#RMS$_EOF            ; Was it end-of-file (EOF)?
        BNEQ    EXIT2                   ; Branch if not EOF error
        BRB     CLOSE                   ; Close and exit if EOF
PUT:    $PUT    RAB=DST_RAB             ; Write 66-byte record to output
        BLBS    R0,LOOP                 ; On success, continue loop
        BRB     EXIT4                   ; On error, signal and exit
;
; Close the source and destination files.
;
F_ERR:  PUSHL   FAB$L_STV(R6)            ; Push STV and STS fields
        PUSHL   FAB$L_STS(R6)            ; on stack
        CALLS   #2, G^LIB$SIGNAL         ; Signal file error
        BRB     EXIT                     ; Exit
R_ERR:  PUSHL   RAB$L_STV(R6)            ; Push STV and STS fields
        PUSHL   RAB$L_STS(R6)            ; on stack
        CALLS   #2, G^LIB$SIGNAL         ; Signal file error
CLOSE:  $CLOSE  FAB=DST_FAB              ; Close output file
        $CLOSE  FAB=SRC_FAB              ; Close input file
EXIT:   $EXIT_S                          ; Exit
        .END    CREATEIDX                ; Specify starting address
```

This example program creates an indexed file with a primary key and two alternate keys that are defined by appropriate key definition control blocks (XABKEY). For efficiency, the file is divided into areas consisting of a data area and an index area for each key using multiple allocation control blocks (XABALL).

In each XABKEY, the DAN and IAN arguments (XAB$B_DAN and XAB$B_IAN fields) indicate the area identification number (AID) of the corresponding XABALL. By setting the RAB$V_LOA bit in RAB field RAB$L_ROP, the program directs RMS to use the DFL and IFL arguments (XAB$W_DFL and XAB$W_IFL fields) to determine the maximum initial fill size (in bytes) for data and index buckets (each bucket contains the number of blocks specified in the XABALL BKZ argument, XAB$B_BKZ field).

These are the XABKEY and XABALL control blocks for the primary key (the NAME key) in this example:

```
DST_KEY0:                                ; Primary key is Name
        $XABKEY REF=0,-                  ; Key reference number
                DAN=0,-                  ; Define data XABALL
                DFL=1536,-               ; Define data fill of 75%
                FLG=<DUP>,-              ; Allow duplicate keys
                DTP=DSTG,-               ; Decending sort order
                IAN=1,-                  ; Define index XABALL
                IFL=1536,-               ; Initial index fill 75%
                PROLOG=3,-               ; Request prolog 3
   .
   .
   .
```

```
DST_ALL0:
        $XABALL AID=0,-                 ; Data area definition
                ALQ=328,-               ; Allocation quantity and
                AOP=<CBT>,-             ; contiguous best try
                BKZ=4,-                 ; Bucket size of 4 blocks
                DEQ=112,-               ; Default extension quantity
                NXT=DST_ALL1            ; Designate next XAB
DST_ALL1:
        $XABALL AID=1,-                 ; Primary key index area
                ALQ=8,-                 ; Allocation quantity and
                AOP=<CBT>,-             ; contiguous best try
                BKZ=4,-                 ; Bucket size of 4 blocks
                DEQ=4,-                 ; Default extension quantity
                NXT=DST_ALL2            ; Designate next XAB
```

The allocation information was obtained using the File Definition Language (FDL) editor which is especially useful when you are creating large indexed files. The DCL commands CREATE/FDL and CONVERT can be used to create files by using an FDL file produced by the Edit/FDL utility, without any programming. Instead of using the multiple XABs for the key definition and area allocations in this program, a simpler approach is to use the FDL file produced by the Edit/FDL utility by invoking the FDL routines FDL$PARSE and FDL$RELEASE (for more information on these routines, see the *OpenVMS Utility Routines Manual*).

Fixed-length records are copied from the sequential input file on the local node to the indexed file on the remote node. Each variable-length output record is initially 66 bytes long and may be extended to a maximum of 128 bytes.

B.3.4 Processing File Specifications

The file name and file specification services, Parse and Search, are used for relatively complex operations such as processing wildcard characters.

Before you can perform operations on a file, you must establish a path to the file. You do this by specifying the file specification string address and size (FAB$L_FNA and FAB$B_FNS) fields (and possibly the default file specification string address and size fields) of the FAB to describe an ASCII string within the program. In this ASCII string, you can have a concatenation of the network node name; a logical or device name; the directory name; and the file name, type, and version number.

If a logical name is used, RMS translates the logical name into its equivalent file specification before it applies defaults to any missing components of the file specification. If the logical name is a search list logical name, RMS translates each element of the search list into an equivalent file specification before it applies defaults to that element. When using the Search service, a file specification that may contain a search list logical name must be handled as if wildcard characters were present in the file specification.

The Parse service is required prior to the Search service in order to examine the file specification for wildcard characters or a search list. If the file is found, the Parse service sets a NAM block bit that RMS uses internally and sets an appropriate value in the wildcard character context which is used as input by the Search service. The Parse service is invoked once, then the Search service is repetitively invoked as many times as there are files that match the original file specification.

If a wildcard is present, the Search service attempts to find all files that match the file specification. If an asterisk (*) is in the directory field, all directories on the specified device are searched for files that match the remaining file specification components. As with the use of wildcard characters, when a search list logical name is present, a single Parse service and multiple Search services return all files that match the file specification. With search lists, however, all list elements are searched for matching file specifications in the specified order without regard to uniqueness between the resulting file specifications. Search lists can be used in place of (or in addition to) wildcard characters to specify a more efficient search order, which can mean different combinations for the device, directory, file name, file type, and version number parts of a file specification. Search lists can also contain wildcard characters, if needed.

In summary, the Parse and Search services use a search list logical name very much like a wildcard. Unlike the case of opening a file, in which the first instance where the file is found successfully ends the use of additional search list file specifications, the Parse and Search services use all search list file specifications.

Example B–4 shows how the $PARSE and $SEARCH macros can be used in wildcard processing.

Example B–4 Wildcard Processing Using Parse and Search Services

```
        .TITLE  WILD
;
;  Program to accept wildcard characters in input (partial) file
;  specification and display full file specification.
;
        $NAMDEF                         ; NAM block definitions

        .PSECT  DATA,NOEXE,WRT
NAM_BLK:
        $NAM    RSA=RES_STR,-           ; Result buffer address
                RSS=NAM$C_MAXRSS,-      ; Result buffer size
                ESA=EXP_STR,-           ; Expanded buffer address
                ESS=NAM$C_MAXRSS        ; Expanded buffer size
FAB_BLK:
        $FAB    FOP=NAM,-               ; Use NAM block option
                NAM=NAM_BLK,-           ; Pointer to NAM block
                FNA=INP_STR             ; Addr of file name string

EXP_STR:                                ; Expanded string buffer
        .BLKB   NAM$C_MAXRSS
RES_STR:                                ; Resultant string buffer
        .BLKB   NAM$C_MAXRSS
RES_STR_D:                              ; Resultant string descriptor
        .BLKL   1
        .LONG   RES_STR
INP_STR:                                ; Input string buffer
        .BLKB   NAM$C_MAXRSS
INP_STR_D:                              ; Input string descriptor
        .LONG   NAM$C_MAXRSS
        .LONG   INP_STR
INP_STR_LEN:                            ; Input string length
        .BLKL   1
```

(continued on next page)

Example B–4 (Cont.) Wildcard Processing Using Parse and Search Services

```
PROMPT_D:                               ; User prompt string
        .ASCID  /Please enter the file specification: /

        .PSECT  CODE,EXE,NOWRT
        .ENTRY  WILD,^M<>
        PUSHAB  INP_STR_LEN             ; Address for string length
        PUSHAB  PROMPT_D                ; Prompt string descriptor
        PUSHAB  INP_STR_D               ; String buffer descriptor
        CALLS   #3,G^LIB$GET_INPUT      ; Get input string value
        BLBC    R0,EXIT                 ; Quit on error
;
; Store user input string and perform initial parse to
; set up RMS context for subsequent search.
;
        MOVB    INP_STR_LEN, -          ; Set string size
                FAB_BLK+FAB$B_FNS
        $PARSE  FAB=FAB_BLK             ; Parse the file spec
        BLBC    R0,F_ERR                ; Quit and signal on error
;
; Search until all possibilities are exhausted.
;
SEARCH_LOOP:
        $SEARCH FAB=FAB_BLK             ; Find next file
        BLBC    R0,SRCHERR              ; Any more?
;
; Print out the resultant string from the search operation
;
        MOVZBL  NAM_BLK+NAM$B_RSL, -
                RES_STR_D               ; Set string length
        PUSHAB  RES_STR_D               ; String descriptor
        CALLS   #1,G^LIB$PUT_OUTPUT     ; Output the result
        BLBC    R0,EXIT                 ; Quit on error
        BRB     SEARCH_LOOP             ; Go for more
SRCHERR:                                ; If error is "No more files",
        CMPL    R0,#RMS$_NMF            ; this is normal completion
        BEQL    S_EXIT                  ; of the search loop.
F_ERR:  PUSHL   FAB_BLK+FAB$L_STV       ; Push STV and STS on stack
        PUSHL   FAB_BLK+FAB$L_STS       ; in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal error
S_EXIT: MOVL    #1,R0                   ; Suppress "No More Files"
EXIT:   RET
        .END WILD
```

This program is designed to locate all files corresponding to a partial file specification input. The program prompts the user for an input string, which can consist of a partial file specification, using the wildcard characters and/or any type of logical name, including a search list logical name. In many respects, this program emulates the DCL command DIRECTORY, which is discussed in the *OpenVMS DCL Dictionary*.

The program illustrates the use of the $PARSE and $SEARCH file name processing macros. Here is the program statement that invokes the Parse service for parsing the file name string:

```
$PARSE FAB=FAB_BLK
```

Before invoking the Parse service ($PARSE macro), the program moves the input string length to the file name string (FAB$B_FNS) field. If the Parse service returns an error completion status, the program branches to the F_ERR error routine.

Assuming no error, the program searches the disk directories specified by the expanded string area address field in the NAM block (NAM$L_ESA) until all possible files conforming to the partial file specification input are found. Here is the program line that invokes the Search service:

```
$SEARCH FAB=FAB_BLK
```

A status test is performed immediately after the $SEARCH macro. If an error is detected, the program branches to the SRCHERR label. If a no-more-files condition is detected, RMS returns the RMS$_NMF message to indicate that all files that match the specification have been found. (This error, however, is not signaled.)

This program contains two run-time library routines: LIB$GET_INPUT and LIB$PUT_OUTPUT. The LIB$GET_INPUT routine inputs a record from the current controlling input device, specified by SYS$INPUT, using the Get service. The LIB$PUT_OUTPUT routine outputs a record (line) to the current controlling output device, specified by SYS$OUTPUT, using the Put service. Both routines are discussed in greater detail in the *OpenVMS RTL Library (LIB$) Manual*.

B.3.5 Connecting and Disconnecting Record Streams

To associate or disassociate a file with one or more record streams, RMS provides the Connect and Disconnect services, which are invoked using the $CONNECT and $DISCONNECT macros.

Before reading and writing file records, the program must open (or create) the input and output files and then connect the files to the appropriate record streams by executing the $OPEN (or $CREATE) macro followed by the $CONNECT macro.

Closing a file implicitly disconnects the record stream. Use the Disconnect service to explicitly disconnect a record stream that is not to be used immediately. This keeps the file open but releases various data structures for use by other processes until your program needs the record stream.

Example B–5 shows a program in which a user-entered reply determines which key path is selected to access the indexed file created in Example B–3. The user-entered value determines the value specified for the RAB$B_KRF field. The RAB$B_KRF value is set before the connect operation occurs because this field is input to the Connect service.

Example B–5 Use of the Connect Service and Multiple Keys

```
       .TITLE   MULTIKEY
;
REC_SIZE=128
        .PSECT  DATA NOEXE,LONG
;                                                         ** RMS DATA **
MODFAB: $FAB    FNM=<DATA_OUTPUT.DAT>,-        ; FAB file spec.
                FAC=<GET>,-                    ; Get access needed
                SHR=<GET, UPD, PUT>,-          ; Allow Get, Update, Put
                MRS=REC_SIZE                   ; Specify record size
MODRAB: $RAB    FAB=MODFAB,-                   ; RAB; indicate FAB
                MBF=3,-                        ; Use 3 buffers
                UBF=REC_MODBUF,-               ; Specify buffer
                USZ=REC_SIZE,-
                KRF=0                          ; Primary is default key
REC_START:      .LONG    REC_SIZE              ; Record buffer
                .ADDRESS REC_MODBUF
REC_MODBUF:     .BLKB    REC_SIZE
;                                              TERMINAL I/O DATA **
MPRO0:  .ASCID  /  /
MPRO1:  .ASCID  /Enter list order: 1-by name, 2-by city, 3-by state, 9-end  :/
ENTRYERR:       .ASCID  /* * Value entered must be 1, 2, 3, or 9. * */
;
REGANS:         .LONG    1
                .ADDRESS REGBUF
REGBUF:         .BLKB    1
;
DONE:   .ASCID /Press RETURN to continue/
;
        .PSECT  CODE
START:  .WORD   ^M<>
INPUT:  PUSHAL  MPRO0                   ; Get input
        PUSHAL  MPRO1                   ; Display prompt
        PUSHAL  REGANS
        CALLS   #3, G^LIB$GET_INPUT
        BLBC    R0,FINIBR
        CMPB    #^A/1/,REGBUF           ; Test value of menu answer
        BEQLU   PRIM                    ; 1 means primary
        CMPB    #^A/2/,REGBUF           ; Continue testing
        BEQLU   ALT1                    ; 2 means first alternate
        CMPB    #^A/3/,REGBUF           ; Continue testing
        BEQLU   ALT2                    ; 3 means second alternate
        CMPB    #^A/9/,REGBUF           ; Continue testing
        BEQLU   FINIBR                  ; 9 means end program
BADANS: PUSHAL  ENTRYERR                ; otherwise, display error message
        CALLS   #1, G^LIB$PUT_OUTPUT
        BLBC    R0,FINIBR
        BRB     INPUT                   ; Entry error; retry
FINIBR: BRW FINI                    ; branch extender
PRIM:   MOVB    #0,MODRAB+RAB$B_KRF     ; Set key of reference in RAB
        BRB     OPEN
ALT1:   MOVB    #1,MODRAB+RAB$B_KRF     ; Set key of reference in RAB
        BRB     OPEN
ALT2:   MOVB    #2,MODRAB+RAB$B_KRF     ; Set key of reference in RAB
OPEN:   $OPEN   FAB=MODFAB              ; Open file
        BLBS    R0,CONN
        BRW     ERROR_OPEN
CONN:   $CONNECT RAB=MODRAB             ; Connect record stream
        BLBS    R0,NEXT
        BRW     ERROR
```

(continued on next page)

Example B–5 (Cont.) Use of the Connect Service and Multiple Keys

```
NEXT:   $GET    RAB=MODRAB              ; Get record
        CMPL    #RMS$_EOF,R0            ; Test if EOF
        BEQLU   CLEAN
        BLBC    R0,ERROR
        MOVZWL  RAB$W_USZ+MODRAB,REC_START ; Set ASCII descriptor length
        PUSHAL  REC_START                  ; Display each record
        CALLS   #1, G^LIB$PUT_OUTPUT
        BLBS    R0,NEXT
        BRB     FINI                    ; Repeat until EOF
CLEAN:  $CLOSE  FAB=MODFAB              ; Close file
        BLBC    R0,ERROR_OPEN
        PUSHAL  MPRO0
        CALLS   #1, G^LIB$PUT_OUTPUT
        BLBC    R0,FINI
        PUSHAL  DONE
        PUSHAL  REGANS
        CALLS   #2, G^LIB$GET_INPUT
        BLBC    R0,FINI
        BRW     INPUT
;
ERROR_OPEN:
        PUSHL   MODFAB+FAB$L_STV             ; Error opening
        PUSHL   MODFAB+FAB$L_STS             ; file. Signal error
        CALLS   #2, G^LIB$SIGNAL             ; using LIB$SIGNAL.
        BRB     FINI                         ; End program
ERROR:  PUSHL   MODRAB+RAB$L_STV             ; Record-related error
        PUSHL   MODRAB+RAB$L_STS
        CALLS   #2, G^LIB$SIGNAL             ; Signal error, then
        $CLOSE  FAB=MODFAB                   ; close file
FINI:   RET
        .END    START
```

Here the SHR argument limits access to processes that perform the Get service, Put service, and Update service. If you anticipate no file modifications as your program accesses the file, you can improve performance by having the SHR argument limit access to processes that use the Get service (SHR=GET).

Errors are signaled according to the recommended practice of using the FAB$L_STS and FAB$L_STV fields for file errors and RAB$L_STS and RAB$L_STV fields for record errors.

B.3.6 Other File-Processing Operations

Other file services include the Display, Erase, Extend, Remove, and Rename services, which can be invoked using the $DISPLAY, $ERASE, $EXTEND, $REMOVE, and $RENAME macros, respectively.

Example B–6 illustrates the use of the Rename service to rename a file from directory [USER] named NAMES.DAT to directory [USER.HISTORY] named OLD_NAMES.DAT.

Example B–6 Use of the Rename Service

```
        .TITLE  RENAME
;
;  Program that renames a file into a different directory and
;  displays the resultant string.
;
        .PSECT  DATA,NOEXE,WRT
;
;  Define old FAB, old NAM, new FAB, new NAM, and buffers
;
OLD_FAB:                                ; Define old file FAB
        $FAB    FNM=<[USER]NAMES.DAT>,-
                NAM=OLD_NAM             ; Pointer to NAM block
OLD_NAM:                                ; Define old file NAM
        $NAM    ESA=EXP_OLD,-           ; Equivalence string
                ESS=NAM$C_MAXRSS,-      ; address and size
                RSA=RES_OLD,-           ; Resultant string
                RSS=NAM$C_MAXRSS        ; address and size
NEW_FAB:                                ; Define new file FAB
        $FAB    FNM=<[USER.HISTORY]OLD_NAMES.DAT>,-
                NAM=NEW_NAM             ; Pointer to NAM block
NEW_NAM:
        $NAM    ESA=EXP_NEW,-           ; Equivalence string
                ESS=NAM$C_MAXRSS,-      ; address and size
                RSA=RES_NEW,-           ; Resultant string
                RSS=NAM$C_MAXRSS        ; address and size

EXP_OLD:                                ; Old file equivalence
        .BLKB   NAM$C_MAXRSS            ; string buffer
EXP_NEW:                                ; New file equivalence
        .BLKB   NAM$C_MAXRSS            ; string buffer
RES_OLD:                                ; Old file resultant
        .BLKB   NAM$C_MAXRSS            ; string buffer
RES_OLD_D:                              ; String descriptor
        .BLKL   1
        .LONG   RES_OLD
RES_NEW:                                ; New file resultant
        .BLKB   NAM$C_MAXRSS            ; string buffer
RES_NEW_D:                              ; String descriptor
        .BLKL   1
        .LONG   RES_NEW
;
MESS:   .ASCID /has been successfully relocated to /
;
        .PSECT  CODE,EXE,NOWRT
        .ENTRY  RENAME,^M<>
                                        ;  Rename file
;
        $RENAME OLDFAB=OLD_FAB, NEWFAB=NEW_FAB
        BLBC    R0,ERROR
                                        ; Set up descriptors
;
```

(continued on next page)

Example B–6 (Cont.) Use of the Rename Service

```
        MOVZBL  OLD_NAM+NAM$B_RSL,RES_OLD_D
        MOVZBL  NEW_NAM+NAM$B_RSL,RES_NEW_D
;
        PUSHAL  RES_OLD_D               ; Push resultant name,
        CALLS   #1,G^LIB$PUT_OUTPUT     ; display old file spec.
        BLBC    R0,TERM_ERROR           ; Branch on error
        PUSHAL  MESS                    ; Push message on stack,
        CALLS   #1,G^LIB$PUT_OUTPUT     ; display message
        BLBC    R0,TERM_ERROR           ; Branch on error
        PUSHAL  RES_NEW_D               ; Push resultant name,
        CALLS   #1,G^LIB$PUT_OUTPUT     ; display new file spec.
        BLBS    R0,DONE                 ; Branch on success
TERM_ERROR:
        PUSHL   R0                      ; Signal output error
        CALLS   #1,G^LIB$SIGNAL         ; from R0
        BRB     DONE
ERROR:  PUSHL   OLD_FAB+FAB$L_STV       ; Push STV and STS on
        PUSHL   OLD_FAB+FAB$L_STS       ; stack (reverse order)
        CALLS   #2,G^LIB$SIGNAL         ; Signal error
DONE:   RET
        .END RENAME
```

This program uses the Rename service to change both the directory and the name of the object file, which is being replaced by a new file (created by a separate program). If the Rename service executes correctly, the resultant file specification of the old file, the message defined by the ASCII descriptor following the label MESS, and the resultant file specification of the new file are displayed as verification that the Rename service successfully completed.

B.3.7 Retrieving and Inserting Records

The record-processing services provided by RMS insert records into a file and retrieve records from a file. These services are the Find, Get, and Put services, which can be invoked by the $FIND, $GET, and $PUT macros, respectively.

Example B–7 illustrates the use of the $GET and $PUT macros. It connects the input and output record streams, reads a record from an indexed file, and writes the record to a relative file. The program illustrates the use of the key string buffer, the key string descriptor, and the key string length when reading indexed records, and it includes the use of a user prompt string.

Example B–7 Use of the Get and Put Services

```
        .TITLE  LOOKUP
;
; This program looks up records in the input file and
; writes the records to the output file.

        .PSECT  DATA,WRT,NOEXE
INFAB:  $FAB    FNM = <INFILE:>,-       ; Input file logical name
                SHR = <GET,PUT,UPD,DEL> ; Allow read/write sharing
```

(continued on next page)

Example B–7 (Cont.) Use of the Get and Put Services

```
INRAB:  $RAB    FAB = INFAB,-           ; Pointer to FAB
                KBF = INP_STR,-         ; Key buffer
                KRF = 0,-               ; Primary key
                RAC = KEY,-             ; Keyed access
                ROP = WAT,-             ; Wait for record
                UBF = REC_BUFF,-        ; Record buffer
                USZ = REC_SIZE          ; and size
OUTFAB: $FAB    FNM = <OUTFILE:>,-      ; Output file logical name
                BKS = 3,-               ; 3 blocks per bucket
                MRS = REC_SIZE,-        ; Maximum record size
                ORG = REL,-             ; Relative file
                RAT = CR                ; Implied carriage control
OUTRAB: $RAB    FAB = OUTFAB,-          ; Pointer to FAB
                RBF = REC_BUFF          ; Output uses same buffer
                                        ; as input
REC_SIZE = 132                          ; Maximum size records
REC_BUFF:
        .BLKB   REC_SIZE                ; Record buffer
INP_STR:                                ; Key string buffer
        .BLKB   REC_SIZE
INP_STR_D:                              ; Key string descriptor
        .LONG   REC_SIZE
        .LONG   INP_STR
INP_STR_LEN:                            ; Key string length
        .BLKL   1
PROMPT_D:                               ; User prompt string
        .ASCID  /Please input key value: /
        .PSECT  CODE,NOWRT,EXE
;
; Initialization - Open input and output files and connect streams
;
        .ENTRY  LOOKUP,^M<>             ; No registers to save
        $OPEN   FAB=INFAB               ; Open input file
        BLBC    R0,EXIT1                ; Quit on error
        $CONNECT        RAB=INRAB       ; Connect to input
        BLBC    R0,EXIT2                ; Quit on error
        $CREATE FAB=OUTFAB              ; Create output file
        BLBC    R0,EXIT3                ; Quit on error
        $CONNECT        RAB=OUTRAB      ; Connect to output
        BLBC    R0,EXIT4                ; Quit on error
        BRB     READ                    ; Skip error branching
EXIT1:  MOVAL   INFAB, R6               ; Keep INFAB address
        BRW     F_ERR                   ; Signal FAB error
EXIT2:  MOVAL   INRAB, R6               ; Keep INRAB address
        BRW     R_ERR                   ; Signal RAB error
EXIT3:  MOVAL   OUTFAB, R6              ; Keep OUTFAB address
        BRB     F_ERR                   ; Signal FAB error
EXIT4:  MOVAL   OUTRAB, R6              ; Keep OUTRAB address
        BRB     R_ERR                   ; Signal RAB error
;
; Loop to copy records
;
READ:
        PUSHAB  INP_STR_LEN             ; Address for string length
        PUSHAB  PROMPT_D                ; Prompt string descriptor
        PUSHAB  INP_STR_D               ; String buffer descriptor
        CALLS   #3,G^LIB$GET_INPUT      ; Get input string value
        BLBS    R0,GET                  ; Quit on error or end-of-file
        CMPL    R0,#RMS$_EOF            ; Was error end-of-file?
        BEQL    DONE                    ; Successful completion
        BRB     EXIT                    ; Error otherwise
```

(continued on next page)

Example B–7 (Cont.) Use of the Get and Put Services

```
GET:    MOVB    INP_STR_LEN, -          ; Set key size
                INRAB+RAB$B_KSZ
        $GET    RAB=INRAB               ; Get a record
        BLBS    R0,PUT                  ; Put if successful
        CMPL    R0,#RMS$_RNF            ; No such record?
        BEQL    READ                    ; Try again
        BRB     EXIT2                   ; Error otherwise
PUT:    MOVW    INRAB+RAB$W_RSZ, -      ; Set the record size
                OUTRAB+RAB$W_RSZ        ; for output
        $PUT    RAB=OUTRAB              ; Write the record
        BLBC    R0,EXIT4                ; Quit on error
        BRB     READ                    ; Go back for more
;
; Close files and exit
;
F_ERR:  PUSHL   FAB$L_STV(R6)           ; Push STV and STS on
        PUSHL   FAB$L_STS(R6)           ; stack in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal message
        BRB     EXIT
R_ERR:  PUSHL   RAB$L_STV(R6)           ; Push STV and STS on
        PUSHL   RAB$L_STS(R6)           ; stack in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal message
DONE:   $CLOSE  FAB=INFAB               ; Close input
        $CLOSE  FAB=OUTFAB              ; and output
EXIT:   RET                             ; Return with status in R0
        .END    LOOKUP
```

This program writes records from an existing indexed input file into a newly created relative output file.

The program configures the file-sharing field (FAB$B_SHR) in the input FAB to permit sharing of the file by processes that use the Get, Put, Update, and Delete services.

The output FAB sets the bucket size field (FAB$B_BKS) at 3 blocks per bucket, limits the record size in the output file to 132 bytes, specifies the relative file organization, and specifies an implicit carriage control when the file output is directed to a terminal.

The RAB for the input file establishes the key data, sets the WAIT record option, and defines the record buffer. The output RAB locates the record buffer. The rest of the first program section assigns values and allocates space to various program variables. After the program opens and creates the two files and connects the record streams, it executes a series of instructions at label READ that input the required key values and the user prompt. Then the program uses the $GET and $PUT macros to invoke the respective services for retrieving and inserting the records. The $GET macro uses the INRAB and the $PUT macro uses the OUTRAB, as shown in the following program statements:

```
$GET   RAB=INRAB

$PUT   RAB=OUTRAB
```

Each time the program reads or writes a record, it performs a status check. If the status check is successful, the program branches back to the READ label for the next record. If any of the status checks indicate an error, the program branches to the appropriate error handler before exiting.

When the program completes the records transfers, it branches to the DONE label to close the record and exit.

B.3.8 Deleting Records

This service can only be used with relative and indexed files. Example B–8 illustrates the use of the Delete service.

Example B–8 Use of the Delete Service

```
        .TITLE  DELETE
;
; This program looks up records in the input file and
; deletes those records.
;

        .PSECT  DATA,WRT,NOEXE
INFAB:   $FAB   FNM = <INFILE:>,-       ; Input file logical name
                FAC = <DEL,GET>         ; DEL access
INRAB:   $RAB   FAB = INFAB,-           ; Pointer to FAB
                KBF = INP_STR,-         ; Key buffer
                KRF = 0,-               ; Primary key
                RAC = KEY               ; Keyed access
REC_SIZE = 132                          ; Maximum size records
INP_STR:                                ; Key string/record buffer
        .BLKB   REC_SIZE
INP_STR_D:                              ; Key string descriptor
        .LONG   REC_SIZE
        .LONG   INP_STR
INP_STR_LEN:                            ; Key string length
        .BLKL   1
KEY_PMT_D:                              ; Key value prompt string
        .ASCID  /Please enter key value: /
        .PSECT  CODE,NOWRT,EXE
;
; Initialization - Open file and connect stream
;
        .ENTRY  DELETE,^M<>             ; No registers to save
        $OPEN   FAB=INFAB               ; Open input file
        BLBC    R0,F_ERR                ; Quit on error
        $CONNECT        RAB=INRAB       ; Connect to input
        BLBC    R0,R_ERR                ; Quit on error
;
; Delete record loop
;
READ:
        PUSHAB  INP_STR_LEN             ; Address for string length
        PUSHAB  KEY_PMT_D               ; Prompt string descriptor
        PUSHAB  INP_STR_D               ; String buffer descriptor
        CALLS   #3,G^LIB$GET_INPUT      ; Get input string value
        BLBS    R0,FIND                 ; Quit on error or end-of-file
        CMPL    R0,#RMS$_EOF            ; Was error end-of-file?
        BEQL    DONE                    ; Successful completion
        BRB     EXIT                    ; Error otherwise
FIND:   MOVB    INP_STR_LEN, -          ; Set key size
                INRAB+RAB$B_KSZ
        $FIND   RAB=INRAB               ; Locate the record
        BLBS    R0,DEL                  ; Continue if found
        CMPL    R0,#RMS$_RNF            ; No such record?
        BEQL    READ                    ; Try again
        BRB     R_ERR                   ; Error otherwise
DEL:    $DELETE RAB=INRAB               ; Delete the record
        BLBC    R0,R_ERR                ; Quit on error
        BRB     READ                    ; Go back for more
```

(continued on next page)

Example B–8 (Cont.) Use of the Delete Service

```
;
; Close files and exit
;
F_ERR:  PUSHL   FAB$L_STV+INFAB         ; Push STV and STS on
        PUSHL   FAB$L_STS+INFAB         ; stack in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal message
        BRB     EXIT
R_ERR:  PUSHL   RAB$L_STV+INRAB         ; Push STV and STS on
        PUSHL   RAB$L_STS+INRAB         ; stack in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal message
DONE:   $CLOSE  FAB=INFAB               ; Close files
EXIT:   RET                             ; Return with status in R0
        .END    DELETE
```

This program uses a key to find and delete a record. To use the $DELETE macro, the $FAB macro for the file must set the FAB$V_DEL bit as shown in the following code example:

```
INFAB:  $FAB    FNM=<INFILE:>,-
                FAC=<DEL>
```

The following program statement invokes the Delete service and points to the input RAB:

```
$DELETE       RAB=INRAB
```

B.3.9 Updating Records

Example B–9 illustrates the use of the Update service.

Example B–9 Use of the Update Service

```
        .TITLE  UPDATE
;
; This program looks up records in the input file and
; updates those records.
;

        .PSECT  DATA,WRT,NOEXE
INFAB:  $FAB    FNM = <INFILE:>,-       ; Input file logical name
                FAC = <GET,UPD>         ; Read and Write access
INRAB:  $RAB    FAB = INFAB,-           ; Pointer to FAB
                KBF = INP_STR,-         ; Key buffer
                KRF = 0,-               ; Primary key
                RAC = KEY,-             ; Keyed access
                RBF = INP_STR           ; Record buffer
REC_SIZE = 132                          ; Maximum size records
INP_STR:                                ; Key string/record buffer
        .BLKB   REC_SIZE
INP_STR_D:                              ; Key string descriptor
        .LONG   REC_SIZE
        .LONG   INP_STR
INP_STR_LEN:                            ; Key string length
        .BLKL   1
KEY_PMT_D:                              ; Key value prompt string
        .ASCID  /Please input key value: /
DATA_PMT_D:                             ; Data value prompt string
        .ASCID  /Please input new record value: /
```

(continued on next page)

Example B–9 (Cont.) Use of the Update Service

```
        .PSECT  CODE,NOWRT,EXE
;
; Initialization - Open file and connect stream
;
        .ENTRY  UPDATE,^M<>             ; No registers to save
        $OPEN   FAB=INFAB               ; Open input file
        BLBC    R0,FAB_E                ; Quit on error
        $CONNECT        RAB=INRAB       ; Connect to input
        BLBC    R0,RAB_E                ; Quit on error
        BRB     READ                    ; Begin update loop
FAB_E:  BRW     F_ERR                   ; File (FAB) error
RAB_E:  BRW     R_ERR                   ; Record (RAB) error
;
; Update record loop
;
READ:
;
; Prompt for key value to look up.
;
        PUSHAB  INP_STR_LEN             ; Address for string length
        PUSHAB  KEY_PMT_D               ; Prompt string descriptor
        PUSHAB  INP_STR_D               ; String buffer descriptor
        CALLS   #3,G^LIB$GET_INPUT      ; Get input string value
        BLBS    R0,FIND                 ; Quit on error or end-of-file
        CMPL    R0,#RMS$_EOF            ; Was error end-of-file?
        BEQL    ALL_D                   ; Successful completion
        BRW     EXIT                    ; Error otherwise
ALL_D:  BRW     DONE
FIND:   MOVB    INP_STR_LEN, -          ; Set key size
                INRAB+RAB$B_KSZ
        $FIND   RAB=INRAB               ; Locate the record
        BLBS    R0,UPD                  ; Continue if found
        CMPL    R0,#RMS$_RNF            ; No such record?
        BEQL    READ                    ; Try again
        BRB     R_ERR                   ; Error otherwise
;
; Prompt for new data record.
;
UPD:
        PUSHAB  INP_STR_LEN             ; Address for string length
        PUSHAB  DATA_PMT_D              ; Prompt string descriptor
        PUSHAB  INP_STR_D               ; String buffer descriptor
        CALLS   #3,G^LIB$GET_INPUT      ; Get input string value
        BLBC    R0,EXIT                 ; Quit on error
        MOVW    INP_STR_LEN, -          ; Set record size
                INRAB+RAB$W_RSZ
        $UPDATE RAB=INRAB               ; Write the record
        BLBC    R0,R_ERR                ; Quit on error
        BRW     READ                    ; Go back for more
;
; Close files and exit
;
F_ERR:  PUSHL   FAB$L_STV+INFAB         ; Push STV and STS on
        PUSHL   FAB$L_STS+INFAB         ; stack in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal message
        BRB     EXIT
```

(continued on next page)

Example B–9 (Cont.) Use of the Update Service

```
R_ERR:  PUSHL   RAB$L_STV+INRAB         ; Push STV and STS on
        PUSHL   RAB$L_STS+INRAB         ; stack in reverse order
        CALLS   #2, G^LIB$SIGNAL        ; Signal message
DONE:   $CLOSE  FAB=INFAB               ; Close files
EXIT:   RET                             ; Return with status in R0
        .END    UPDATE
```

This program uses a key and a new record entered from the terminal to update a record in the input file.

To use the $UPDATE macro, the $FAB macro for the file must specify that the FAB$V_UPD bit is marked in the file access (FAB$B_FAC) field as shown in the following code example:

```
INFAB:  $FAB   FNM=<INFILE:>,-
               FAC=<GET,UPD>
```

Before updating a record, the program uses the Find service to locate the record by executing the $FIND macro located at the FIND label:

```
$FIND   RAB=INRAB
```

B.3.10 Using Block I/O

In addition to the major types of record access provided by the sequential, random by key value or relative record number, and random by RFA access modes, RMS provides another means to access data in a file: block I/O.

Block I/O operations let you directly read or write the blocks of a file. These operations are provided for users who must keep system overhead to a minimum and need no interpretation of file data as logical records, yet still want to take advantage of RMS file accessibility. Block I/O is an intermediate step between the RMS record operations and direct use of $QIO system services.

The three block I/O services are invoked using the $READ, $SPACE, and $WRITE macros, respectively.

- The Read service transfers a specified number of bytes to memory.
- The Space service positions a file forward or backward a specified number of blocks.
- The Write service writes a specified number of bytes to a file.

The Read and Write services always begin on a block boundary.

In addition to the Read, Space, and Write services, you can use the following services on a record stream connected for block I/O operations:

- The Disconnect service ($DISCONNECT macro)
- The Flush service ($FLUSH macro)
- The Next Volume service ($NXTVOL macro)
- The Rewind service ($REWIND macro)

These services perform miscellaneous operations or disconnect the record stream. They do not work on the contents of the records themselves.

You cannot perform block I/O operations on shared files. That is, file access for block I/O operations is denied unless the FAB$V_UPI or the FAB$V_NIL bit is set in the FAB$B_SHR field.

You specify block I/O operations for a record stream by setting the FAB$V_BIO bit in the file access (FAB$B_FAC) field as input to the Open or Create services. If you intend to write to the file, you must set the PUT option in the FAB$B_FAC field; if you intend to read from the file, you must set the GET option in the FAB$B_FAC field. If you set the FAB$V_BIO bit when you create a relative or indexed file, RMS omits prolog processing for indexed files and initial space prezeroing in relative files.

For files of unknown organization, block I/O is the only form of processing allowed. Processing proceeds identically to that for block I/O to the relative file organization described previously.

B.3.11 Mixed Block and Record I/O

How and when RMS allows you to switch between record I/O and block I/O depends on the organization of the file being accessed.

When you access sequential files, RMS allows you to switch between record I/O and block I/O with each record operation, if desired. To enable I/O switching for a record stream connected to a sequential file, use the following procedure:

1. Set the FAB$V_BRO option in the FAB$B_FAC field as input to the Create or Open service.
2. Clear the RAB$L_ROP field RAB$V_BIO option as input to the Connect service.

This procedure informs RMS that it should check the RAB$V_BIO option in the RAB$L_ROP field after each operation.

To do a block I/O operation:

1. Set the RAB$L_ROP field RAB$V_BIO option.
2. Invoke a block I/O service (Read, Space, or Write).

To do a record I/O operation:

1. Clear the RAB$L_ROP field RAB$V_BIO option.
2. Invoke a record I/O service.

Use care if you do choose to mix record and block I/O operations for sequential files. When you switch operations on disk devices, the context of the current record, the next record, and the next block pointer is undefined. Thus, the first operation after the switch must not use sequential record access mode. For magnetic tape devices, the context of the next record or next block indicates the start of the following block on the tape for the first operation after the switch.

As previously noted, you usually set the FAB$B_FAC field FAB$V_BRO option only to indicate that you want to mix record I/O and block I/O operations. If you decide that you want to perform block I/O processing only, you can set the RAB$L_ROP field RAB$V_BIO option after you open the file but before you invoke the Connect service. This connect-time operation overrides the setting of the FAB$V_BRO option for the current record stream and indicates to the Connect service that you only intend to do block I/O for this file, thus eliminating the need to allocate internal I/O buffers. (However, you must still allocate buffers for block I/O operations in your application program.) If you set the FAB$V_BRO option when you create an indexed file, the key definition XABs for that file must be present.

When you access relative or indexed files, switching is available only if you close and reopen the file. RMS does not permit both types of I/O simultaneously. When multiple record streams are used, all record streams must use the same type of I/O, either record I/O or block I/O.

You specify the I/O type when you create or open a file by selecting either the block I/O option (FAB$V_BIO bit set) or the record I/O option (FAB$V_BIO bit clear). For relative and indexed files, the decision to use block I/O or record I/O for a file can be postponed, if desired, until the record stream is connected by the following procedure:

1. Set the FAB$B_FAC field FAB$V_BRO option when you are opening (or creating) the file.
2. Indicate the appropriate operation to the Connect service by either setting the RAB$V_BIO bit in the RAB$L_ROP for block I/O or by clearing it for record I/O.

B.3.12 Next Block Pointer (NBP)

For block I/O operations to sequential files on disk devices, RMS maintains an internal next block pointer (NBP) that does the following functions:

- Points to the beginning of the file following execution of a Connect service if the RAB$V_EOF option in the RAB$L_ROP field of the RAB is cleared. If the RAB$L_ROP field RAB$V_EOF option is set, the NPB points to the block following the end of the file. The RAB$V_EOF option is relevant only for sequential files doing block I/O processing.
- Points to the block following the highest numbered block transferred by a read or write operation.
- Points to the next block following an operation invoked by the Space service.

An explicit Extend service is required for relative and indexed files because RMS does not automatically extend a file's allocation when using block I/O processing.

Example B–10 illustrates how to copy a file using block I/O.

Example B–10 Use of Block I/O

```
        .TITLE  BLOCKIO
;
; This program copies the input file to the output file.
; It illustrates block I/O using the RMS $READ and $WRITE
; macros.
;
        .PSECT  DATA,WRT,NOEXE
INFAB:  $FAB    FNM = <INFILE:>,-       ; Input file name
                FAC = <BIO,GET>         ; Block I/O read operations
INRAB:  $RAB    FAB = INFAB,-           ; Pointer to FAB
                BKT = 0,-               ; Start with current block
                UBF = REC_BUFF,-        ; Record buffer
                USZ = REC_SIZE          ;   and size
```

(continued on next page)

Example B–10 (Cont.) Use of Block I/O

```
OUTFAB: $FAB    FNM = <OUTFILE:>,-      ; Output file name
                FOP = CBT,-             ; Try for contiguous file
                MRS = REC_SIZE,-        ; Maximum record size
                FAC = <BIO,PUT>,-       ; Block I/O write operations
                RAT = CR                ; Implied carriage control
OUTRAB: $RAB    FAB = OUTFAB,-          ; Pointer to FAB
                BKT = 0,-               ; Start with current block
                RBF = REC_BUFF          ; Output uses same buffer
                                        ;   as input
REC_SIZE = 1024                         ; Maximum record size
REC_BUFF:
        .BLKB   REC_SIZE                ; Record buffer

        .PSECT  CODE,NOWRT,EXE
;
; Initialization - Open input and output files and connect streams
;
        .ENTRY  BLOCKIO,^M<>            ; No registers to save
        $OPEN   FAB=INFAB               ; Open input file
        BLBC    R0,EXIT1                ; Quit on error
        $CONNECT        RAB=INRAB       ; Connect to input
        BLBC    R0,EXIT2                ; Quit on error
        MOVL    INFAB+FAB$L_ALQ,-       ; Set proper size
                OUTFAB+FAB$L_ALQ        ;   for output
        $CREATE FAB=OUTFAB              ; Create output file
        BLBC    R0,EXIT3                ; Quit on error
        $CONNECT        RAB=OUTRAB      ; Connect to output
        BLBC    R0,EXIT4                ; Quit on error
;
; Copy loop
;
READ:   $READ   RAB=INRAB               ; Get a block
        BLBS    R0,WRITE                ; Write the block
        CMPL    R0,#RMS$_EOF            ; Was error end-of-file?
        BEQL    DONE                    ; Successful completion
        BRB     EXIT2                   ; If not, signal error
WRITE:  MOVW    INRAB+RAB$W_RSZ, -      ; Set the record size
                OUTRAB+RAB$W_RSZ        ;   for output
        $WRITE  RAB=OUTRAB              ; Write the block
        BLBC    R0,EXIT4                ; Quit on error
        BRB     READ                    ; Go back for more
;
; Error Signaling
;
EXIT1:  MOVL    INFAB+FAB$L_STS,R2      ; Move STS into R2
        MOVL    INFAB+FAB$L_STV,R3      ; Move STV into R3
        BRB     EXIT                    ; Signal error
EXIT2:  MOVL    INRAB+RAB$L_STS,R2      ; Move STS into R2
        MOVL    INRAB+RAB$L_STV,R3      ; Move STV into R3
        BRB     EXIT                    ; Signal error
```

(continued on next page)

Example B–10 (Cont.) Use of Block I/O

```
EXIT3:  MOVL    OUTFAB+FAB$L_STS,R2     ; Move STS into R2
        MOVL    OUTFAB+FAB$L_STV,R3     ; Move STV into R3
        BRB     EXIT                    ; Signal error
EXIT4:  MOVL    OUTRAB+RAB$L_STS,R2     ; Move STS into R2
        MOVL    OUTRAB+RAB$L_STV,R3     ; Move STV into R3
        BRB     EXIT                    ; Signal error
;
; Close files and exit
;
DONE:   $CLOSE  FAB=INFAB               ; Close input and
        $CLOSE  FAB=OUTFAB              ;   output files
        RET                             ; Return w/ success in R0
EXIT:   PUSHL   R3                      ; Push STV and STS
        PUSHL   R2                      ; on stack
        CALLS   #2, G^LIB$SIGNAL        ; Signal error
        RET                             ; Return w/ status in R0
        .END    BLOCKIO
```

This example program uses block I/O to transfer the contents of the input file to the output file. The following program data statements specify block I/O read operations from the input file by setting the FAB$V_BIO bit (block I/O) and the FAB$V_GET bit (read) in the FAB$B_FAC field of the input file's FAB:

```
INFAB:  $FAB    FNM = <INFILE>, -       ;Input file name
                FAC = <BIO,GET>,-
```

The following data statements specify block I/O write operations to the output file by setting the FAB$V_BIO bit (block I/O) and the FAB$V_PUT bit (write) in the FAB$B_FAC field of the output file's FAB:

```
OUTFAB: $FAB    FNM = <INFILE>, -       ;Output file name
                FAC = <BIO,PUT>,-
```

The input file's contents are copied until the end of file is encountered. Any errors are signaled with the convention of using both the STS and STV fields of the appropriate control block.

Index

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W

X