Professional™
300 series

PRO/RMS–11
Macro Programmer's Guide

Order No. AA-P099A-TK

Developer's Tool Kit

digital
software

PRO/RMS–11
Macro Programmer's Guide

Order No. AA-P099A-TK

November 1982

This document is a reference manual describing the macros and symbols that make up the interface between a MACRO–11 program and the operation routines of Record Management Services (RMS–11) for the Professional Operating System (P/OS).

OPERATING SYSTEM AND VERSION: P/OS V1.0 or later
SOFTWARE VERSION: RMS–11 Version 2.0

digital equipment corporation · maynard, massachusetts

First Printing, November 1982

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

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

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

ZK2167

HOW TO ORDER ADDITIONAL DOCUMENTATION

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

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

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

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

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

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

DIRECT MAIL ORDERS (CANADA)

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

DIRECT MAIL ORDERS (INTERNATIONAL)

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

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

CONTENTS

CONTENTS

CONTENTS

EXAMPLES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual is a guide to the use of RMS-11 in programs written in MACRO-11. It contains information necessary to writing MACRO-11 programs and subprograms that use RMS-11 operations.

INTENDED AUDIENCE

This manual is intended for both the MACRO-11 programmer who wants to use RMS-11 operations and the high-level language programmer who wants to use RMS-11 operations in a MACRO-11 subprogram.

Before reading this manual, you should read:

- RMS-11: An Introduction

STRUCTURE OF THIS DOCUMENT

- Chapter 1, Introduction to RMS-11 with MACRO-11, introduces RMS-11 macros and symbols that are the interface between a MACRO-11 program and RMS-11 operation routines.
- Chapter 2, RMS-11 Programming in MACRO-11, shows how to use RMS-11 macros and symbols in a MACRO-11 program.
- Chapter 3, Processing Directories and Files, shows how to use directory and file operations to process directories and files.
- Chapter 4, Processing Records and Blocks, shows how to use stream operations and either record or block operations to process records or blocks.
- Chapter 5, Operation Macro Descriptions, describes in detail each RMS-11 operation macro, the control blocks it uses, the options you can specify in each control block field, and the values returned in control block fields.
- Chapter 6, Control Block Fields, summarizes the use of each control block, field, value, and mask.
- Chapter 7, Example Programs, contains programs and program segments that illustrate the uses of some major RMS-11 features.
- Appendix A, Completion Codes and Fatal Error Codes, lists RMS-11 completion symbols, values, and meanings.
- Appendix B, Assembly-Time Messages, lists the messages that RMS-11 macros can generate at assembly time.

- Appendix C, Macros That Declare Symbols and Other Macros, describes RMS-11 macros that declare other RMS-11 macros and define RMS-11 symbols.
- Appendix D, RMS-11 with Different Operating Systems, describes the differences among the behaviors of RMS-11 with various operating systems.
- The index includes a major entry for each RMS-11 macro, control block field mnemonic, keyword macro argument, and symbol family.

ASSOCIATED DOCUMENTS

- PRO/RMS-11: An Introduction introduces RMS-11 to new users.

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are used in statement formats in this document:

UPPERCASE	Uppercase characters within a string indicate characters that you must include in the string; you can type the characters in uppercase or lowercase.
lowercase	Lowercase characters within a string indicate a user-selected variable; text following the statement format defines the syntax of the variable.
[ ]	Square brackets indicate that the enclosed string is optional user input.
...	A horizontal ellipsis indicates that the immediately preceding optional string (enclosed in square brackets) may be repeated.
other	A nonalphabetic character (except a square bracket or a period that is part of an ellipsis) indicates a character that you must include in the string.

Numbers in this manual that give the values of RMS-11 symbols are in octal radix (base 8) unless otherwise indicated; all other numbers in this manual are in decimal radix (base 10).

SUMMARY OF TECHNICAL CHANGES

This revision contains the following technical changes:

- The new operation macros $ENTER, $PARSE, $REMOVE, $RENAME, and $SEARCH are documented, along with the related NAM block fields FNB, RSA, RSL, RSS, and WCC.

- The new facility for wildcard file specification is documented.

- Random access to a sequential file with fixed-length records (similar to random access to a relative file) is documented.

- The new "print" record-output handling is documented, along with the related symbol FB$PRN for the RAT field of the FAB.

- The new sequential block access is documented; the previous block access (formerly called block I/O) is now called VBN access (virtual block number access).

- The addition of the success handler facility for file operation macros ($CLOSE, $CREATE, $DISPLAY, $ERASE, $EXTEND, and $OPEN) is documented.

- The obsolete RMS-11 initialization macros $INIT and $INITIF are no longer documented. These macros are now defined as no-ops in the RMS-11 macro library RMSMAC.MLB; their previous functions are no longer needed because RMS-11 is now self-initializing. However, programs that use the $INIT and $INITIF macros in their previous senses remain valid under RMS-11 Version 2.0.

- Each XAB type now has a distinct name; the following are the new names:

ALL block	Area allocation XAB
DAT block	File date XAB
KEY block	File key XAB
PRO block	File protection XAB
SUM block	File summary XAB

- The following symbol declaration macros are documented:

FAB$BT	Declare FAB value and mask symbols
NAM$BT	Declare NAM block value and mask symbols
RAB$BT	Declare RAB value and mask symbols
XAB$BT	Declare XAB value and mask symbols
XBAOF$	Declare ALL block symbols
XBDOF$	Declare DAT block symbols
XBKOF$	Declare KEY block symbols
XBPOF$	Declare PRO block symbols
XBSOF$	Declare SUM block symbols

- The description of each operation macro includes the use and meaning of each associated control block field.

- The value of each RMS-11 user symbol is documented.
- The structure of each RMS-11 user control block is documented.

CHAPTER 1

INTRODUCTION TO RMS-11 WITH MACRO-11

RMS-11 macros and symbols provide access to RMS-11 operations from a MACRO-11 program.

1.1 ADVANTAGES OF USING RMS-11 MACROS

When you use RMS-11 operations from a high-level language, the language restricts your options for some operations. If you cannot accept these restrictions, you can write your program (or some of its modules) in MACRO-11; this allows you full access to RMS-11 options.

1.2 RMS-11 MACROS AND SYMBOLS

RMS-11 macros and symbols define the interface between a MACRO-11 program and RMS-11 operation routines. Definitions for these macros and symbols are in the RMS-11 macro library, RMSMAC.MLB.

RMS-11 macros allow your program to:

- Call RMS-11 operations
- Declare and manipulate control blocks, through which your program communicates with RMS-11 operation routines
- Declare and manipulate space pools
- Declare needed RMS-11 facilities
- Extract (from the macro library RMSMAC.MLB) definitions for RMS-11 macros and symbols

The following sections introduce RMS-11 macros and symbols.

1.2.1 Operations

An RMS-11 operation macro calls a routine that performs an RMS-11 operation. The name of an operation macro is the name of the corresponding operation, with a prefixed dollar sign ($). The following are the RMS-11 operation macros:

Directory Operation Macros	File Operation Macros	Stream Operation Macros	Record Operation Macros	Block Operation Macros
$ENTER	$CLOSE	$CONNECT	$DELETE	$READ
$PARSE	$CREATE	$DISCONNECT	$FIND	$WRITE
$REMOVE	$DISPLAY	$FLUSH	$GET	
$RENAME	$ERASE	$FREE	$PUT	
$SEARCH	$EXTEND	$REWIND	$TRUNCATE	
	$OPEN		$UPDATE	

An RMS-11 operation returns a value called a completion code that indicates either a successful operation or an error. RMS-11 completion symbols give names to these completion codes.

When your program uses an RMS-11 operation macro to call an operation routine, it can specify completion handlers (one for a successful completion, one for an error completion) that RMS-11 calls when the operation completes. The RMS-11 completion-return macro ($RETURN) generates a proper return from a completion handler to the calling point in your program.

1.2.2 Control Blocks and Fields

Your program and RMS-11 operation routines communicate by passing data in blocks called control blocks. Each control block is divided into fields; each field has a 3-letter mnemonic name.

An RMS-11 block-declaration macro allocates space for a control block and initializes fields containing the block length and block identifier. There is a block-declaration macro for each kind of control block.

An RMS-11 field-initialization macro sets an initial value for a control block field at assembly time. There are field-initialization macros for most control block fields (those that you might reasonably want to initialize).

An RMS-11 field-access macro manipulates the value of a control block field during program execution. There are field-access macros for copying values to and from fields ($STORE and $FETCH), for comparing field values with other values ($COMPARE), and for setting, clearing, and testing bits in fields ($SET, $OFF, and $TESTBITS).

RMS-11 code and mask symbols give names to the codes and bit masks used in many fields. This allows your program to determine the details of an RMS-11 operation without using the numeric values associated with those details.

RMS-11 field-offset symbols give names to the locations of fields within their control blocks. Because RMS-11 field-initialization and field-access macros are based on field names, your program need not use field-offset symbols.

RMS-11 control blocks and their general uses are as follows:

- ALL (area allocation) block - contains information about a file area.
- DAT (file date) block - contains file dates and the file revision number.
- FAB (file access block) - contains general information about a file and how a program will access it.
- KEY (file key) block - contains information about a file index and its key.
- NAM (file name) block - contains special information about the device, directory, and specification for the file, along with wildcarding information.
- PRO (file protection) block - contains file owner and protection information.
- RAB (record access block) - contains general information about a stream and a record or block, and how the program accesses the record or block.
- SUM (file summary) block - contains the number of areas and indexes in the file, and a version number indicating the internal structure level of the file.

1.2.3 Pools

RMS-11 conserves space by dynamically allocating and deallocating space set aside in pools. RMS-11 pool-declaration macros allocate space for pools.

An RMS-11 routine called the get-space routine handles pooled space. You can substitute your own get-space routine for the RMS-11 routine; you can use RMS-11 get-space-address macros to initialize the address of the get-space routine at assembly time (GSA$), to change the address to that of a different routine during program execution ($SETGSA), and to return the address of the current routine during program execution ($GETGSA).

1.2.4 Facilities

The RMS-11 facilities-declaration macro (ORG$) assists RMS-11 in determining exactly which routines your program needs during program execution.

1.2.5 Macros That Declare Symbols and Other Macros

To extract the definition of an RMS-11 macro from the macro library, your program must declare the macro in a .MCALL assembler directive.

Many RMS-11 macros declare related macros and define related symbols; some RMS-11 macros have the sole purpose of declaring related macros and defining related symbols. Using these macros simplifies the job of declaring macros and defining symbols in your program.

For example, the FAB-declaration macro FAB$B declares FAB field-initialization macros and FAB offset, code, and mask symbols; the $FBCAL macro declares all directory and file operation macros; the $RMSTAT macro declares all completion symbols.

CHAPTER 2

RMS-11 PROGRAMMING

. To use RMS-11 operations in a MACRO-11 program, your program must:

- **Declare RMS-11 macros and symbols**

  Before your program refers to an RMS-11 macro or symbol, it must extract its definition from the RMS-11 macro library. Section 2.1 shows how to declare macros and symbols.

- **Declare RMS-11 facilities**

  To help RMS-11 decide which RMS-11 program modules are needed for your program, your program must declare some of the RMS-11 operations that it uses. Section 2.2 shows how to declare RMS-11 facilities.

- **Declare and use pool space**

  RMS-11 dynamically allocates and deallocates space for some of its requirements; this space is separated into five pools. Using RMS-11 pool-declaration macros, you specify the size of each pool. Section 2.3 shows how to declare pool space.

- **Declare and initialize control blocks**

  Your program and RMS-11 operation routines communicate by passing data back and forth in control block fields. Using RMS-11 block-declaration and field-initialization macros, your program allocates space for control blocks and (optionally) assigns initial values for fields. Section 2.4 shows how to declare and initialize control blocks.

- **Use RMS-11 operations**

  Your program uses RMS-11 operation routines to perform record management services; the routines return values that show the results of the operations. Your program uses RMS-11 operation macros to call these operation routines. Section 2.5 shows how to call RMS-11 operation routines and how to handle returns from the routines.

Your program may also:

- **Include completion handlers**

  An RMS-11 operation routine returns either a success completion code or an error completion code. Your program can include special routines (called success handlers and error handlers) that operation routines call automatically when operations complete. Section 2.6 shows how to write completion handlers.

- **Use its own get-space routines**

  RMS-11 uses a routine (called a get-space routine) to allocate and deallocate space. RMS-11 has a get-space routine, but you can also supply others of your own. Section 2.7 shows how to use get-space routines and how to write a get-space routine.

Finally, you must:

- **Assemble the program**

  When you assemble your program, it needs macro and symbol definitions from RMS-11; these are in a macro library, which your assembler command line must reference. RMS-11 macros detect some kinds of errors during assembly, and print messages that identify the errors. Section 2.8 shows how to assemble your program.

- **Build the task**

  When you build your task, you must use an RMS-11 resident library.

2.1 DECLARING RMS-11 MACROS AND SYMBOLS

Before your program refers to an RMS-11 macro or symbol, it must extract its definition from the RMS-11 macro library.

Your program can use the .MCALL assembler directive to extract the definition of any RMS-11 macro (but not a symbol) from the macro library. For example, to extract the definition of the macro $CLOSE, use the .MCALL directive in the format:

```
.MCALL  $CLOSE                  ;Declare RMS-11 $CLOSE macro
```

Your program can use RMS-11 macros to extract definitions for RMS-11 symbols, and for some groups of other RMS-11 macros. Appendix C lists RMS-11 macros (with their arguments) that declare symbols and other macros.

2.2 DECLARING RMS-11 FACILITIES

To help RMS-11 decide which RMS-11 program modules your program needs, your program declares some of the operations that it uses. To do this, it uses the facilities-declaration macro ORG$ in the format:

```
.MCALL ORG$                     ;Declare ORG$ macro
ORG$ fileorg[,<operation[,operation]...>]
```

where **fileorg** is a keyword indicating a file organization and each **operation** is a keyword indicating an operation that your program uses for a file of that organization.

A separate ORG$ macro is required for each different file organization that your program processes, except that no ORG$ macro is required for an organization that will be processed using only directory operations and block access.

The **fileorg** keyword argument to the ORG$ macro is one of the following:

IDX	Indexed file organization
REL	Relative file organization
SEQ	Sequential file organization

Each **operation** argument to an ORG$ macro is one of the following:

CRE	CREATE operation
DEL	DELETE operation
FIN	FIND operation
GET	GET operation
PUT	PUT operation
UPD	UPDATE operation

These are the only operations that your program explicitly declares with the ORG$ macro; support for other operations is handled automatically.

For example, suppose that your program:

- Creates both sequential and indexed files
- Uses FIND, GET, PUT, and UPDATE operations for sequential files
- Uses FIND, GET, PUT, and DELETE operations for indexed files

Then the proper ORG$ macros are:

```
ORG$    SEQ,<FIN,GET,PUT,UPD> ;Declare FIND, GET, PUT, and UPDATE
                              ; operations for sequential files
ORG$    IDX,<FIN,GET,PUT,DEL> ;Declare FIND, GET, PUT, and DELETE
                              ; operations for indexed files
```

The results of ORG$ macros are additive. For example, if one portion of your program specifies

```
ORG$    SEQ,<GET,PUT>
```

and another specifies

```
ORG$    SEQ,<GET,UPD>
```

then the effect is the same as specifying

```
ORG$    SEQ,<GET,PUT,UPD>
```

Note also that **all** ORG$ macros must occur in modules that are contained in the root segment of your task (not overlaid). Use of ORG$ macros is optional in tasks linked with an RMS-11 memory-resident library.

2.3 DECLARING AND USING POOL SPACE

RMS-11 dynamically allocates and deallocates space for some of its requirements; this space is separated into five pools:

- Internal FAB and index descriptor block (IFAB/IDB) pool
- Internal RAB (IRAB) pool

- Key buffer pool
- I/O buffer pool
- Buffer descriptor block (BDB) pool

RMS-11 has a get-space routine that manages these pools, and that allocates and deallocates space to meet the needs of RMS-11 operations; however, you can supply other get-space routines and direct RMS-11 to use a different routine (and, optionally, different pools) instead of its own.

If you use only the RMS-11 get-space routine, declare pool space using the pool-declaration macros described below. If you use your own get-space routine, read Section 2.7; it shows how to write the routine, and how to manage the pools.

To declare space for pools, use pool-declaration macros in the format:

```
POOL$B                                  ;Begin pool declarations
P$FAB   fabcount                        ;Space for IFABs in IFAB/IDB pool
P$IDX   indexcount                      ;Space for IDBs in IFAB/IDB pool
P$RAB   rabcount                        ;Space for IRABs for sequential
                                        ;  and relative files and for
                                        ;  block-accessed indexed files
                                        ;  in IRAB pool
P$RABX  rabxcount,keysize,keychanges ;Space for IRABs for
                                        ;  record-accessed indexed
                                        ;  files in IRAB pool, and
                                        ;  space for key buffers in
                                        ;  key buffer pool
P$BUF   bufcount                        ;Space for I/O buffers in I/O
                                        ;  buffer pool
P$BDB   bdbcount                        ;Space for BDBs in BDB pool
POOL$E                                  ;End pool declarations
```

If your program uses multiple pool declarations, the results are cumulative.

The following sections show how to compute the values of arguments to the pool-declaration macros.

2.3.1 Internal FAB and Index Descriptor Block Pool

Internal FABs (IFABs) and index descriptor blocks (IDBs) are the same size and so share a pool (the IFAB/IDB pool). The total size of the pool is the sum of the following:

- The largest number of IFABs that your program uses at the same time, times 48 bytes. Specify this largest number of IFABs (not multiplied by 48) as the **fabcount** argument to the P$FAB macro.

  A directory operation uses one IFAB, which is returned to the pool before the operation completes.

  A CREATE or OPEN operation uses one IFAB, which is committed while the file is open; a CLOSE operation releases the IFAB. A DISPLAY or EXTEND operation uses no new IFABs; it uses the IFAB already committed to the open file. An ERASE operation uses one IFAB, which is released before the operation completes.

- The largest number of IDBs that your program uses at the same time, times 48 bytes. Specify this largest number of IDBs (not multiplied by 48) as the **indexcount** argument to the P$IDX macro.

  Your program uses one IDB for each index of each indexed file opened for record access (rather than block access). The IDBs for an indexed file are committed when the file is opened (by a CREATE or OPEN operation) and are released when the file is closed (by a CLOSE operation).

2.3.2 Internal RAB Pool

Internal record access blocks (IRABs) have a separate pool. The size of the IRAB pool is the largest number of streams that your program will have connected at the same time, times the size of an IRAB (32 bytes).

Specify the largest number of streams connected to sequential files, relative files, and block-access indexed files (not multiplied by 32) as the **rabcount** argument to the P$RAB macro. Specify the largest number of streams connected to record-access indexed files as the **rabxcount** argument to the P$RABX macro.

If the sum of the **rabcount** and **rabxcount** arguments is larger than the largest number of streams that will ever be connected simultaneously, you may deduct the excess from the **rabcount** argument that you specify.

An IRAB is committed when a stream is connected and is released when the stream is disconnected or the file is closed (using the associated FAB).

2.3.3 Key Buffer Pool

Key buffers have a separate pool. (These key buffers are different from those specified by the KBF and KSZ fields of the RAB.)

Each time a stream is connected to an indexed file (for record access), the CONNECT operation requests space from the key buffer pool; the space is released when the stream is disconnected or the file is closed.

Compute the size (in bytes) of the request that the CONNECT operation makes as follows:

1. Begin with the size of the largest key for the file.
2. Multiply by 2.
3. Add the **number** of alternate keys for the file that are allowed to change during updating.
4. Add 1.
5. Round up (if necessary) to a multiple of 4.

If your program performs complex sequences of CONNECT and DISCONNECT (or CLOSE) operations for record-access indexed files with different key sizes, the key buffer pool may become fragmented (and therefore contain unusable space). In this case, the total size of the key buffer pool should be larger than the sum of the requirements for each connected stream.

Each P$RABX macro that your program uses (in the format P$RABX rabxcount,keysize,keychanges) allocates a number of bytes for the key buffer pool that is equal to

(rabxcount) x ((keysize * 2) + keychanges + 1)

The expression ((keysize * 2) + keychanges + 1) is rounded up (if necessary) to a multiple of 4.

You can use P$RABX macros to precisely tailor the size of the key buffer pool, or to provide extra space against possible fragmentation problems. A good compromise is to choose the arguments to the P$RABX macro as follows:

- Choose **rabxcount** as the largest number of streams that will be connected to record-access indexed files.
- Choose **keysize** as the largest key in any file that will be processed.
- Choose **keychanges** as the maximum number of changeable keys in any file that will be processed.

2.3.4 I/O Buffer Pool

The I/O buffers for RMS-11 operations come either from the central buffer pool or from a private buffer pool. (These are RMS-11 internal I/O buffers, and are different from the I/O buffers specified in the RBF, RSZ, UBF, and USZ fields of the RAB.)

Your program can specify a private buffer pool for a directory or file operation (except CLOSE, DISPLAY, or EXTEND). If your program does not specify a private buffer pool, these operations use the central buffer pool.

All other operations that require I/O buffers use the same pool as the CREATE or OPEN operation that opened the file.

The minimum size of the central I/O buffer pool is the sum of the sizes of the I/O buffers that your program will need from it at the same time (ignoring I/O buffers supplied from private buffer pools). Specify the size (in bytes) of the central buffer pool as the **iopoolsize** argument to the P$BUF macro.

Specify the size (in bytes) of a private buffer pool for an operation in the 1-word BPS field of the FAB and the address in the 1-word BPA field of the FAB. If your program specifies a private buffer pool for a CREATE or OPEN operation, the entire pool is reserved for and managed by that file until the file is closed.

Your program needs space from buffer pools for the following:

- One 512-byte I/O buffer for any directory or file operation (except CLOSE, DISPLAY, or EXTEND). This space is released before the operation completes.

- One 512-byte I/O buffer for a DISPLAY or EXTEND operation for a record-access relative or indexed file; the space is returned when the operation completes.

- I/O buffers for a CONNECT operation:

  - One I/O buffer for a record-access stream connected to a sequential disk file. The I/O buffer uses 512 bytes times the multiblock count for the stream.

  - One I/O buffer for a record-access stream connected to a file on a unit-record device. The number of bytes in the I/O buffer is equal to the default block size for the device, rounded up (if necessary) to a multiple of 4 bytes.

  - One or more I/O buffers for a stream connected to a relative file. Each I/O buffer uses 512 bytes times the bucket size for the file. If you use the multibuffer count to specify additional buffers, the requirement increases accordingly.

  - Two or more I/O buffers for a stream connected to an indexed file. Each I/O buffer uses 512 bytes times the bucket size for the file. If you use the multibuffer count to specify additional buffers, the requirement increases accordingly.

  I/O buffers for a connected stream are retained until the stream is disconnected by a DISCONNECT or CLOSE operation.

If your program uses the I/O buffer pool for complex sequences of operations that use I/O buffers for different files, the pool may become fragmented. In that case, you may want to either allocate extra space in the I/O buffer pool, or limit fragmentation through the judicious use of private buffer pools.

2.3.5 Buffer Descriptor Block Pool

Your program requires one 20-byte buffer descriptor block (BDB) for each I/O buffer (whether from the central or a private pool) that it uses at the same time; these BDBs are allocated and returned at the same time as their associated I/O buffers. (I/O buffer requirements are described in the previous section.)

In addition, a block-access stream (for any file) or a record-access stream that will write to a relative file requires an additional BDB; a record-access stream that will write to an indexed file requires two additional BDBs. These BDBs are returned when the stream is disconnected (or the file is closed).

An EXTEND operation for a record-access relative or indexed file also requires an additional BDB, which is returned when the operation completes.

Therefore the size of the BDB pool is the largest number of BDBs required at any one time, times 20 bytes. Specify this largest number of BDBs (not multiplied by 20) as the **bdbcount** argument to the P$BDB macro.

2.4 DECLARING AND INITIALIZING CONTROL BLOCKS

Your program and RMS-11 operation routines communicate by passing data back and forth in control block fields. Using RMS-11 block-declaration and field-initialization macros, you allocate space for control blocks and (optionally) assign initial values for fields.

To declare a control block and initialize its fields, use block-declaration and field-initialization macros as follows:

1. Make sure the control block is word-aligned by using the .EVEN directive:

```
.EVEN                   ;Word-align block
```

2. Specify a label so that your program can refer symbolically to the address of the control block.

```
label:
```

3. Begin the block declaration with one of the following macros:

```
FAB$B                   ;Begin FAB declaration
NAM$B                   ;Begin NAM block declaration
RAB$B                   ;Begin RAB declaration
XAB$B   XB$ALL          ;Begin ALL block declaration
XAB$B   XB$DAT          ;Begin DAT block declaration
XAB$B   XB$KEY          ;Begin KEY block declaration
XAB$B   XB$PRO          ;Begin PRO block declaration
XAB$B   XB$SUM          ;Begin SUM block declaration
```

4. Initialize (optionally) fields with field-initialization macros of one of the forms:

```
F$fld   arg             ;Initialize FAB field
N$fld   arg             ;Initialize NAM block field
R$fld   arg             ;Initialize RAB field
X$fld   arg             ;Initialize XAB field
```

In each of these forms, **fld** is the mnemonic for a field in the control block; **arg** is an argument suitable for the value of the field. Chapter 6 describes field-initialization macros and their arguments.

5. End the block declaration with one of the following macros:

```
FAB$E                   ;End FAB declaration
NAM$E                   ;End NAM block declaration
RAB$E                   ;End RAB declaration
XAB$E                   ;End XAB declaration
```

2.5 USING RMS-11 OPERATIONS

Your program uses RMS-11 operation routines to perform record management services. Using RMS-11 operation macros, you call these operation routines. The routines return values in control block fields that show the results of the operations.

To use RMS-11 operation routines, your program must:

- **Set up control block fields**

  The values that your program places in control block fields specify the details of the service you want from the RMS-11 operation routine. Section 2.5.1 shows how to set up control block fields.

- **Chain control blocks**

  Some RMS-11 operation routines (stream, record, and block operation routines) read only RAB fields; others (directory and file operation routines) read FAB fields and, if your program supplies them, fields in NAM blocks and XABs. Your program chains these blocks (using address pointers) so that the operation routine can find them. Section 2.5.2 shows how to chain control blocks.

- **Call operation routines**

  You use RMS-11 operation macros to call RMS-11 operation routines. Section 2.5.3 shows how to call operation routines.

- **Handle returns**

  Section 2.5.4 shows how to handle returns from operation routines.

- **Examine returned values**

  When an RMS-11 operation routine completes its execution, it has placed values in control block fields that show the results of the operation. Your program should examine these values to determine the results. Section 2.5.5 shows how to examine returned values.

2.5.1 Setting Up Control Block Fields

The values that your program places into control block fields specify the details of the service you want from the RMS-11 operation routine. The description of each operation macro in Chapter 5 discusses the control block fields that are read by that operation.

Three RMS-11 field-access macros help you place values into control block fields:

- $STORE places a specified value into a field.
- $SET sets bits in a field.
- $OFF clears bits in a field.

2.5.1.1 **$STORE Macro** - Use the $STORE macro to copy a value from a specified location to a control block field. The format for the $STORE macro is:

```
$STORE src,fld,reg
```

where **src** is a an address in memory; **fld** is a field mnemonic; and **reg** is a general purpose register (R0 through R5) containing the address of the control block.

The $STORE macro looks up the size of the destination field, so that it can copy the correct number of bytes or words. If the source is a register and the destination is a 1-byte field, then the low byte of the register is copied; if the source is a register and the destination is a multiword field, then the contents of the specified register and following registers are copied.

The $STORE macro generates an error during assembly if you use an illegal address mode for the source. For multiword fields, illegal address modes are autoincrement deferred, autodecrement deferred, and indexed deferred.

It is also illegal to specify the program counter (PC) as the source or to specify a register as source in such a way that the source overlaps the register that contains the control block address.

At execution time, the $STORE macro copies the contents of the specified location to the control block field. The number of bytes or words copied is the same as the field size for the mnemonic. Chapter 6 gives the size of each control block field.

For example, suppose that you want to specify indexed file organization in the FAB for a file, and suppose that the address of that FAB is stored in register R2. Then the proper macro is:

```
$STORE    #FB$IDX,ORG,R2      ;Indexed file organization
```

Suppose that you want to chain a NAM block whose label is NAMBLK to the same FAB. Then the proper macro is:

```
$STORE    #NAMBLK,NAM,R2      ;Chain NAM block
```

Suppose that you want to set the allocation quantity (ALQ field) of the same FAB to the value stored in a location labeled ALQVAL. Then the proper macro is:

```
$STORE    ALQVAL,ALQ,R2       ;Load allocation quantity
```

and (because ALQ is a 2-word field) two words are copied from ALQVAL to the ALQ field.

2.5.1.2 **$SET Macro** - Use the $SET macro to set bits in a 1-byte or 1-word control block field. The $SET macro logically ORs a given mask into the control block field. Therefore for each bit set in the mask, the $SET macro sets the corresponding bit in the field; the other bits are not changed.

Note that you use the $SET macro only if you want to leave some bits in a field undisturbed; if you want to set specified bits and clear all others, use the $STORE macro.

The format for the $SET macro is:

```
$SET mask,fld,reg
```

where **mask** is an address in memory containing bits to be set; **fld** is the mnemonic for a control block field; and **reg** is a general purpose register (R0 through R5) containing the address of the control block.

If the field is not a 1-byte or 1-word field, the $SET macro generates an error during assembly.

RMS-11 has symbols for masks for each bit-oriented control block field. Therefore your program can use these symbols instead of numerical values.

For example, suppose you want to specify rewind-on-close in the FAB for a file, but do not want to disturb other bits in the FOP field of the FAB; suppose also that the address of the FAB is in register R2. Then the proper macro is:

```
$SET #FB$RWC,FOP,R2             ;Rewind-on-close
```

As another example, suppose you want to specify key-duplicates-allowed and key-changes-allowed for an index, but do not want to disturb other bits in the FLG field of the KEY block; suppose also that the address of the KEY block is in register R4. Then the proper macro is:

```
$SET #XB$DUP!XB$CHG,FLG,R4      ;Allow key duplicates and changes
```

2.5.1.3 **$OFF Macro** - Use the $OFF macro to clear bits in a 1-byte or 1-word control block field. The $OFF macro logically ANDs the 1's complement of a given mask into the control block field. Therefore for each bit set in the mask, it clears the corresponding bit in the field; the other bits are not changed.

Note that you use the $OFF macro only if you want to leave some bits in a field undisturbed; if you want to clear the entire field, use the $STORE macro (with a source value of #0).

The format for the $OFF macro is:

```
$OFF mask,fld,reg
```

where **mask** is an address in memory containing bits to be cleared; **fld** is the mnemonic for a control block field; and **reg** is a general purpose register (R0 through R5) containing the address of the control block.

If the field is not a 1-byte or 1-word field, the $OFF macro generates an error during assembly.

RMS-11 has symbols for masks for each bit-oriented control block field. Therefore your program can use these symbols instead of numerical values.

For example, suppose you want to specify no-rewind-on-close in the FAB for a file, but do not want to disturb other bits in the FOP field of the FAB; suppose also that the address of the FAB is in register R2. Then the proper macro is:

```
$OFF #FB$RWC,FOP,R2             ;No rewind-on-close
```

As another example, suppose you want to specify no-key-duplicates-allowed and no-key-changes-allowed for an index, but do not want to disturb other bits in the FLG field of the KEY block; suppose also that the address of the KEY block is in register R4. Then the proper macro is:

```
$OFF #XB$DUP!XB$CHG,FLG,R4     ;No key duplicates or changes
```

2.5.2 Chaining Control Blocks

An RMS-11 directory operation or file operation uses at least one FAB; you specify FABs in the operation macros that call the operation routines.

For some directory operations, a NAM block is required; it is optional for other directory operations and for file operations. You specify a NAM block and XABs for an operation by chaining them to the FAB for the operation.

2.5.2.1 Chaining a NAM Block to a FAB - Specify the NAM block associated with a FAB by placing its address in the 1-word NAM field of the FAB.

2.5.2.2 Chaining XABs to a FAB - Specify the XABs associated with a FAB by placing the address of the first XAB in the 1-word XAB field of the FAB; in each XAB, specify the address of the next XAB in the chain in the 1-word NXT field of the XAB; in the last XAB in the chain, specify 0 in the NXT field.

Follow these rules in ordering XABs in a chain:

- Place ALL blocks together in the chain. Each ALL block is "numbered" by the value in the 1-byte AID field of the ALL block; chain ALL blocks so that these numbers are in ascending order. For the CREATE operation, begin with 0 and do not skip numbers in the ascending sequence; for other operations, you can skip numbers in the sequence.
- Place no more than one DAT block in the chain.
- Place KEY blocks together in the chain. Each KEY block is "numbered" by the value in the 1-byte REF field of the KEY block; chain KEY blocks so that these numbers are in ascending order. For the CREATE operation, begin with 0 and do not skip numbers in the ascending sequence; for other operations, you can skip numbers in the sequence.
- Place no more than one PRO block in the chain.
- Place no more than one SUM block in the chain.

2.5.2.3 Chaining a FAB to a RAB (CONNECT Operation) - The CONNECT operation creates a stream for a file. A FAB specifies the file; a RAB specifies the stream. Specify the address of the FAB for the file in the 1-word FAB field of the RAB for the stream.

2.5.3 Calling Operation Routines

Use RMS-11 operation macros to call operation routines. You can specify arguments for the operation routine either by giving them as arguments to the operation macro, or by placing them in an argument block in memory.

2.5.3.1 Call with Macro Arguments - Call an operation routine (except RENAME) using an operation macro with arguments in the format:

```
$macroname blkaddr[,[erraddr][,sucaddr]]
```

where **$macroname** is the name of an operation macro (except $RENAME); **blkaddr** is the address of a FAB (for a directory or file operation) or a RAB (for a stream, record, or block operation); **erraddr** is the address of an error handler for the operation; and **sucaddr** is the address of a success handler for the operation.

For example, if you want to open a file using a FAB at address INFAB and want to use a success handler at address SUCCES, the macro call would be:

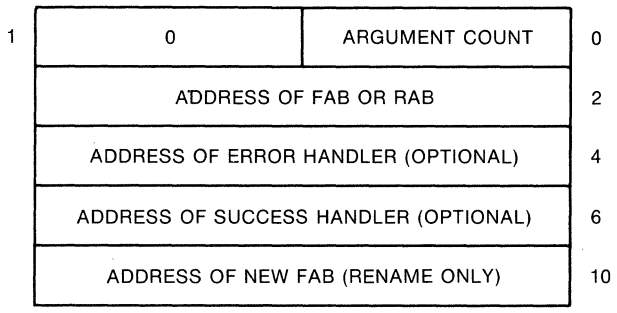
```
$OPEN #INFAB,,#SUCCES
```

Call the RENAME operation using the $RENAME operation macro with arguments in the format:

```
$RENAME oldfabaddr,[erraddr],[sucaddr],newfabaddr
```

where **oldfabaddr** is the address of a FAB for the old file specification; **erraddr** is the address of an error handler for the operation; **sucaddr** is the address of a success handler for the operation; and **newfabaddr** is the address of a FAB for the new file specification.

2.5.3.2 Call with Arguments in Memory - To call an operation routine using an operation macro with arguments in an argument block in memory, omit the arguments to the macro, store the address of the argument block in register R5, and store the argument block in memory as follows:



ZK-1097-82

The argument count is 4 for a RENAME operation; otherwise it is one of the following:

- 1 - no completion handlers
- 2 - error handler, but no success handler
- 3 - success handler

If the operation has no error handler, but either has a success handler or the operation is RENAME, specify -1 as the address of the error handler; if the operation has no success handler, but the operation is RENAME, specify -1 as the address of the success handler.

2.5.4 Handling Returns

An RMS-11 file or directory operation returns a completion status code in the 1-word STS field of the FAB and, for some completions, a completion status value in the 1-word STV field of the FAB.

An RMS-11 stream, record, or block operation returns a completion status code in the 1-word STS field of the RAB and, for some completions, a completion status value in the 1-word STV field of the RAB.

Appendix A lists completion codes.

Your program should examine the STS field contents to determine whether the operation was successful; even if the operation returned an error completion, your program may be able to handle the error and recover.

The program can handle the return (based on the completion code) either in the code that immediately follows the operation macro, or in special routines (called completion handlers) that the operation can call. Section 2.6 shows how to write completion handlers.

There are two kinds of fatal RMS-11 errors:

- If the FAB or RAB address you specify is not the address of a valid and idle FAB or RAB, or if the argument block you provide is invalid, RMS-11 cannot return values, even in the STS field. RMS-11 issues a BPT instruction, leaving status information in the following registers:

  R0: RMS-11 fatal error code
  R1: Stack pointer (at time of entry to RMS-11 routine)
  R2: Program counter (entry return same as @R1)
  R3: Address of system impure area

- If RMS-11 detects the corruption of memory-resident data structures, or if it detects inconsistent internal states, it cannot proceed with its operations. In these cases, RMS-11 halts execution with a BPT instruction; if it can identify the error, RMS-11 leaves an error completion in R0.

Appendix A lists the symbols and values for RMS-11 fatal error codes.

2.5.5 Examining Returned Values

When an RMS-11 operation routine completes its execution, it has placed values in control block fields that show the results of the operation. Your program should examine these values to determine the results. The description of each operation macro in Chapter 6 discusses the control block fields that return values for that operation.

Three RMS-11 field-access macros help you examine values in control block fields:

- $FETCH copies a value from a field to a specified location.
- $COMPARE compares a field value to a specified value.
- $TESTBITS determines whether specified bits in a field are set.

2.5.5.1 **$FETCH Macro** - Use the $FETCH macro to copy a value from a control block field to a specified location. The format for the $FETCH macro is:

```
$FETCH dst,fld,reg
```

where **dst** is an address in memory; **fld** is the mnemonic for a control block field; and **reg** is a general purpose register (R0 through R5) containing the address of the control block.

The $FETCH macro looks up the size of the source field, so that it can copy the correct number of bytes or words. If the destination is a register and the source is a 1-byte field, then the byte is copied to the low byte of the register and the high byte is cleared. if the destination is a register and the source is a multiword field, then the multiword field is copied to the specified register and following registers.

The $FETCH macro generates an error during assembly if you use an illegal address mode for the destination. For multiword fields, illegal address modes are autoincrement deferred, autodecrement deferred, and indexed deferred. Immediate mode is illegal for $FETCH, regardless of field size.

It is also illegal to use the program counter (PC) as the destination or to specify a register for the destination in such a way that the destination overlaps the register that contains the control block address.

At execution time, the $FETCH macro copies the contents of the control block field to the specified location. The number of bytes or words copied is the same as the field size for the mnemonic. Chapter 6 gives the size of each control block field.

As an example of the use of the $FETCH macro, suppose that you want to fetch the allocation quantity (ALQ field) from a FAB to a location labeled ALQSAV, and suppose also that the address of the FAB is in register R3. Then the proper macro is:

```
$FETCH  ALQSAV,ALQ,R3          ;Save allocation quantity
```

and two words are copied from the ALQ field to memory beginning at ALQSAV.

2.5.5.2 **$COMPARE Macro** - Use the $COMPARE macro to compare the contents of a 1-byte or 1-word control block field with a specified value. The format for the $COMPARE macro is:

```
$COMPARE src,fld,reg
```

where **src** is an address in memory; **fld** is the mnemonic for a control block field; and **reg** is a general purpose register (R0 through R5) containing the address of the control block.

If the given field is not a 1-byte or 1-word field, the $COMPARE macro generates an error during assembly.

At execution time, the $COMPARE macro executes a machine instruction that compares the source value and the field contents. The instruction executed depends on the size of the specified field and on the specified source:

- TSTB for a 1-byte field and the source #0
- TST for a 1-word field and the source #0
- CMPB for a 1-byte field and a source other than #0
- CMP for a 1-word field and a source other than #0

Chapter 6 gives the size of each control block field.

For example, suppose that you want to compare the value in the RSZ field of a RAB with a value stored in a location labeled RSZSAV, and suppose also that the address of the RAB is stored in register R2. Then the proper macro is:

```
$COMPARE RSZSAV,RSZ,R2          ;Compare record size
```

Suppose that you want to compare the same RSZ field to the value of a symbol, RECSIZ. Then the proper macro is:

```
$COMPARE #RECSIZ,RSZ,R2         ;Compare record size
```

2.5.5.3 **$TESTBITS Macro** - Use the $TESTBITS macro to test the values of bits in a 1-byte or 1-word control block field. Chapter 6 gives the size of each control block field. The format for the $TESTBITS macro is:

```
$TESTBITS mask,fld,reg
```

where **mask** is an address in memory containing bits to be tested; **fld** is the mnemonic for a control block field; and **reg** is a general purpose register (R0 through R5) containing the address of the control block.

If the given field is not a 1-byte or 1-word field, the $TESTBITS macro generates an error during assembly.

At execution time, the $TESTBITS macro executes a machine instruction that tests the bits specified in the mask. The instruction executed depends on the size of the specified field:

- BITB for a 1-byte field
- BIT for a 1-word field

For example, suppose you want to determine whether the terminal device is set in the DEV field of a FAB, and suppose that the address of the FAB is in register R3 Then the proper macro is:

```
$TESTBITS #FB$TRM,DEV,R3        ;Terminal device?
```

As another example, suppose that you want to determine whether either the contiguous-area or the hard-location bit is set in the AOP field of an ALL block, and suppose that the address of the ALL block is in register R2. Then the proper macro is:

```
$TESTBITS #XB$CTG!XB$HRD,AOP,R2 ;Contiguous or hard location?
```

2.6 WRITING COMPLETION HANDLERS

Recall that when you use an RMS-11 operation macro, you can specify the addresses of completion handlers for the operation; if you do so, the operation automatically calls the error handler (for a nonfatal error completion) or the success handler (for a success completion). After the completion handler executes, control is returned to the point immediately following the operation macro.

When execution control passes to your completion handler, it finds the following situation:

- Register R5 contains the address of the argument block for the operation.
- The second word of the argument block contains the address of the FAB or RAB for the operation. (Recall that the STS and STV fields of the FAB or RAB contain the completion code and completion value for the operation.)
- If the operation was RENAME, the fifth word of the argument block contains the address of a second FAB for the operation.
- Other blocks are chained as they were when you used the operation macro that called the operation routine.

A completion handler cannot determine from these values which RMS-11 operation was executed, or what part of your program called the operation routine. You can, however, use the 1-word CTX field of the FAB or the 1-word CTX field of the RAB to indicate the context of the operation; RMS-11 does not disturb values in CTX fields.

The completion handler must preserve the stack pointer (SP), and must end with the RMS-11 completion-return macro in the format:

```
$RETURN                         ;End of completion handler
```

2.7 USING GET-SPACE ROUTINES

Your program can provide and use get-space routines other than the one provided with RMS-11. It can set an initial get-space routine at assembly time, and it can change to other routines during program execution. Section 2.7.1 shows how to specify get-space routines, and how to obtain the address of the current get-space routine. Section 2.7.2 shows how to write a get-space routine.

2.7.1 Specifying Get-Space Routines

To specify a get-space routine at assembly time, use the GSA$ macro in the format:

```
GSA$    address                 ;Initialize get-space routine
                                ;  address
```

where **address is the get-space routine entry address.** If you specify 0 as the address, or if you do not use the GSA$ macro, the initial get-space routine for the program is the RMS-11 routine.

To change the get-space routine during program execution, use the $SETGSA macro in the format:

```
$SETGSA pointer                 ;Change get-space routine
```

where **pointer is the address of a location that contains the get-space routine entry address.** If you specify the entry-point address as 0, the new get-space routine established is the RMS-11 routine.

To obtain the address (in R0) of the current get-space routine during program execution, use the $GETGSA macro in the format:

```
$GETGSA                         ;Get-space routine address into R0
```

If the address returned in R0 is 0, the current get-space routine is the RMS-11 routine.

2.7.2 Writing a Get-Space Routine

A get-space routine handles space in contiguous blocks. For a request for space, it allocates a contiguous block of space (or denies the request); for a release of space, it accepts a contiguous block of space.

A get-space routine must have a proper interface to calling routines, and it should handle unallocated space properly.

2.7.2.1 Get-Space Routine Interface - When RMS-11 calls a get-space routine, it either requests or releases a block of space. For a request for space, registers R0 through R2 contain the following values:

```
R0   Address of pool free-space list (see next section)
R1   Size (in bytes) of requested block
R2   0
```

If the get-space routine fills the request, it must clear the C bit and return the address of the first word of the allocated block in R0; if it does not fill the request, it must set the C bit. In either case, the routine must preserve the stack and registers R3 through R6.

For a release of a block of space, registers R0 through R2 contain the following values:

```
R0   Address of pool free-space list (see next section)
R1   Size (in bytes) of released block
R2   Address of first word being released
```

For a release-space operation, the get-space routine returns no values; however, it must preserve the stack and registers R3 through R6.

2.7.2.2 **Pool Free-Space Lists** - When RMS-11 calls your get-space routine, the address of a pool free-space list is in register R0. This free-space list specifies free space in one of the five pools described in Section 2.3; you can use this pool (which may or may not have adequate free space), or you can use a pool of your own.

The free-space list chains free contiguous blocks of the pool. The first word of each block contains the address of the next block; if the first word of a block is 0, it is the last block in the list.

Blocks in the list are ordered by ascending virtual addresses; their addresses are word-aligned; their sizes are multiples of 4 bytes (allocations and deallocations must be rounded up to a multiple of 4, if necessary).

The second word of each block contains the size (in bytes) of the block, including the 4-byte header; the first "block" in the list contains 0 in its second word, since it is the header block for the list.

Your get-space routine can use the specified pool list to get space for RMS-11; if it does this, it must properly maintain the list, and must (if possible) merge blocks back into the pool.

The system routines $RQCB and $RLCB are suitable for handling pool free-space lists. These routines have interfaces that meet the requirements for your get-space routine; therefore your program can jump to $RQCB (for a space request) or $RLCB (for a space release).

2.8 ASSEMBLING THE PROGRAM

When you assemble your program, you must cause the assembler to get RMS-11 macro and symbol definitions from a library, and you may have to correct errors indicated by messages from RMS-11 macros.

2.8.1 Assembling with the RMSMAC Macro Library

When you assemble your program, the assembler needs definitions for the RMS-11 macros and symbols that your program uses; these are in the RMS-11 macro library, RMSMAC.MLB. Include the following reference to the RMS-11 macro library in your assembler command string:

```
LB:[1,1]RMSMAC.MLB/ML
```

2.8.2 Assembly-Time Errors from RMS-11 Macros

RMS-11 macros detect some errors during assembly. For each such error, a macro issues a .PRINT or .ERROR assembler directive with a message. Appendix B describes RMS-11 macro-generated messages and their meanings.

CHAPTER 3

PROCESSING DIRECTORIES AND FILES

This chapter discusses use of RMS-11 directory and file operations. The next sections discuss information and usage common to several directory and file operations:

- Device characteristics
- Logical channels
- File specifications and identifiers
- Private buffer pools
- Completion status

The sections after those provide an overview of the operations themselves (see Chapter 5 for detailed discussions):

- Directory operations (except SEARCH): ENTER, REMOVE, RENAME and PARSE
- File operations: CREATE, OPEN, DISPLAY, ERASE, EXTEND, and CLOSE

Finally, the last sections discuss:

- SEARCH operation
- Writing wildcard loops

3.1 DEVICE CHARACTERISTICS

A directory or file operation (except CLOSE, DISPLAY, or EXTEND) returns device characteristics. These characteristics are returned as masks in the 1-byte DEV field of the FAB. The device characteristics are:

- Printer or terminal (indicated by the set FB$CCL mask in the 1-byte DEV field of the FAB and the set FB$REC mask in the 1-byte DEV field of the FAB; for a terminal, the FB$TRM mask in the 1-byte DEV field of the FAB is also set); RMS-11 treats a printer or terminal as a unit-record device.
- Disk, DECtape, or DECTAPE II (indicated by the set FB$MDI mask in the 1-byte DEV field of the FAB); RMS-11 treats a disk, DECtape, or DECTAPE II as a disk device.

- Unit-record device (indicated by the set FB$REC mask in the 1-byte DEV field of the FAB).
- Non-ANSI magtape or cassette tape (indicated by the set FB$SDI mask in the 1-byte DEV field of the FAB and the set FB$REC mask in the 1-byte DEV field of the FAB); RMS-11 treats a non-ANSI magtape or a cassette tape as a unit-record device.
- ANSI-format magtape (indicated by the set FB$SQD mask in the 1-byte DEV field of the FAB).

3.2 LOGICAL CHANNELS

An RMS-11 directory or file operation (except CLOSE, DISPLAY, or EXTEND) requires a logical channel; this channel is a path from the program to a specified device.

When your program executes a CREATE or OPEN operation on the channel, the path is extended to the target file; until the file is closed, the channel is reserved for the specified FAB.

Your program specifies the logical channel for a directory operation or for a CREATE, ERASE, or OPEN operation in the 1-byte LCH field of the FAB; the channel must not already be in use by the task.

You can specify the initial device assignment for a logical channel in a Task Builder command file. The Task Builder also provides default initial device assignments for certain channels. Other logical channels are unassigned when your task begins executing.

During task execution, channel assignments are made or changed by use of the ALUN$ system directive. For example, RMS-11 uses the ALUN$ directive to assign a logical channel for a directory operation or for a CREATE, ERASE, or OPEN operation; if the FAB and NAM block specify a device or device identifier, RMS-11 assigns the channel to that device; if the FAB and NAM block do not specify a device or device identifier, RMS-11 retains the device-channel assignment (if any), or assigns the channel to the device SY:.

3.3 FILE SPECIFICATIONS AND IDENTIFIERS

A file specification consists of the following elements (in the order given):

- Device specification - the device where the file resides
- Directory specification - the directory on the device through which the file can be found
- File name - the name by which the file is known in the directory
- File type - the type by which the file is known in the directory
- File version - the version number by which the file is known in the directory

RMS-11 operations construct and use file specification strings and file identifiers to specify files. These strings and identifiers include:

- User-provided file specification strings
- Expanded file specification strings
- Resultant file specification strings
- File, directory, and device identifiers

This section discusses these strings and identifiers as they are used for nonwildcard operations; wildcard use is described in Section 3.8.

For a CREATE, ENTER, ERASE, OPEN, PARSE, REMOVE, or RENAME operation, your program specifies two strings to be used in generating a full file specification:

- A file specification string, called the **file string** (your program specifies the address of the file string in the 1-word FNA field of the FAB and the length of the string in the 1-byte FNS field of the FAB)
- A default file specification string, called the **default string** (your program specifies the address of the default string in the 1-word DNA field of the FAB and the length of the string in the 1-byte DNS field of the FAB)

The operation routine uses these two strings to form an internal merged file specification string, called the **merged string.** The operation initially forms the merged string as follows:

- It begins by taking available elements from the file string.
- It then supplies missing elements from the default string (if they are available there). The operation (when it completes) returns masks describing the results of this merge in the 1-word FNB field of the NAM block (if you supplied a NAM block for the operation).

If elements are still missing from the merged string, the operation next adds the following elements:

- Device - If the logical channel specified in the LCH field of the FAB is already assigned to a device, that device is used; otherwise the device SY: is used.
- Directory - The task's current default directory is used.
- File name, type, and version - Nulls are used.

If the operation is the PARSE operation, the merged string is complete. If you provided a NAM block, the PARSE operation returns the device identifier in the 2-word DVI field of the NAM block; if you provided an expanded string buffer, the PARSE operation returns the expanded string in the expanded string buffer (whose address is in the 1-word ESA field of the NAM block). (Note that the device specification in an expanded string has usually been translated to the specification for a physical device.)

An operation other than PARSE continues by examining the FB$FID mask in the FOP field of the FAB. If the FB$FID mask is set, the operation adds the following elements:

- Device - If a device identifier is given in the NAM block, that device overrides the device in the merged string and the device specification is deleted from the merged string.
- Directory - If a directory identifier is given in the NAM block, that directory overrides the directory in the merged string and the directory specification is deleted from the merged string.
- File identifier - If a file identifier is given in the NAM block and if the operation is ERASE or OPEN, that file overrides the directory, file name, type, and version in the merged string and the specifications for those elements are deleted from the merged string.

The merged string is then copied to the expanded string buffer (if you supplied one) as described for the PARSE operation above. The merged string plus applicable identifiers are called the **fully qualified file specification,** and define the file upon which the operation will be performed.

The device, directory, and file identifiers for the file are returned in the NAM block (if you supplied one). These identifiers can be used as input to subsequent directory and file operations to speed processing by eliminating directory and file lookups.

Note that a complete file specification is relevant only to a disk file. Only the device specification is relevant for a file on a unit-record device. Irrelevant elements are not processed, and appear in the expanded string only if your program provides them in the file string or default string.

NOTE: NULL, 0, OR -1 VERSION NUMBER

If the version specification has not been deleted and is null, 0, or -1, it will later be replaced with the version number of the target file.

A version number of -1 identifies the target file as the (otherwise) specified file with the lowest version number; a version number of -1 is illegal for a CREATE or ENTER operation, or for the new file specification for a RENAME operation.

For an ERASE, OPEN, REMOVE, or RENAME (old specification) operation, a null or 0 version number specifies the target file as the (otherwise) specified file with the highest version number.

For a CREATE, ENTER, or RENAME (new specification) operation, a null or 0 version number specifies that the operation is to create a new entry whose version number is one greater than the highest-numbered version of the (otherwise) specified file.

3.4 PRIVATE BUFFER POOLS

Many RMS-11 operations require space from a buffer pool. A directory or file operation (except CLOSE, DISPLAY, or EXTEND) allows your program to specify a private buffer pool. Your program specifies the address of the pool in the 1-word BPA field of the FAB; it specifies the size (in bytes) of the pool in the 1-word BPS field of the FAB.

The CLOSE operation returns (in the BPA and BPS fields) the address and size of the private buffer pool (if any) specified for the CREATE or OPEN operation that opened the file; until the file is closed, the pool is dedicated to the open file and must not be used for other purposes.

If your program does not specify a private buffer pool, the operation uses the central buffer pool (which your program declares using pool-declaration macros); if your program specifies a private buffer pool, the operation uses that pool.

The CLOSE, DISPLAY, and EXTEND operations, and all stream, record, and block operations use the pool specified by the CREATE or OPEN operation that opened the file.

3.5 COMPLETION STATUS

A directory or file operation returns a completion status code in the 1-word STS field of the FAB, and a completion status value in the 1-word STV field of the FAB.

3.6 DIRECTORY OPERATIONS

RMS-11 directory operations affect only directory entries (not the contents of files). The directory operations are:

- ENTER: create a directory entry
- REMOVE: delete a directory entry
- RENAME: replace a directory entry
- PARSE: analyze a file specification
- SEARCH: search directories

The next sections provide an overview of the directory operations (except for the SEARCH operation, which is discussed in Section 3.8).

3.6.1 ENTER Operation

A file specified as temporary when it was created has no directory entry; a file also has no directory entry if the entry has been deleted by the REMOVE operation.

Your program can use the ENTER operation to create a directory entry for a file; this makes it possible for your program (and other programs) to specify the file to RMS-11 by its file specification.

The ENTER operation uses the device and directory parts of the fully qualified file specification to determine the target directory; it then creates an entry in that directory using the file name, type, and

version from the fully qualified file specification, and the file identifier specified in the NAM block.

3.6.2 REMOVE Operation

Your program can delete the directory entry for a file by using the REMOVE operation; this does not affect either the existence of the file or the file contents, but only removes the path to the file.

The device and directory elements from the fully qualified file specification specify the target directory; the file name, type, and version elements from the fully qualified file specification identify the entry to be removed from the directory.

3.6.3 RENAME Operation

Your program can replace the directory entry for a file by using the RENAME operation. The fully qualified file specification for the new directory entry must not specify a new device for the file, but otherwise it can specify elements different from the old file specification: directory, file name, file extension, and file version number.

If you do not specify a device, the device associated with the old file specification is used.

For both the old and new directory entries, the RENAME operation uses the device and directory elements of the fully qualified file specification to determine the target directory; it uses the file name, type, and version elements of the fully qualified file specification to identify the entry to be removed or created.

3.6.4 PARSE Operation

Your program can use the PARSE operation to analyze a file specification, or to prepare for a series of wildcard operations (described in Section 3.8). The results of the PARSE operation are described in detail in Section 3.3.

3.7 FILE OPERATIONS

RMS-11 file operations affect files as whole entities (but not individual records or blocks in files). The file operations are:

- CREATE: create a file (and a corresponding directory entry) and open the file for processing
- OPEN: open an existing file for processing
- DISPLAY: write file information to control blocks
- ERASE: delete file contents (records or blocks) and remove directory entry

- EXTEND: increase the allocation for a file
- CLOSE: close an open file

The next sections discuss file operations.

3.7.1 CREATE Operation

The CREATE operation creates a new file and opens it for processing; unless the file is specified as a temporary file, the CREATE operation also creates a directory entry for the file.

The CREATE operation uses the device and directory elements of the fully qualified file specification to determine the target directory; it then uses the file name, type, and version of the fully qualified file specification to form the entry in that directory.

3.7.2 OPEN Operation

Your program can establish an access path to a file by using the OPEN operation. This makes file information available to your program, and enables your program to use the following operations for the file:

- DISPLAY operation (to make more file information available to your program).
- EXTEND operation (to allocate more space for the file).
- CONNECT operation (to establish a path to file records or blocks). The CONNECT operation enables your program to use other stream operations and either record operations or block operations.
- CLOSE operation (to release resources committed to the open file). The CLOSE operation terminates the access path established by the CREATE or OPEN operation that opened the file.

3.7.3 DISPLAY Operation

If your program uses the OPEN operation to open a file, but does not provide control blocks and buffers for all the information that the OPEN operation can return, you may want to use the DISPLAY operation to obtain additional information while the file is open.

3.7.4 ERASE Operation

Your program can erase the contents of a file by using the ERASE operation, and (optionally) remove its directory entry.

Unless your program provides a file identifier in the NAM block and sets the FB$FID mask in the 1-word FOP field of the FAB, the ERASE operation also removes the specified directory entry for the file.

The ERASE operation uses the fully qualified file specification to determine the target file. If the operation removes the directory entry, it uses the device and directory elements of the fully qualified file specification to determine the target directory, and

the file name, type, and version elements to determine the entry to be removed.

3.7.5 EXTEND Operation

Your program can increase the allocation for an open file by using the EXTEND operation. Note that RMS-11 automatically extends the file allocation when it needs more space; you can use the EXTEND operation to make large extensions (avoiding repeated automatic extensions) or exact extensions (avoiding wasteful automatic extensions).

3.7.6 CLOSE Operation

Your program can close an open file by using the CLOSE operation. This releases task and system resources (other than the file itself) and makes those resources available for other uses.

3.8 WRITING WILDCARD LOOPS

You can include wildcard characters in an RMS-11 file specification and use the PARSE and SEARCH operations to identify files that match the wildcard specification. This allows you to program a wildcard loop that successively (and selectively, if you wish) processes files matching the wildcard specification.

An advantage of RMS-11 wildcarding over system wildcard commands is that your processing can be selective. For example, if you use a system wildcard command to rename a group of files, the entire group is renamed; if you use a wildcard loop in a program, the program can fully examine information about each file and even the contents of each file to decide whether to rename it.

The next three sections show:

- The structure of a wildcard loop and the behavior of directory and file operations in the loop
- How to write a wildcard loop that nonselectively uses the ERASE, REMOVE, or RENAME operation on successive matching files
- How to write a wildcard loop that selectively performs directory and file operations on successive matching files

3.8.1 Introduction to Wildcarding

This discussion assumes that you want to write a program loop that uses a wildcard input file specification, and that you want to use the same control blocks (FAB and NAM block) for all operations associated with the wildcard loop.

A series of wildcard operations can be viewed as having four steps:

1. Initializing for wildcarding
2. Finding the next matching file

3. Operating on the found file

4. Ending wildcarding

The next sections discuss these steps.

3.8.1.1 Initializing for Wildcarding - The PARSE operation initializes control blocks (FAB and NAM block) for wildcard operations. Place the $PARSE macro before the wildcard loop in your program.

The PARSE operation sets the NB$WCH mask in the 1-word FNB field of the NAM block to show that wildcard operations are in progress. (Your program must clear the NB$WCH mask if it will not perform SEARCH operations after a PARSE operation.)

The PARSE operation also forms a match-pattern in the expanded string buffer (whose address is in the 1-word ESA field of the NAM block); this match-pattern is used by subsequent wildcard SEARCH operations.

A series of SEARCH operations requires a NAM block that specifies both expanded string and resultant string buffers. (The resultant string buffer is specified in the 1-word RSA field of the NAM block.) Your program must not alter the expanded string, the resultant string, or other NAM block contents between the PARSE operation and the end of the subsequent series of SEARCH operations.

3.8.1.2 Finding the Next Matching File - The SEARCH operation finds the next file (if any) that matches the wildcard input file specification. (If the SEARCH operation cannot find another matching file, wildcarding ends; see Section 3.8.1.4.)

The SEARCH operation returns a fully qualified file specification in the resultant string buffer, along with device, directory, and file identifiers for the found file.

The SEARCH operation in your wildcard loop can either be explicit (your loop contains the $SEARCH macro) or, for some operations, implicit (RMS-11 automatically performs the SEARCH operation). If you use the explicit SEARCH operation, place the $SEARCH macro inside the loop but before other operation macros.

If you use an ERASE, REMOVE, or RENAME (old FAB) operation in the loop with the FB$FID mask in the 1-word FOP field of the FAB cleared, RMS-11 implicitly performs a SEARCH operation (to find the next matching file) before performing the ERASE, REMOVE, or RENAME operation. This allows your wildcard loop to omit the $SEARCH macro. (If the implicit SEARCH operation cannot find another matching file, wildcarding ends; see Section 3.8.1.4.)

3.8.1.3 Operating on the Found File - A number of directory and file operations are wildcard-transparent in the sense that they preserve both wildcard context information and information about the last-found file. This means that your program can use the operations within a wildcard loop without changing the wildcard context; the series of wildcard operations is continuable.

These wildcard-transparent operations are CLOSE, DISPLAY, and EXTEND, and (if the FB$FID mask in the 1-word FOP field of the FAB is set) ERASE, OPEN, REMOVE, and RENAME (old FAB).

3.8.1.4 Ending Wildcarding - A series of wildcard operations (using a specific FAB and NAM block) ends when a directory or file operation discards wildcard context information or when your program clears the NB$WCH mask in the 1-word FNB field of the NAM block.

Typically, the operation that ends wildcarding is a SEARCH operation that cannot find another matching file. It returns the ER$NMF completion status code and clears the NB$WCH mask in the 1-word FNB field of the NAM block.

If your program exits from a wildcard loop before the SEARCH operation fails to find a matching file, the NB$WCH mask in the 1-word FNB field of the NAM block is still set, and your program must clear it.

Executing the PARSE operation during a wildcard series ends that series and initializes control blocks for a new series.

Executing a CREATE or ENTER operation, or an OPEN operation with the FB$FID mask in the 1-word FOP field of the FAB cleared, ends the wildcard series for that FAB.

3.8.2 Nonselective ERASE, REMOVE, or RENAME Wildcard Operations

You can write a wildcard loop that performs nonselective ERASE, REMOVE, or RENAME operations on successive matching files, where RMS-11 implicitly performs a SEARCH operation before each ERASE, REMOVE, or RENAME operation.

To do this, do the following:

1. Use the PARSE operation to initialize control block fields for wildcarding.
2. Clear the FB$FID mask in the 1-word FOP field of the FAB (for the RENAME operation, the old FAB). This causes the ERASE, REMOVE, or RENAME operation to perform an implicit SEARCH operation before performing its own processing.
3. Use the ERASE, REMOVE, or RENAME operation to operate on the next matching file.
4. Examine the STS field of the FAB. If it contains the ER$NMF completion status code, there was not another matching file; in that case, go to step 7.
5. Perform other in-loop processing (such as reporting the file specification of the erased, removed, or renamed file).
6. Go to step 2.
7. The wildcard series is finished; continue with other processing.

The following program segment illustrates this procedure, performing the ERASE operation. In the program segment, FABADR is a label giving the address of the FAB for the operations, and R0 is used (for the $STORE and $COMPARE macros) to contain the address of the FAB.

```
        $PARSE    #FABADR          ;Set up for wildcarding

LOOP:   MOV       #FABADR,R0       ;FAB address to R0
        $STORE    #0,FOP,R0        ;Use implicit search
                                   ;  (FB$FID off)
        $ERASE    #FABADR          ;Try to erase next file
        $COMPARE  #ER$NMF,STS,R0   ;Was there a matching file?
        BEQ       DONE             ;No more matching files
        . . .                      ;Other in-loop processing
        BR        LOOP             ;On to next matching file

DONE:   . . .                      ;Continue with other
                                   ;  processing
```

3.8.3 Selective Wildcard Operations

You can write a wildcard loop that performs directory and file operations on selected matching files, where your program explicitly performs a SEARCH operation at the beginning of each iteration of the loop. To do this, do the following:

1. Use the PARSE operation to initialize control block fields for wildcarding.

2. Use the SEARCH operation to obtain information about the next file that matches the wildcard specification.

3. Examine the STS field of the FAB. If it contains the ER$NMF completion status code, there was not another matching file; in that case, go to step 6.

4. Perform directory and file operations on the found file. If ERASE, OPEN, REMOVE, or RENAME operations are included, be sure the FB$FID mask in the 1-word FOP field of the FAB (for the RENAME operation, the old FAB) is set.

   Do not perform CREATE, ENTER, or PARSE operations, or OPEN operations with the FB$FID mask cleared; these operations end wildcarding.

   Do not perform ERASE, REMOVE, or RENAME operations with the FB$FID mask cleared; these operations perform an implicit SEARCH operation, advancing to the next matching file.

5. Go to step 2.

6. The wildcard series is finished; continue with other processing.

The following program segment illustrates the procedure, performing the ERASE operation on selected files. In the program segment, FABADR

is a label giving the address of the FAB for the operations, and R0 is used (for the $COMPARE macro) to contain the address of the FAB.

```
            $PARSE    #FABADR           ;Set up for wildcarding

LOOP:       $SEARCH   #FABADR           ;Find next matching file
            MOV       #FABADR,R0        ;FAB address to R0
            $COMPARE  #ER$NMF,STS,R0    ;Any more matching files?
            BEQ       DONE              ;No more matching files
            . . .                       ;Decide whether to delete
                                        ;  file (if so, Z-bit on)
            BNE       NOOP              ;Don't delete file
            MOV       #FABADR,R0        ;FAB address to R0
            $SET      #FB$FID,FOP,R0    ;Explicit SEARCH already done
            $ERASE    #FABADR           ;Erase file contents
NOOP:       . . .                       ;Other in-loop processing
            BR        LOOP              ;On to next matching file

DONE:       . . .                       ;Continue with other
                                        ;  processing
```

CHAPTER 4

PROCESSING RECORDS AND BLOCKS

This chapter describes use of RMS-11 stream, record, and block operations; its major sections are:

- Completion status
- Streams
- Record processing
- Block processing

4.1 COMPLETION STATUS

A stream, record, or block operation returns a completion status code in the 1-word STS field of the RAB; it may also return a completion status value in the 1-word STV field of the RAB.

4.2 STREAMS

A stream is a path from your program to the data in a file. The CONNECT operation establishes a stream; for the CREATE or OPEN operation that opened the file, your program specified either record access or block access.

If it specified record access, the stream is a record stream and supports only stream operations and record operations; if it specified block access, the stream is a block stream and supports only stream operations and block operations.

For the CONNECT operation, your program specifies the FAB for the file (in the 1-word FAB field of the RAB), and the CONNECT operation returns an internal stream identifier (in the 1-word ISI field of the RAB). All stream, block, and record operations (except CONNECT) identify the file using the internal stream identifier; the DISCONNECT operation terminates the stream, and clears the internal stream identifier.

4.3 RECORD PROCESSING

This section describes use of RMS-11 record processing. Its subsections are:

- Record streams: the paths from your program to file records
- Record context: the "current location" of a stream in a file
- Record access modes: the ways your program can access records
- Record buffers: the locations of records in your program's space
- Locate mode: a way of speeding record processing
- Stream operations: stream operations for a record stream
- Record operations: operations that access records

4.3.1 Record Streams

A record stream is a path from your program to the records in a file. Your program establishes a record stream when it uses the CONNECT operation to connect a stream to a file (opened for record access by an earlier CREATE or OPEN operation). A record stream supports stream operations and record operations, but not block operations.

If the target file for a stream is a relative or indexed file, your program can establish more than one stream for the file; if, in addition, your program specifies access sharing, more than one task can establish streams for the file.

4.3.2 Record Context

A record stream has a record context, which consists of a current-record context and a next-record context. Some record operations use the current record or next record as the target for the operation; some stream and record operations change the current-record context, the next-record context, or both.

The notion of "following record" is important to record context because the next-record context is often established as the record "following" the current record. The precise meaning of "following record" depends on the file organization:

- In a sequential file, the record following a given record is the one immediately following it in physical sequence.
- In a relative file, the record following a given record is the one in the first higher-numbered cell that contains a record.
- In an indexed file, a record follows another only with respect to an index; each index imposes an order on the file records. The record following a given record (under a given index) is the record whose record key is the smallest in the file that is greater than the record key of the given record; among records having identical record keys, a record written later follows a record written earlier.

Note that although an operation may establish the next-record context, that context is not evaluated until another operation uses it. For example, if your program connects a stream to a relative file that contains records only in cells 5 and 10, a sequential access GET operation returns the record in cell 5 and establishes both current-record and next-record context; if another stream or task then inserts a record in cell 7 before your program executes a second sequential access GET operation, that GET operation returns the new record (cell 7), even though the record did not exist when the next-record context was established.

4.3.3 Record Access Modes

The record operations FIND, GET, and PUT allow your program to specify a record access mode (in the 1-byte RAC field of the RAB); the record access mode determines the target record for the operation. The record access modes are:

- Sequential access
- Key access
- RFA access

The next sections discuss these access modes.

4.3.3.1 Sequential Access - Your program specifies sequential access by setting the RB$SEQ code in the 1-byte RAC field of the RAB. A sequential access FIND or GET operation has as its target the next record. (Exception: a sequential access GET operation that immediately follows any FIND operation has as its target the current record, which is the record found by the FIND operation.)

The target of a sequential access PUT operation depends on the file organization, as follows:

- For a sequential file, a series of sequential access put operations must begin with the next-record context at the end-of-file. The series of PUT operations adds new records at the end-of-file.
- For a relative file, a series of sequential access PUT operations must begin with the next-record context set such that the first cell examined is empty (unless the RB$UIF mask in the 1-word ROP field of the RAB is set. The series of PUT operations adds new records in successive cells; if a nonempty cell is encountered, the PUT operation returns the ER$REX completion (unless the RB$UIF mask is set, in which case the existing record is overwritten).
- For an indexed file, a series of sequential access PUT operations does not depend on the next-record context; however, a PUT operation in the series returns the ER$SEQ completion if the value of the record primary key for the operation is less than the value of the record primary key for the preceding PUT operation.

A sequential access FIND or GET operation sets the current-record context to the target record, and sets the next-record context to the record following the target record. Sequential access PUT operations leave both the current-record and next-record contexts undefined.

This targeting and context setting means, generally speaking, that a series of sequential access operations operates on successive records. Specifically, series of sequential access operations result as follows:

- A series of sequential access FIND operations sets the stream context to successive records.
- A series of sequential access GET operations reads successive records.
- A series of sequential access PUT operations writes successive records (for an indexed file, possibly interspersed with existing records).
- A series of paired sequential access FIND and sequential access GET operations reads successive records.

4.3.3.2 Key Access - Your program specifies key access by setting the RB$KEY code in the 1-byte RAC field of the RAB. A key access FIND, GET, or PUT operation has as its target the record that your program specifies by specifying the key. For a relative file or for a sequential disk file with fixed-length records, your program specifies the key as a relative record number. Specify the relative record number in the 1-word KBF field of the RAB and the key size as 0 or 4 in the 1-byte KSZ field of the RAB.

For a FIND or GET operation for an indexed file, your program specifies the index of reference and a key buffer that contains the record key. Specify the index of reference in the 1-byte KRF field of the RAB, the address of the key buffer in the 1-word KBF field of the RAB, and the key size in the 1-byte KSZ field of the RAB.

A key access FIND or GET operation sets the current-record context to the record that is the target of the operation; a key access PUT operation leaves the current-record context undefined.

A key access FIND or PUT operation does not affect the next-record context; a key access GET operation sets the next-record context to the record following the target record.

The target of a key access FIND, GET, or PUT operation depends on the operation and on the file organization:

- For a relative file or for a sequential disk file with fixed-length records, the key is a positive integer and specifies the position of the record in the file. This key is the relative record number (RRN) for the record; RRN 1 specifies the first record, and so forth.

  If your program sets the RB$KGT mask in the 1-word ROP field of the RAB, a FIND or GET operation searches for a record whose RRN is greater than the given RRN; if it sets the RB$KGE mask in the 1-word ROP field of the RAB, the operation searches for a record whose RRN is greater than or equal to the given RRN; if it sets neither of these masks, the operation searches for a record with the given RRN.

Note that a FIND, GET, or PUT operation to a relative file or to a sequential disk file with fixed length records returns the RRN for the target record in the 2-word BKT field of the RAB.

- For a FIND or GET operation to an indexed file, the key specifies a record in the file whose record key matches the given key. Your program specifies both the key to be matched and the file index; the key data type must agree with the key data type for the index (string, packed decimal, binary, or signed integer).

  For a string key, your program specifies the portion of the key that must be matched. If the value in the 1-byte KSZ field of the RAB is nonzero but is smaller than the record key, then only that smaller initial portion of the key must match.

  If your program sets the RB$KGT mask in the 1-word ROP field of the RAB, a FIND or GET operation searches for a record whose key is greater than the given key; if it sets the RB$KGE mask in the 1-word ROP field of the RAB, the operation searches for a record whose key is greater than or equal to the given key; if it sets neither of these masks, the operation searches for a record whose key exactly matches the given key.

- For a PUT operation to an indexed file, the key (for each index) is in the record. The operation has no true target; the record is inserted at the proper place and each index is updated.

This targeting and context setting means that although the target of the key access operation is a random (selected) record, the record context allows subsequent sequential access processing. Therefore your program can use key access to "jump" to a selected point in a file, then use sequential access to process successive records.

4.3.3.3 RFA Access - Your program specifies RFA access by setting the RB$RFA code in the 1-byte RAC field of the RAB. An RFA access FIND or GET operation has as its target the record that your program specifies by RFA (record file address). (The FIND, GET, and PUT operations return the RFA for the target record; if your program saves the RFA, it can use RFA access for the record in subsequent FIND and GET operations.) Specify the RFA in the 3-word RFA field of the RAB.

An RFA access FIND or GET operation sets the current-record context to the record that is the target of the operation. An RFA access FIND operation does not affect the next-record context; an RFA access GET operation set the next-record context to the record following the target record.

This targeting and context setting means that although the target of the RFA access operation is a random (selected) record, the record context allows subsequent sequential access processing. Therefore your program can use RFA access to "jump" to a selected point in a file, then use sequential access to process successive records.

4.3.4 Record Buffers

A PUT or UPDATE operation transfers a record from a record buffer (in your program's space) to a file; for a VFC record, the operation also transfers the fixed-length portion of the record from a separate record header buffer. Your program specifies the address of the record buffer in the 1-word RBF field of the RAB and the size of the record in the 1-word RSZ field of the RAB; for a VFC record, your program also specifies the address of the record header buffer in the 1-word RHB field of the RAB.

A GET operation transfers a record from a file to an RMS-11 internal I/O buffer and to a user buffer in your program's space. Your program specifies the address of the user buffer in the 1-word UBF field of the RAB and its size in the 1-word USZ field of the RAB. Along with the record, the GET operation returns the address of the record in the 1-word RBF field of the RAB and its size in the 1-word RSZ field of the RAB.

For a VFC record, a GET operation also transfers the fixed-length portion of the record to a separate record header buffer in your program's space. Your program specifies the address of the record header buffer in the 1-word RHB field of the RAB.

Exception: if your program specifies locate mode for a GET operation, RMS-11 may not transfer the record to the user buffer; see the next section for a discussion of locate mode.

4.3.5 Locate Mode

The GET and PUT operations normally use RMS-11 internal I/O buffers as intermediate storage between your program's buffers (record or user buffers) and the file. By specifying locate mode for a GET or PUT operation, your program requests RMS-11 to transfer records only between its I/O buffers and the file, thus saving time. Your program specifies locate mode by setting the RB$LOC mask in the 1-word ROP field of the RAB.

If your program specifies locate mode for a GET operation, RMS-11 may transfer the record only to its internal I/O buffer (but not to the user buffer). The GET operation routine decides whether to honor the locate-mode request or to transfer the record to the user buffer anyway; the operation returns the address and size of the retrieved record (informing your program of the record's location -- the user buffer or the I/O buffer).

If your program specifies locate mode for a PUT operation, RMS-11 recognizes that the record may already be in its I/O buffer and if so transfers it to the file from there.

Your program has (in the 1-word RBF field of the RAB) the address of a location (in the I/O buffer if possible, otherwise in the user buffer) that is suitable for building the next record; this address is returned either by a previous locate-mode PUT operation or by an initial locate-mode CONNECT operation. Therefore, if you use the CONNECT operation for a stream that will use locate-mode PUT operations, your program must specify locate mode for the CONNECT operation, and must specify a user buffer (the address in the 1-word UBF field of the RAB and the size in the 1-word USZ field of the RAB).

Note that specifying locate mode for a PUT operation has no effect unless the file is sequential, the access mode is sequential, and the record format is other than stream record format.

4.3.6 Stream Operations

Stream operations affect stream context and I/O buffers (but not file records). The stream operations for a record stream are:

- CONNECT: establish a record stream
- FLUSH: write unwritten buffers for a stream
- FREE: free locked bucket for a stream
- NXTVOL: set stream context to beginning of next volume
- REWIND: set stream context to beginning of current volume
- DISCONNECT: terminate a record stream

The next sections discuss these operations.

4.3.6.1 CONNECT Operation - Your program uses the CONNECT operation to establish a record stream. (The stream is a record stream because your program specified record access for the CREATE or OPEN operation for the file.)

The current-record context after a CONNECT operation is undefined; the next-record context is (by default) the first record in the file.

For an indexed file, your program must specify an initial index of reference so that the record context is initialized properly.

For a sequential file, your program can specify that the initial record context is to be at the end-of-file (instead of the beginning of the file); in that case, the next-record context after the operation is the end-of-file.

For a sequential disk file, your program specifies the number of blocks in the I/O buffer for the stream; for a relative or indexed file, your program specifies the number of I/O buffers for the stream.

If the stream will use locate-mode PUT operations, your program must also specify locate mode and supply a user buffer. The CONNECT operation returns the address of a location suitable for building the first record to be output; see Section 4.3.5.

4.3.6.2 FLUSH Operation - Your program can use the FLUSH operation to write any unwritten buffers for a stream (for example, to increase data integrity by ensuring that all changes have been written to the file); the FLUSH operation does not affect record context, except that the current-record context is undefined for a following TRUNCATE or UPDATE operation to a sequential file.

Note one special case: if the file was opened for deferred writing, but not for write sharing, then the buffer may be controlled by another record stream and will not be written by the FLUSH operation.

4.3.6.3 FREE Operation - Your program can use the FREE operation to free a locked bucket for a stream; the FREE operation does not affect stream context, except that the current-record context is undefined for a following DELETE, TRUNCATE, or UPDATE operation.

4.3.6.4 REWIND Operation - Your program can use the REWIND operation to reset the context for a stream to the beginning-of-file.

The current-record context after the operation is undefined; the next-record context is the first record in the file; for an indexed file, your program specifies the index of reference for the operation so that the stream context is initialized properly.

4.3.6.5 DISCONNECT Operation - Your program can use the DISCONNECT operation to terminate a record stream, thus recovering the resources committed for the stream (primarily pool space). The DISCONNECT operation also discards record context and the internal stream identifier.

4.3.7 Record Operations

Record operations affect stream context, buffers (I/O, user, and record), and file records. The record operations are:

- FIND: transfer a record from a file to an I/O buffer
- GET: transfer a record from a file to an I/O buffer and to a user buffer
- PUT: transfer a record from a user buffer to a file
- DELETE: remove a record from a file
- UPDATE: replace a record in a file
- TRUNCATE: remove the current record and all following records from a sequential file

The next sections discuss these operations.

4.3.7.1 FIND Operation - Your program can use the FIND operation to transfer a record (or part of a record) from a file to an I/O buffer; the FIND operation does not transfer the record to a user buffer.

Your program specifies an access mode (sequential, key, or RFA) for the FIND operation; Section 4.3.3 describes the target record and context-setting for the FIND operation (Section 4.3.3.1 for sequential access, 4.3.3.2 for key access, and 4.3.3.3 for RFA access).

For a relative file or for a sequential disk file with fixed-length records, the FIND operation returns the relative record number (RRN) and the record file address (RFA) for the found record; for other files, the FIND operation returns only the RFA for the found record.

4.3.7.2 GET Operation - Your program can use the GET operation to transfer a record from a file to an I/O buffer and to a user buffer (which your program specifies).

Your program specifies an access mode (sequential, key, or RFA) for the GET operation; Section 4.3.3 describes the target record and context-setting for the GET operation (Section 4.3.3.1 for sequential access, 4.3.3.2 for key access, and 4.3.3.3 for RFA access).

The GET operation returns the address and size of the retrieved record, along with its RFA; for a relative file or for a sequential disk file with fixed-length records, the GET operation also returns the RRN for the retrieved record.

If your program specifies locate mode for the GET operation, it must also specify a user buffer; see Section 4.3.5.

4.3.7.3 PUT Operation - Your program can use the PUT operation to transfer a record from a user buffer to an I/O buffer and to a file.

Your program specifies an access mode (sequential or key) for the PUT operation; Section 4.3.3 describes the target record and context-setting for the PUT operation (Section 4.3.3.1 for sequential access, 4.3.3.2 for key access).

Your program can specify that RMS-11 must honor bucket fill numbers.

For an indexed file, your program can specify that each PUT operation in a series is part of a mass insertion; for a relative file, your program can specify that the PUT operation should overwrite the target record (if any).

The PUT operation returns the RFA for the inserted record; for a relative file or for a sequential disk file with fixed-lentgh records, the PUT operation also returns the RRN for the inserted record.

If your program specifies locate mode for the PUT operation, it must also specify a user buffer. The PUT operation returns the address of a location suitable for building the next output record; see Section 4.3.5.

4.3.7.4 DELETE Operation - Your program can use the DELETE operation to remove a record from a relative or indexed file. The target of a DELETE operation is the current record.

The current-record context after a DELETE operation is undefined; the next-record context is unchanged.

For an indexed file, your program can specify that RMS-11 must use the fast-deletion procedure. However, this procedure is faster because it deletes only those alternate index pointers that it must; future retrieval operations may be slowed by the presence of undeleted alternate index pointers.

4.3.7.5 UPDATE Operation - Your program can use the UPDATE operation to transfer a record from a user buffer to a file (overwriting the existing record). The target of the UPDATE operation is the current record, which is overwritten.

The current-record context after an UPDATE operation is undefined; the next-record context is unchanged.

Your program specifies the record buffer for the record to be inserted (and, for a VFC record, the VFC-header buffer).

4.3.7.6 TRUNCATE Operation - Your program can use the TRUNCATE operation to remove the current record and all following records (through the end-of-file) from a sequential file. The current-record context after a TRUNCATE operation is undefined; the next-record context is the new end-of-file.

4.4 BLOCK PROCESSING

This section describes use of RMS-11 block processing. Its subsections are:

- Block streams: the paths from your program to file blocks
- Block context: the "current location" of a stream in a file
- Block access modes: the ways your program can access blocks
- Block buffers: the locations of blocks in your program's space
- Stream operations: stream operations for a block stream
- Block operations: operations that access blocks

4.4.1 Block Streams

A block stream is a path from your program to the blocks in a file. Your program establishes a block stream when it uses the CONNECT operation to connect a stream to a file (opened for block access by an earlier CREATE or OPEN operation). A block stream supports stream operations and block operations, but not record operations.

4.4.2 Block Context

A block stream has a block context, which consists of a readable-block context and a writable-block context. The READ operation uses the readable-block as its target block; the WRITE operation uses the writable-block as its target block; block operations change both the readable-block and the writable-block contexts.

For a disk file, your program can use the READ or WRITE operation to read or write multiple blocks in a single operation. In that case, reading or writing begins at the readable block or the writable block (respectively), and continues through the number of blocks requested.

4.4.3 Block Access Modes

The block operations READ, SPACE, and WRITE allow your program to specify a block access mode (in the 2-word BKT field of the RAB); the block access mode determines the target block for the operation. The block access modes are:

- Sequential access
- VBN access

The next sections discuss these access modes.

4.4.3.1 Sequential Access - Your program specifies sequential block access by giving the value 0 in the 2-word BKT field of the RAB. A sequential access READ operation has as its target the readable block; it sets the readable-block context to the next-following unread block, and sets the writable-block context to the target block (first block read for that READ operation).

A sequential access WRITE operation has as its target the writable block; it sets both the readable-block and writable-block contexts to the next-following unwritten block.

This targeting and context setting has the following results:

- A series of sequential access READ operations reads successive blocks
- A series of sequential access WRITE operations writes successive blocks
- A series of paired READ and WRITE operations updates successive blocks

4.4.3.2 VBN Access - A VBN access READ or WRITE operation reads or writes blocks beginning with a virtual block that your program specifies. Specify the virtual block number in the 2-word BKT field of the RAB.

Note that your program can use VBN access to move to a random position in a disk file, and then use sequential block access to process blocks sequentially from that point.

4.4.4 Block Buffers

Your program specifies a user buffer for the READ operation; the operation returns the address of the first-read byte and the number of bytes read. Specify the address of the user buffer in the 1-word UBF field of the RAB and its size in the 1-word USZ field of the RAB; the READ operation returns the address of the first-read byte in the 1-word RBF field of the RAB and the number of bytes read in the 1-word RSZ field of the RAB.

Your program specifies the buffer containing the writable data for the WRITE operation. Specify the buffer address in the 1-word RBF field of the RAB and its size in the 1-word RSZ field of the RAB.

4.4.5 Stream Operations

Stream operations affect stream context and I/O buffers (but not file blocks). The stream operations for a block stream are:

- CONNECT: establish a block stream.
- FLUSH: write unwritten buffers for a stream.
- FREE: free a locked block for a stream.
- DISCONNECT: terminate a block stream.

The next sections discuss these operations.

4.4.5.1 CONNECT Operation - Your program uses the CONNECT operation to establish a block stream. (The stream is a block stream because your program specified block access for the CREATE or OPEN operation for the file.)

After a CONNECT operation, both the readable-block and writable-block contexts are the first block in the file.

4.4.5.2 FREE Operation - Your program can use the FREE operation to free a locked block for a stream; the FREE operation does not affect stream context.

4.4.5.3 DISCONNECT Operation - Your program can use the DISCONNECT operation to terminate a block stream, thus recovering the resources committed for the stream. The DISCONNECT operation also discards block context and the internal stream identifier.

4.4.6 Block Operations

Block operations affect stream context, block buffers, and file blocks. The block operations are:

- READ: transfer blocks from a file to a block buffer
- WRITE: transfer blocks from a block buffer to a file

The next sections discuss these operations.

4.4.6.1 READ Operation - Your program can use the READ operation to transfer blocks from a file to a block buffer. Your program specifies an access mode (sequential or VBN) for the READ operation; Section 4.4.3.1 describes sequential access; Section 4.4.3.2 describes VBN access.

4.4.6.2 WRITE Operation - Your program can use the WRITE operation to transfer blocks from a block buffer to a file. Your program specifies an access mode (sequential or VBN) for the WRITE operation; Section 4.4.3.1 describes sequential access; Section 4.4.3.2 describes VBN access.

Note that because the WRITE operation always write to the file immediately, the FLUSH operation has no use for block access.

CHAPTER 5

OPERATION MACRO DESCRIPTIONS

This chapter describes RMS-11 operation macros and the operation routines they call. Each section of the chapter describes an operation macro and its corresponding operation. (For the $FIND, $GET, $PUT, $READ, and $WRITE macros, there is a separate description for each access method.)

Each description is divided into the following parts:

- **FORMAT** - the format for the macro and its parameters
- **CONTROL BLOCKS** - the required and optional control blocks for the operation
- **OPTIONS** - the options that you can select for the operation, and the control block fields and values that control the options
- **STREAM CONTEXT** - the current-record and next-record contexts (for a record stream) or the readable-block and writable-block contexts (for a block stream) after the operation completes
- **RETURNED VALUES** - the values that the operation routine returns in control block fields and buffers
- **CHECKLISTS** - a list of the control block fields that you supply to specify options, and a list of the control block fields that contain returned values

The operation macros are:

- $CLOSE - Close an open file
- $CONNECT - Connect a record stream to an open file
- $CREATE - Create a new file and open it for processing
- $DELETE - Remove a record from a file
- $DISCONNECT - Disconnect a record stream
- $DISPLAY - Write file data into control block fields
- $ENTER - Enter a file specification into a directory
- $ERASE - Erase an existing file
- $EXTEND - Extend the allocation for an open file
- $FIND - Set the stream context to a record in a file

OPERATION MACRO DESCRIPTIONS

- $FLUSH - Write any unwritten buffers for a stream
- $FREE - Unlock a bucket locked by a stream
- $GET - Retrieve a record from a file
- $OPEN - Open an existing file
- $PARSE - Write file data into a NAM block
- $PUT - Insert a record into a file
- $READ - Read blocks from a file
- $REMOVE - Delete a file specification from a directory
- $RENAME - Rename an existing file
- $REWIND - Set stream context to beginning-of-file
- $SEARCH - Search directories for a file specification
- $TRUNCATE - Remove all following records from a file
- $UPDATE - Replace a record in a file
- $WRITE - Write blocks into a file

5.1 $CLOSE MACRO

The $CLOSE macro calls the CLOSE operation routine to close an open file.

FORMAT

The format for the $CLOSE macro is:

```
$CLOSE fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the CLOSE operation.

If you supply a PRO block, the CLOSE operation reads its fields to obtain new owner and protection codes for the file.

To supply XABs (ALL, DAT, KEY, PRO, and SUM blocks) for the CLOSE operation, specify the address of the first XAB in the 1-word XAB field of the FAB; specify the address of the next XAB (if any) in the 1-word NXT field of each XAB; specify 0 in the NXT field of the last XAB.

All KEY blocks must be together in the chain of XABs, and must be in ascending order (by the index reference number in the 1-byte REF field of the KEY block); the index reference numbers need not be consecutive.

All ALL blocks must be together in the chain of XABs, and must be in ascending order (by the area identifier in the 1-byte AID field of the ALL block); the area identifiers need not be consecutive.

Multiple DAT, PRO, or SUM XABs are illegal.

OPTIONS

Internal File Identifier

The CLOSE operation reads the internal file identifier for the file from the 1-word IFI field of the FAB. This identifier was written by the CREATE or OPEN operation when the file was opened.

File Owner and Protection

If you want to change the owner of the target file, specify the project (or group) portion of the owner code in the 1-word PRJ field of the PRO block, and specify the programmer (or member) portion in the 1-word PRG field of the PRO block; if you specify 0 for both these fields, the PRO block (including the PRO field) is ignored.

If you want to change the file protection for the target file, specify the protection code in the 1-word PRO field of the PRO block (and specify a nonzero value in the PRG or PRJ field); if you specify 0 in this field, the operating system uses its defaults.

Marking the File for Deletion

If you want the closed file marked for deletion, set the FB$MKD mask in the 1-word FOP field of the FAB; this causes the operating system to delete the file as soon as it has no accessing programs.

STREAM CONTEXT

The CLOSE operation destroys stream context for any streams connected by the closing file (after writing any unwritten buffers for those streams).

RETURNED VALUES

Private Buffer Pool

The CLOSE operation writes the address of the private buffer pool (if any) for the file in the 1-word BPA field of the FAB; if the CLOSE operation clears the BPA field, the file had no private buffer pool.

If the file had a private buffer pool, the CLOSE operation writes the size (in bytes) of the pool in the 1-word BPS field of the FAB.

Internal File Identifier

The CLOSE operation clears the 1-word IFI field of the FAB.

Completion Status and Value

The CLOSE operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-1 lists control block fields that are input to the CLOSE operation. Table 5-2 lists control block fields that are output by the CLOSE operation.

Table 5-1: CLOSE Input Fields

Block	Field	Description
ALL	AID	Area number
ALL	NXT	Next XAB address
DAT	NXT	Next XAB address
FAB	FOP	File processing option mask
		FB$MKD Mark file for deletion
		FB$RWC Rewind magtape after closing file
FAB	IFI	Internal FAB identifier
FAB	XAB	XAB address
KEY	REF	Index reference number
KEY	NXT	Next XAB address
PRO	NXT	Next XAB address
PRO	PRG	Programmer or member portion of file owner code
PRO	PRJ	Project or group portion of file owner code
PRO	PRO	File protection code
SUM	NXT	Next XAB address

Table 5-2: CLOSE Output Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	IFI	Internal FAB identifier
FAB	STS	Completion status code
FAB	STV	Completion status value

5.2 $CONNECT MACRO

The $CONNECT macro calls the CONNECT operation routine to connect a record stream to an open file, and initialize the stream context.

FORMAT

The format for the $CONNECT macro is:

```
$CONNECT rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the CONNECT operation.

You must supply a FAB for the CONNECT operation.

OPTIONS

File Identification

Specify the address of the FAB in the 1-word FAB field of the RAB. The CONNECT operation reads the internal file identifier for the file from the 1-word IFI field of the FAB.

I/O Buffers

For a sequential disk file, specify the size (in blocks) of the RMS-11 I/O buffer for the stream in the 1-byte MBC field of the RAB; the largest legal value is 63. If you specify 0, the CONNECT operation uses a buffer of one block. For a relative file, an indexed file, or a sequential nondisk file, the CONNECT operation ignores the MBC field.

For a relative or indexed file, specify the number of I/O buffers for the stream in the 1-byte MBF field of the RAB. For a sequential file, specify 0 in the MBF field. If you specify 0, the CONNECT operation uses the minimum number of buffers: one for a sequential or relative file, or two for an indexed file.

User Buffer (Locate Mode for Sequential File)

If you are connecting to a sequential file, and if you intend to execute PUT operations in locate mode for the connected stream, then:

- Specify the address of the user buffer in the 1-word UBF field of the RAB.
- Specify the size (in bytes) of the user buffer in the 1-word USZ field of the RAB.
- Set the RB$LOC mask in the 1-word ROP field of the RAB.

This assures proper handling of the first PUT operation for the stream.

Key of Reference (Indexed File)

For an indexed file, specify the key of reference in the 1-byte KRF field of the RAB. This value specifies the index to be used in establishing initial record context: 0 for the primary index, 1 for the first alternate index, and so forth.

Initial Stream Context (Sequential File)

If you want to initialize the next-record context of a sequential file to the end-of-file, set the RB$EOF mask in the 1-word ROP field of the RAB; if you do not set this mask, the CONNECT operation initializes the next-record context to the first record in the file (or to the end-of-file if the file is empty).

STREAM CONTEXT

For a record-access file, the current-record context after a CONNECT operation is undefined; the next-record context is the first record in the file (under the specified index for an indexed file), or the end-of-file, if the file is empty.

For a block-access file, both the readable-block and writable-block contexts after a CONNECT operation are the first block in the file.

RETURNED VALUES

Internal Stream Identifier

The CONNECT operation writes an internal stream identifier in the 1-word ISI field of the RAB. Do not destroy this identifier; all other stream, record, and block operation routines read it.

Record Buffer

The CONNECT operation copies the value from the UBF field into the 1-word RBF field of the RAB (the record address); this prepares the record buffer for your use in case the first record operation for the stream is a locate-mode PUT operation to a sequential file.

RFA

For block access, the CONNECT operation returns the logical end-of-file value in the 3-word RFA field of the RAB. The first two words of this field are the VBN in which the logical end-of-file occurs, and the third word is the offset of the first byte beyond the logical end-of-file within that block. This logical end-of-file value is meaningful only for disk files.

Completion Status and Value

The CONNECT operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-3 lists control block fields that are input to the CONNECT operation. Table 5-4 lists control block fields that are output by the CONNECT operation.

Table 5-3: CONNECT Input Fields

Block	Field	Description
FAB	IFI	Internal FAB identifier
RAB	FAB	FAB address
RAB	KRF	Key of reference
RAB	MBC	Multiblock count
RAB	MBF	Multibuffer count
RAB	ROP	Record processing option mask
		RB$EOF Position to end-of-file
		RB$LOC Locate mode
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-4: CONNECT Output Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RBF	Record buffer address
RAB	RFA	End-of-file address
RAB	STS	Completion status code
RAB	STV	Completion status value

5.3 $CREATE MACRO

The $CREATE macro calls the CREATE operation routine to create a new file and open it for processing.

FORMAT

The format for the $CREATE macro is:

```
$CREATE fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the CREATE operation.

If you supply a NAM block, the CREATE operation reads its fields to obtain the expanded string buffer, and writes identifiers in its fields.

To supply a NAM block for the CREATE operation, specify the address of the NAM block in the l-word NAM field of the FAB.

Each ALL block that you supply defines one area in the created file, and you can place the area at a specific location. If you supply no ALL blocks, the file has one area; you define this area in the FAB, but you cannot place the area at a specific location. You cannot supply more than one ALL block for a sequential or relative file.

Each KEY block that you supply defines one index for the created file. You must supply at least one KEY block for an indexed file; you cannot supply KEY blocks for a relative or sequential file.

If you supply a PRO block, the CREATE operation reads its fields to obtain the protection for the file.

To supply XABs (ALL, DAT, KEY, PRO, and SUM blocks) for the CREATE operation, specify the address of the first XAB in the l-word XAB field of the FAB; specify the address of the next XAB (if any) in the l-word NXT field of each XAB; specify 0 in the NXT field of the last XAB.

All KEY blocks must be together in the chain of XABs, and must be in ascending order (by the index reference number in the l-byte REF field of the KEY block); the index reference numbers must be consecutive beginning with 0.

All ALL blocks must be together in the chain of XABs, and must be in ascending order (by the area identifier in the l-byte AID field of the ALL block); the area identifiers must be consecutive beginning with 0.

Multiple DAT, PRO, or SUM XABs are illegal.

Note that if the LAN field of a KEY XAB is 0, RMS-11 will use the area specified in the IAN field for the lowest level index for that index.

OPTIONS

File Specification

The CREATE operation constructs the merged string for the target file from the file string, the default string, RMS-11 defaults, and system defaults.

Specify the address of the file string in the 1-word FNA field of the FAB. Specify the size (in bytes) of the file string in the 1-byte FNS field of the FAB; if you specify 0 in the FNS field, the CREATE operation uses no file string.

Specify the address of the default string in the 1-word DNA field of the FAB. Specify the size (in bytes) of the default string in the 1-byte DNS field of the FAB; if you specify 0 in the DNS field, the CREATE operation uses no default string.

If you set the FB$FID mask in the 1-word FOP field of the FAB and supply a NAM block, the CREATE operation reads the device identifier from the 2-word DVI field of the NAM block; if this value is nonzero, the specified device overrides the device in the merged string.

In the same circumstance, the CREATE operation reads the directory identifier from the 3-word DID field of the NAM block; if this value is nonzero, the specified directory overrides the directory in the merged string.

Expanded String Buffer

If you want the CREATE operation to return the expanded string for the created file, provide a buffer for the string. Specify the address of the expanded string buffer in the 1-word ESA field of the NAM block and its size (in bytes) in the 1-byte ESS field of the NAM block; if you specify 0 in the ESS field, the CREATE operation does not return the expanded string.

Supersession of Existing File

If you want to create a file that supersedes an existing file with the same specification, set the FB$SUP mask in the 1-word FOP field of the FAB; if you do not set the FB$SUP mask, and you specify a file that already exists, the CREATE operation returns an error completion and does not create the new file.

Temporary or Marked-for-Delete File

If you want the created file to be a temporary file (one that has no directory entry), set the FB$TMP mask in the 1-word FOP field of the FAB; if you do not set the FB$TMP mask, the created file has a directory entry.

If you want the created file to be deleted when it is closed, set the FB$MKD mask in the 1-word FOP field of the FAB; this causes the operating system to delete the file when it has no accessing programs. If you do not set the FB$MKD mask, the created file is not marked for deletion.

If you want the created file to be a temporary file that is marked for deletion, set the FB$TMD mask in the 1-word FOP field of the FAB; the FB$TMD mask includes the bits for both the FB$TMP and the FB$MKD masks.

File Protection

Specify the protection for the created file in the 1-word PRO field of the PRO block; if you supply no PRO block, the operating system uses its default file protection.

File Organization

Specify a file organization code in the 1-byte ORG field of the FAB. The symbols for file organization codes are:

FB$IDX	Indexed file organization
FB$REL	Relative file organization
FB$SEQ	Sequential file organization

Record Format

Specify the record format code in the 1-byte RFM field of the FAB. The symbols for record format codes are:

FB$FIX	Fixed-length record format
FB$STM	Stream record format
FB$UDF	Undefined record format
FB$VAR	Variable-length record format
FB$VFC	VFC record format

If you specify VFC record format (FB$VFC code in the RFM field), specify the size (in bytes) of the VFC header field in the 1-byte FSZ field of the FAB; if you specify 0, the CREATE operation uses the value 2.

Blocked Records (Sequential Disk File)

If you are creating a sequential disk file, and if you want the file to contain blocked records (records that cannot span block boundaries), set the FB$BLK mask in the 1-byte RAT field of the FAB; if you do not set the FB$BLK mask, records can span block boundaries.

Record-Output Handling

Specify a record-output mask in the 1-byte RAT field of the FAB. This record-output attribute controls the handling of records that are output to a print device (printer or terminal):

- FORTRAN-style record-output specifies FORTRAN-style carriage-control handling.
- Carriage-return record-output specifies that a prefixed linefeed and a suffixed carriage-return must be added to each record on output to a print device.

- The print record-output specifies that the file is in print format. This format is allowed only for files with VFC records for which the fixed header size for each record is 0 or 2 bytes. (RMS-11 treats a header size of 0 as if you had specified 2.)

  When records from the file are written directly to a unit-record device, RMS-11 interprets the first byte of the VFC header as a prefix for the record and the second byte of the header as a suffix for the record. RMS-11 further interprets the prefix/suffix control bytes as follows:

  - If the top bit of the control byte is clear, the entire byte is used as a count of the number of carriage return/line feed pairs with which to prefix or suffix the record.

  - If the top bit of the control byte is set, the low 5 bits of the byte are used as the prefix or suffix character.

If you specify none of these attributes, records are output without special handling.

The symbols for record-output masks are:

```
FB$CR   Add CRLF to print record (LF-record-CR)
FB$FTN  FORTRAN-style carriage-control character in record
FB$PRN  VFC print record handling
```

Record Size

Specify the record size (in bytes) in the 1-word MRS field of the FAB (unless you have specified undefined record format). For fixed-length records, the CREATE operation uses this value as the record size; for variable-length records, the CREATE operation uses this value as the maximum record size; for VFC records, the CREATE operation uses this value as the maximum size of the variable portion of each record.

If you specify a nonzero value in the MRS field, RMS-11 checks the size of each record written to the file against the MRS-field value, and returns an error completion if the record size is inappropriate; if you specify 0 in the MRS field, RMS-11 does not check record sizes against the MRS-field value.

Maximum Record Number

If you specify relative file organization (FB$REL value in the ORG field), specify the maximum record number in the 2-word MRN field of the FAB. If you specify a nonzero value in the MRN field, RMS-11 checks the record number of each record written to the file against the MRN-field value, and returns an error completion if the record number is too large; if you specify 0 in the MRN field, RMS-11 does not check record numbers against the MRN-field value.

Private Buffer Pool

If you want the CREATE operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the CREATE operation uses the central buffer pool.

The pool that the CREATE operation uses is also used by the DISPLAY and EXTEND operations, and by stream and record operations while the file is open.

Logical Channel

Specify the logical channel for the CREATE operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

The logical channel that the CREATE operation uses is also used by the DISPLAY and EXTEND operations, and by stream and record or block operations while the file is open.

Retrieval Pointers

Specify the number of retrieval pointers for the open file in the 1-byte RTV field of the FAB. If you specify 0, the CREATE operation uses the operating system default; if you specify -1, the CREATE operation maps as much of the file as possible.

Requested Access

Specify one or more requested-access masks in the 1-byte FAC field of the FAB. This mask determines the access that the creating program has while the file is open. Regardless of what you specify, the CREATE operation includes the mask FB$PUT (for record access) or FB$WRT (for block access). The symbols for requested-access masks are:

FB$DEL	Request find/get/delete access
FB$GET	Request find/get access
FB$PUT	Request put access
FB$REA	Request block read access
FB$TRN	Request find/get/truncate access
FB$UPD	Request find/get/update access
FB$WRT	Request block write access

Note that FB$REA and FB$WRT override any record access requested.

Access Sharing

Specify the kinds of access that your program is willing to share with other programs by setting an access-sharing mask in the 1-byte SHR field of the FAB. The symbols for access-sharing masks are:

FB$GET	Share find/get access
FB$NIL	No access sharing
FB$WRI	Share find/get/put/update/delete access
FB$UPI	Share any access (user-provided interlock)

The kinds of access sharing are:

- **Shared read access**

  Your program is willing to allow other programs to read the file, but not to write it.

- **Shared write access**

  Your program is willing to allow other programs to both read and write the file. Shared write access is not allowed for a sequential file unless the file has undefined record format and your program opens the file for block access; shared write access is also not allowed for a relative or indexed file that your program opens for block access. In such cases, RMS-11 automatically converts the shared write access specification to a shared read access specification internally.

- **No shared access**

  Your program is not willing to allow other programs to either read or write the file. RMS-11 does, however, allow other programs to read the file unless your program also requests some form of write access (which is always the case for CREATE).

Deferred Writing

If you want deferred buffer writing for the open file, set the FB$DFW mask in the 1-word FOP field of the FAB; This means that RMS-11 does not necessarily write its buffers during a write-type operation (DELETE, PUT, or UPDATE), but instead writes buffers only when it needs them for other operations (or when your program executes the FLUSH operation for the stream).

If you do not set the FB$DFW mask, the DELETE, PUT, and UPDATE operations write buffers to the file immediately.

Note that record operations always use a form of deferred buffer writing for sequential files, and that block operations never use deferred buffer writing. Therefore you need only decide whether to use deferred writing for a record stream to a relative or indexed file.

File Locking

If you want the file to remain unlocked even if it is closed abnormally, set the FB$DLK mask in the 1-word FOP field of the FAB; if you do not set the FB$DLK mask, the operating system locks the file if it is closed abnormally.

Single-Area Unlocated File

If you want the created file to have only one area, and if you do not want to place the area at a specific location on disk, then you supply no ALL blocks for the CREATE operation, but rather specify the following file attributes in FAB fields (as described in sections below):

- File allocation size
- Default file extension size
- File bucket size
- File contiguity

Multiarea or Located File

If you want to place the created file at a specific location on disk, or if you want a created indexed file to have more than one area, then you supply ALL blocks for the CREATE operation and you specify the following area attributes in ALL block fields (as described in sections below):

- Area allocation size
- Default area extension size
- Area bucket size
- Area contiguity
- Area alignment
- Area location

Specify the area number for each area in the 1-byte AID field of the ALL block for the area.

Allocation Size

For a single-area unlocated file, specify the file allocation size (in blocks) in the 2-word ALQ field of the FAB. For a multiarea or located file, specify the area allocation size (in blocks) in the 2-word ALQ field of the ALL block for each area.

Default Extension Size

For a single-area unlocated file, specify the default extension size (in blocks) for the file in the 1-word DEQ field of the FAB. For a multiarea or located file, specify the default extension size (in blocks) for each area in the 1-word DEQ field of the ALL block for the area.

Bucket Size (Relative or Indexed File)

For a single-area unlocated file, specify the bucket size (in blocks) for the file in the 1-byte BKS field of the FAB. For a multiarea or located file, specify the bucket size (in blocks) for each area in the 1-byte BKZ field of the ALL block for the area.

The largest allowed bucket size is 32 blocks; the smallest is 0. If you specify a bucket size of 0, the CREATE operation uses 1-block buckets for the file or area.

Area Location

If you want to place an area at a particular location on disk, specify an alignment mask in the 1-byte ALN field of the ALL block for the area. Cylinder alignment (available only for the VAX-11 AME) places the area at a specified cylinder; logical block alignment places the area at a specified logical block; virtual block alignment (not allowed for area 0) places the area near a specified virtual block. If you specify no alignment mask, the CREATE operation places the area at any convenient location. The symbols for alignment masks are:

```
XB$CYL  Cylinder alignment
XB$LBN  Logical block alignment
XB$VBN  Virtual block alignment
```

Specify the number of the cylinder, logical block, or virtual block in the 2-word LOC field of the ALL block for the area.

If you do not want the file to be created unless the specified area location is available, set the XB$HRD mask in the 1-byte AOP field of the ALL block for the area. If you do not set this mask, the CREATE operation creates the file even if it must place the area at an alternate location. Note that hard location at a virtual block location is illegal.

The CREATE operation creates areas by extending the file if either of the following is true:

- You specify placement for areas other than area 0 (in which case the CREATE operation ignores the FB$CTG mask).
- You specify contiguity in one or more ALL blocks, but not in the FAB for the file.

Otherwise the CREATE operation creates the entire file as a single operation, and, if you specified contiguity in the FAB, creates the entire file as a single contiguous extent.

Note also that because virtual block alignment is always possible, specifying hard location for virtual block alignment has no effect.

Contiguity

If you want a file to be contiguous, set the FB$CTG mask in the 1-word FOP field of the FAB and (for a multiarea file) do not specify disk location for any area except (optionally) area 0; if the CREATE operation cannot create a contiguous file, it returns an error completion; if you do not set this mask, the CREATE operation does not attempt to create a contiguous file.

If you want an area of a multiarea or located file to be contiguous, set the XB$CTG mask in the 1-byte AOP field of the ALL block for the area. If you set this mask and the CREATE operation cannot create a contiguous area, it returns an error completion; if you do not set this mask, the CREATE operation does not attempt to create a contiguous area.

Indexes

If you specify indexed file organization (FB$IDX value in the ORG field), you must supply at least one KEY block for the CREATE operation. Each KEY block you supply defines one index for the created file.

Specify the reference number for each index in the 1-byte REF field of the KEY block for the index. Specify 0 for the primary index, 1 for the first alternate index, and so forth. Chain KEY blocks so that the reference numbers are in consecutive order, and so that there are no intervening XABs of other types (ALL, DAT, PRO, or SUM blocks).

Key Name

If you want to define a key name for the index, place the key name string in a 32-character buffer. Specify the address of this buffer in the 1-word KNM field of the KEY block for the index. If you specify 0 in the KNM field, the index has no key name.

Index Key Data Type

Specify a key data type code in the 1-byte DTP field of the KEY block for each index. The symbols for key data type codes are:

XB$BN2	16-bit unsigned integer
XB$BN4	32-bit unsigned integer
XB$IN2	15-bit signed integer
XB$IN4	31-bit signed integer
XB$PAC	Packed decimal number
XB$STG	String

Key Segments

Specify the size and position of each key segment in the 8-byte SIZ field of the KEY block and the 8-word POS field of the KEY block for the index. (Only a string key can have more than one segment.)

The first byte of the SIZ field is for the size (in bytes) of the first key segment, the second byte is for the second segment, and so forth. If the key is to have fewer than eight segments, specify 0 in the remaining bytes of the SIZ field. (The CREATE operation does not check segment sizes after the first 0 it encounters in the SIZ field.)

The first word of the POS field is for the position of the first key segment, the second word is for the second segment, and so forth. If the key has fewer than eight segments, the CREATE operation ignores the remaining words of the POS field. (The first position in a record is position 0.)

Key Changes

For an alternate index, if you want to allow the key to change during update operations, set the XB$CHG mask in the 1-byte FLG field of the KEY block and the XB$DUP mask in the 1-byte FLG field of the KEY block for the index; if you do not set these masks, RMS-11 returns an error if a program attempts to change the value of a record key during updating.

Key Duplications

If you want to allow duplicate keys in an index, set the XB$DUP mask in the 1-byte FLG field of the KEY block for the index. If you do not set this mask, RMS-11 returns an error if a program attempts to insert or update a record that would create a duplicate record key. Note that the XB$DUP mask must be set if record keys in the index are to be changeable during update.

Null Keys

If you want to omit null keys from an alternate index, set the XB$NUL mask in the 1-byte FLG field of the KEY block for the index, and (for a string key) specify the null character for the key in the 1-byte NUL field of the KEY block (the null value for a nonstring key is 0).

If you do not set the XB$NUL mask, all keys are included in the index; if you set the XB$NUL mask, a nonstring key with a 0 value or a string key with an all-null value will not appear in that alternate index.

Index Areas

Specify areas for the data records and for the levels of the index:

- The area for data records in the 1-byte DAN field of the KEY block.
- The area for the lowest index level in the 1-byte LAN field of the KEY block.
- The area for higher index levels in the 1-byte IAN field of the KEY block.

Note that the bucket sizes of the LAN and IAN areas of a given index must be identical.

Bucket Fill Numbers

Bucket fill numbers guide the PUT and UPDATE operations in deciding how many records to place in each bucket. A bucket fill number of 0 is usually appropriate, and specifies that buckets should be filled completely.

A nonzero bucket fill number specifies the number of bytes that should be filled in each bucket. If the specified bucket fill number is less than half the bucket size, it is rounded up to half the bucket size; if the specified number is more than the bucket size, it is rounded down to the bucket size.

Specify the fill numbers for data buckets and index buckets: the fill number for data buckets in the 1-word DFL field of the KEY block, and the fill number for index buckets in the 1-word IFL field of the KEY block.

Longest Record Length

If you specify block access for the created file, and you plan to copy an existing file into the new file, you can specify the length of the longest record in the new file in the 1-word LRL field of the FAB.

RETURNED VALUES

Internal File Identifier

The CREATE operation writes an internal file identifier in the 1-word IFI field of the FAB. (The CLOSE operation clears the internal file identifier.)

The CLOSE, CONNECT, DISPLAY, and EXTEND operations read the internal file identifier; do not alter the IFI field while the file is open.

Device Characteristics

The CREATE operation returns device characteristics as masks in the 1-byte DEV field of the FAB. The device characteristics are:

- Printer or terminal (indicated by the set FB$CCL mask in the 1-byte DEV field of the FAB and the set FB$REC mask in the 1-byte DEV field of the FAB; for a terminal, the FB$TRM mask in the 1-byte DEV field of the FAB is also set); RMS-11 treats a printer or terminal as a unit-record device.

- Disk, DECtape, or DECTAPE II (indicated by the set FB$MDI mask in the 1-byte DEV field of the FAB); RMS-11 treats a disk, DECtape, or DECTAPE II as a disk device.

- Unit-record device (indicated by the set FB$REC mask in the 1-byte DEV field of the FAB).

- Non-ANSI magtape or cassette tape (indicated by the set FB$SDI mask in the 1-byte DEV field of the FAB and the set FB$REC mask in the 1-byte DEV field of the FAB); RMS-11 treats a non-ANSI magtape or a cassette tape as a unit-record device.

- ANSI-format magtape (indicated by the set FB$SQD mask in the 1-byte DEV field of the FAB).

Device, Directory, and File Identifiers

If you supply a NAM block, the CREATE operation writes a device identifier in the 2-word DVI field of the NAM block, a directory identifier in the 3-word DID field of the NAM block, and a file identifier in the 3-word FID field of the NAM block.

You must save these identifiers if you want to open (after closing) or erase the file using its file identifiers.

Expanded String

If you specify a buffer for the expanded string for the file (ESA and ESS fields in the NAM block), the CREATE operation writes the file specification for the created file in this buffer, and writes the length (in bytes) of the specification string in the 1-byte ESL field of the NAM block.

File Specification Characteristics

The CREATE operation sets masks in the 1-word FNB field of the NAM block to show which file specification elements were present in the file string and default string. These masks and their meanings are:

NB$NOD	Node in file string or default string
NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$QUO	Quoted string in file string or default string
NB$NAM	File name in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Wildcarding

The CREATE operation clears the NB$WCH mask in the 1-word FNB field of the NAM block; this shows that no wildcard context exists after the CREATE operation. It also clears the 1-byte RSL field of the NAM block to show that no resultant string was returned.

Extension Sizes

The CREATE operation returns the size (in blocks) of each allocation it makes. If you created only area 0 using FAB fields, the CREATE operation writes the size of the allocation in the 2-word ALQ field of the FAB. If you created areas using ALL blocks, the CREATE operation writes the size of each area allocation in the 2-word ALQ field of the ALL block for the area.

Completion Status and Value

The CREATE operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-5 lists control block fields that are input to the CREATE operation. Table 5-6 lists control block fields that are output by the CREATE operation.

Table 5-5: CREATE Input Fields

Block	Field	Description
ALL	AID	Area number
ALL	ALN	Initial area alignment request
		XB$CYL Cylinder alignment
		XB$LBN Logical block alignment
		XB$VBN Virtual block alignment
ALL	ALQ	Initial area allocation request size (blocks)
ALL	AOP	Area option mask
		XB$CTG Contiguous area request
		XB$HRD Area hard location request
ALL	BKZ	Area bucket size (blocks)
ALL	DEQ	Area default extension size (blocks)
ALL	LOC	Initial area location request
ALL	NXT	Next XAB address
DAT	NXT	Next XAB address
FAB	ALQ	Initial file allocation request size (blocks)
FAB	BKS	File bucket size (blocks)
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DEQ	Permanent file default extension size (blocks)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)
FAB	FAC	Requested access mask
		FB$DEL Request find/get/delete access
		FB$GET Request find/get access
		FB$PUT Request put access
		FB$REA Request block read access
		FB$TRN Request find/get/truncate access
		FB$UPD Request find/get/update access
		FB$WRT Request block write access
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	FOP	File processing option mask
		FB$CTG Contiguous file request
		FB$DFW Defer writing
		FB$DLK No file locking on abnormal close
		FB$FID Use information in NAM block
		FB$MKD Mark file for deletion
		FB$SUP Supersede existing file
		FB$TMD Temporary file, mark for deletion
		FB$TMP Temporary file
FAB	FSZ	Fixed control area size for VFC records (bytes)
FAB	LCH	Logical channel number
FAB	LRL	Longest record length
FAB	MRN	Maximum record number
FAB	MRS	Maximum record size (bytes)
FAB	NAM	NAM block address

(continued on next page)

Table 5-5 (cont.): CREATE Input Fields

Block	Field	Description
FAB	ORG	File organization code
		FB$IDX Indexed file organization
		FB$REL Relative file organization
		FB$SEQ Sequential file organization
FAB	RAT	Record handling mask
		FB$BLK Blocked records
		FB$CR Add CRLF to print record (LF-record-CR)
		FB$FTN FORTRAN-style carriage-control character in record
		FB$PRN VFC print record handling
FAB	RFM	Record format code
		FB$FIX Fixed-length record format
		FB$STM Stream record format
		FB$UDF Undefined record format
		FB$VAR Variable-length record format
		FB$VFC VFC record format
FAB	RTV	Retrieval pointer count
FAB	SHR	Shared access mask
		FB$GET Share find/get access
		FB$NIL No access sharing
		FB$WRI Share find/get/put/update/delete access
FAB	XAB	XAB address
KEY	DAN	Data area number
KEY	DFL	Data bucket fill factor
KEY	DTP	Key data type code
		XB$BN2 16-bit unsigned integer
		XB$BN4 32-bit unsigned integer
		XB$IN2 15-bit signed integer
		XB$IN4 31-bit signed integer
		XB$PAC Packed decimal number
		XB$STG String
KEY	FLG	Index option mask
		XB$DUP Duplicate record keys allowed
		XB$CHG Record key changes allowed on update
		XB$NUL Null record keys not indexed
KEY	IAN	Higher level index area number
KEY	IFL	Index bucket fill factor
KEY	KNM	Key name buffer address
KEY	LAN	Lowest index level area number
KEY	NUL	Null key character
KEY	NXT	Next XAB address
KEY	POS	Key segment positions
KEY	REF	Index reference number
KEY	SIZ	Key segment sizes (bytes)

(continued on next page)

Table 5-5 (cont.): CREATE Input Fields

Block	Field	Description
NAM	ESA	Expanded string buffer address
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESS	Expanded string buffer size (bytes)
PRO	NXT	Next XAB address
PRO	PRO	File protection code
SUM	NXT	Next XAB address

Table 5-6: CREATE Output Fields

Block	Field	Description
ALL	ALQ	Initial area allocation size (blocks)
FAB	ALQ	Initial file allocation size (blocks)
FAB	DEV	Device characteristic mask
		FB$CCL Carriage-control device
		FB$MDI Multidirectory device
		FB$REC Record-oriented device
		FB$SDI Single-directory device
		FB$SQD Sequential device
		FB$TRM Terminal device
FAB	IFI	Internal FAB identifier
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESL	Expanded string length (bytes)
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
		NB$WCH Wildcard context established (cleared)
NAM	RSL	Resultant string length (bytes) (cleared)

5.4 $DELETE MACRO

The $DELETE macro calls the DELETE operation routine to remove a record from a relative or indexed file. The target of the DELETE operation is the current record. The current record must be locked; it was automatically locked when the current-record context was set, but you must not have unlocked it with a FREE operation.

If the stream has no current-record context, or if the current record is not locked, the DELETE operation returns an error completion.

FORMAT

The format for the $DELETE macro is:

```
$DELETE rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the DELETE operation.

OPTIONS

Internal Stream Identifier

The DELETE operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Fast Deletion (Indexed File)

If the file is an indexed file, and if its alternate indexes allow duplicate keys, then you can speed up the DELETE operation by using the fast-deletion procedure. However, this procedure is faster because it deletes only those alternate index pointers that it must; future retrieval operations may be slowed by the presence of undeleted alternate index pointers.

To use the fast-deletion procedure with the DELETE operation, set the RB$FDL mask in the 1-word ROP field of the RAB. If you do not set this mask, the DELETE operation does not use the fast-deletion procedure.

STREAM CONTEXT

The current-record context after a DELETE operation is undefined; the next-record context is unchanged.

RETURNED VALUES

Completion Status and Value

The DELETE operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-7 lists control block fields that are input to the DELETE operation. Table 5-8 lists control block fields that are output by the DELETE operation.

Table 5-7: DELETE Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	ROP	Record processing option mask
		RB$FDL Fast deletion

Table 5-8: DELETE Output Fields

Block	Field	Description
RAB	STS	Completion status code
RAB	STV	Completion status value

5.5 $DISCONNECT MACRO

The $DISCONNECT macro calls the DISCONNECT operation routine to terminate a stream and disconnect it, releasing the internal resources it was using. The stream context is lost; you cannot reestablish the same stream context by reconnecting the stream with the CONNECT operation.

FORMAT

The format for the $DISCONNECT macro is:

```
$DISCONNECT rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the DISCONNECT operation.

OPTIONS

Internal Stream Identifier

The DISCONNECT operation reads the internal stream identifier from the 1-word ISI field of the RAB.

STREAM CONTEXT

The DISCONNECT operation terminates the stream; therefore there is no stream context after the DISCONNECT operation.

RETURNED VALUES

Internal Stream Identifier (Cleared)

The DISCONNECT operation clears the internal stream identifier from the 1-word ISI field of the RAB.

Completion Status and Value

The DISCONNECT operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-9 lists control block fields that are input to the DISCONNECT operation. Table 5-10 lists control block fields that are output by the DISCONNECT operation.

Table 5-9: DISCONNECT Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier

Table 5-10: DISCONNECT Output Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	STS	Completion status code
RAB	STV	Completion status value

5.6 $DISPLAY MACRO

The $DISPLAY macro calls the DISPLAY operation routine to write values into control block fields. The DISPLAY operation does not alter the file in any way.

When you use the OPEN operation to open a file, you might not know how many areas or how many indexes the file has. If, however, you supply a SUM block for the OPEN operation, the OPEN operation writes the number of areas and number of keys (indexes) in its fields. You can then supply ALL blocks and KEY blocks so that the DISPLAY operation can fill their fields with values describing the file areas and indexes.

FORMAT

The format for the $DISPLAY macro is:

```
$DISPLAY fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the DISPLAY operation.

If the file is an indexed file, for each ALL block that you supply, the DISPLAY operation fills its fields with values describing the corresponding area (if any) of the file. You need not supply an ALL block for every area of the file. Note that if the file was opened for block access, no information is returned in ALL blocks.

For each KEY block that you supply, the DISPLAY operation fills its fields with values describing the corresponding index (if any) for the file. You need not supply a KEY block for every index of the file. Note that if the file was opened for block access, no information is returned in KEY blocks.

If you supply a PRO block, the DISPLAY operation fills its fields with values showing the owner and protection for the file.

If you supply a DAT block, the DISPLAY operation fills its fields with values showing the creation date, expiration date, revision date, and revision number for the file.

If you supply a SUM block for a relative or indexed file, the DISPLAY operation fills its fields with values showing the number of areas and indexes for the file, and with its prologue version number. (If you are opening the file for block access, the DISPLAY operation returns the number of areas and number of keys as 0, and does not return the prologue version number.)

To supply XABs (ALL, DAT, KEY, PRO, and SUM blocks) for the DISPLAY operation, specify the address of the first XAB in the 1-word XAB field of the FAB; specify the address of the next XAB (if any) in the 1-word NXT field of each XAB; specify 0 in the NXT field of the last XAB.

All KEY blocks must be together in the chain of XABs, and must be in ascending order (by the index reference number in the 1-byte REF field of the KEY block); the index reference numbers need not be consecutive.

All ALL blocks must be together in the chain of XABs, and must be in ascending order (by the area identifier in the 1-byte AID field of the ALL block); the area identifiers need not be consecutive.

Multiple DAT, PRO, or SUM XABs are illegal.

OPTIONS

Internal File Identifier

The DISPLAY operation reads the internal file identifier from the 1-word IFI field of the FAB. This is the value that was written when the file was opened by the CREATE or OPEN operation.

Key Name Buffer

If you want the key name string for an index returned to a buffer, supply a KEY block for the index; specify the address of a 32-byte buffer in the 1-word KNM field of the KEY block. If you do not supply a KEY block for an index, or if you specify 0 in its KNM field, the DISPLAY operation does not return the key name string.

STREAM CONTEXT

The DISPLAY operation does not affect stream context.

RETURNED VALUES

Area Descriptions

For each ALL block that you supply, the DISPLAY operation writes a description in its fields of the corresponding area of the file. Area 0 is described in the ALL block containing 0 in its AID field; area 1 is described in the ALL block containing 1 in its AID field; and so forth.

The DISPLAY operation writes three sizes for a file area: the size (in blocks) of the unused portion of the area in the 2-word ALQ field of the ALL block, the default area extension size (in blocks) in the 1-word DEQ field of the ALL block, and the area bucket size (in blocks) in the 1-byte BKZ field of the ALL block.

The DISPLAY operation clears the 1-byte AOP field of the ALL block and the 1-byte ALN field of the ALL block.

Key Descriptions

For each KEY block that you supply, the DISPLAY operation writes a description in its fields of the corresponding index of the file. The primary index is described in the KEY block containing 0 in its REF field; the first alternate index is described in the KEY block containing 1 in its REF field; and so forth.

The DISPLAY operation writes the key data type code in the 1-byte DTP field of the KEY block. The symbols for key data type codes are:

```
XB$BN2  16-bit unsigned integer
XB$BN4  32-bit unsigned integer
XB$IN2  15-bit signed integer
```

XB$IN4	31-bit signed integer
XB$PAC	Packed decimal number
XB$STG	String

The DISPLAY operation writes key segment information for the index: the number of key segments in the 1-byte NSG field of the KEY block, and the total key size (sum of segments, in bytes) in the 1-byte TKS field of the KEY block.

The DISPLAY operation writes the sizes of key segments in the 8-byte SIZ field of the KEY block. The size (in bytes) of the first key segment is in the first byte of the SIZ field, the size of the second segment is in the second byte of the SIZ field, and so forth. If the key has fewer than eight segments, the first byte containing 0 indicates the number of key segments.

The DISPLAY operation writes the positions of key segments in the 8-word POS field of the KEY block. The position (leftmost position is 0) of the first key segment is in the first word of the POS field, the position of the second segment is in the second word of the POS field, and so forth. If the key has fewer than eight segments, the remaining words of the POS field contain unpredictable values.

The DISPLAY operation writes a key-characteristics mask in the 1-byte FLG field of the KEY block. The symbols for key-characteristics masks are:

XB$CHG	Record key changes allowed on update
XB$DUP	Duplicate record keys allowed
XB$INI	No entries yet made in index
XB$NUL	Null record keys not indexed

The DISPLAY operation writes the null-key character in the 1-byte NUL field of the KEY block. This character is meaningful only if the XB$NUL mask in the 1-byte FLG field of the KEY block is set and the DISPLAY operation returns the XB$STG code in the 1-byte DTP field of the KEY block (indicating a string key).

The DISPLAY operation writes area numbers for the index: the area for the data level in the 1-byte DAN field of the KEY block, the area for the lowest index level in the 1-byte LAN field of the KEY block, and the area for higher index levels in the 1-byte IAN field of the KEY block.

The DISPLAY operation writes bucket fill numbers for the index areas: the fill number for the data area in the 1-word DFL field of the KEY block, and the fill number for the index areas in the 1-word IFL field of the KEY block.

The DISPLAY operation writes bucket sizes for index areas: the data area bucket size (in blocks) in the 1-byte DBS field of the KEY block, and the index area bucket size (in blocks) in the 1-byte IBS field of the KEY block.

The DISPLAY operation writes virtual block numbers for the index areas: the virtual block number for the first data bucket in the 2-word DVB field of the KEY block, and the virtual block number of the root index bucket in the 2-word RVB field of the KEY block.

The DISPLAY operation writes the number of levels in the index (not including the data level) in the 1-byte LVL field of the KEY block.

The DISPLAY operation writes the minimum size (in bytes) of a record that contains the key for the index in the 1-word MRL field of the KEY block.

File Owner and Protection (Disk File)

If the file is a disk file, and if you supply a PRO block, the DISPLAY operation writes the project (or group) portion of the file owner code in the 1-word PRJ field of the PRO block, the programmer (or member) portion of the file owner code in the 1-word PRG field of the PRO block, and the file protection code in the 1-word PRO field of the PRO block.

File Dates

If you supply a DAT block for a disk file, the DISPLAY operation writes four values in its fields: the creation date in the 4-word CDT field of the DAT block, the expiration date in the 4-word EDT field of the DAT block, the revision date in the 4-word RDT field of the DAT block, and the revision number (number of times the file has been write-accessed and then closed) in the 1-word RVN field of the DAT block.

File Summary Information

If you supply a SUM block, the DISPLAY operation writes three values in its fields: the number of file areas in the 1-byte NOA field of the SUM block, the number of file indexes in the 1-byte NOK field of the SUM block, and the prologue version number (for a relative or indexed file) in the 1-word PVN field of the SUM block.

Completion Status and Value

The DISPLAY operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-11 lists control block fields that are input to the DISPLAY operation. Table 5-12 lists control block fields that are output by the DISPLAY operation.

Table 5-11: DISPLAY Input Fields

Block	Field	Description
ALL	AID	Area number
ALL	NXT	Next XAB address
DAT	NXT	Next XAB address
FAB	IFI	Internal FAB identifier
FAB	XAB	XAB address
KEY	NXT	Next XAB address
KEY	KNM	Key name buffer address
KEY	REF	Index reference number
PRO	NXT	Next XAB address
SUM	NXT	Next XAB address

Table 5-12: DISPLAY Output Fields

Block	Field	Description
ALL	ALN	Area alignment mask (cleared)
ALL	ALQ	Unused area allocation size (blocks)
ALL	AOP	Area option mask
		XB$CTG Contiguous area (cleared)
		XB$HRD Hard area location (cleared)
ALL	BKZ	Area bucket size (blocks)
ALL	DEQ	Area default extension size (blocks)
DAT	CDT	File creation date
DAT	EDT	File expiration date
DAT	RDT	File revision date
DAT	RVN	File revision number
FAB	STS	Completion status code
FAB	STV	Completion status value
KEY	DAN	Data area number
KEY	DBS	Data area bucket size (blocks)
KEY	DFL	Data bucket fill factor
KEY	DTP	Key data type code
		XB$BN2 16-bit unsigned integer
		XB$BN4 32-bit unsigned integer
		XB$IN2 15-bit signed integer
		XB$IN4 31-bit signed integer
		XB$PAC Packed decimal number
		XB$STG String
KEY	DVB	First data bucket virtual block number
KEY	FLG	Index option mask
		XB$CHG Record key changes allowed on update
		XB$DUP Duplicate record keys allowed
		XB$INI No entries yet made in index
		XB$NUL Null record keys not indexed
KEY	IAN	Higher level index area number
KEY	IBS	Index area bucket size (blocks)
KEY	IFL	Index bucket fill factor
KEY	LAN	Lowest index level area number
KEY	LVL	Number of index levels (not including data level)
KEY	MRL	Minimum length of record containing key (bytes)
KEY	NSG	Key segment count
KEY	NUL	Null key character
KEY	POS	Key segment positions
KEY	RVB	Root index bucket virtual block number
KEY	SIZ	Key segment sizes (bytes)
KEY	TKS	Total key size (sum of key segment sizes) (bytes)
PRO	PRG	Programmer or member portion of file owner code
PRO	PRJ	Project or group portion of file owner code
PRO	PRO	File protection code
SUM	NOA	Number of areas
SUM	NOK	Number of indexes
SUM	PVN	Prologue version number

5.7 $ENTER MACRO

The $ENTER macro calls the ENTER operation routine to insert a file name into a directory file.

FORMAT

The format for the $ENTER macro is:

```
$ENTER fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the ENTER operation.

You must supply a NAM block for the ENTER operation.

To supply a NAM block for the ENTER operation, specify the address of the NAM block in the 1-word NAM field of the FAB.

OPTIONS

File Specification

The ENTER operation constructs the merged string for the target file from the file string, the default string, RMS-11 defaults, and system defaults.

Specify the address of the file string in the 1-word FNA field of the FAB. Specify the size (in bytes) of the file string in the 1-byte FNS field of the FAB; if you specify 0 in the FNS field, the ENTER operation uses no file string.

Specify the address of the default string in the 1-word DNA field of the FAB. Specify the size (in bytes) of the default string in the 1-byte DNS field of the FAB; if you specify 0 in the DNS field, the ENTER operation uses no default string.

Specify the file identifier of the target file in the 3-word FID field of the NAM block.

If you set the FB$FID mask in the 1-word FOP field of the FAB and supply a NAM block, the ENTER operation reads the device identifier from the 2-word DVI field of the NAM block; if this value is nonzero, the specified device overrides the device in the merged string.

In the same circumstance, the ENTER operation reads the directory identifier from the 3-word DID field of the NAM block; if this value is nonzero, the specified directory overrides the directory in the merged string.

Expanded String Buffer

If you want the ENTER operation to return the expanded string for the created file, provide a buffer for the string. Specify the address of the expanded string buffer in the 1-word ESA field of the NAM block. Specify the size (in bytes) of the expanded string buffer in the

1-byte ESS field of the NAM block; if you specify 0 in the ESS field, the ENTER operation does not return the expanded string.

Private Buffer Pool

If you want the ENTER operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the ENTER operation uses the central buffer pool.

Logical Channel

Specify the logical channel for the ENTER operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

RETURNED VALUES

Expanded String

If you specified an expanded string buffer, the ENTER operation returns the expanded string in the buffer, and writes the length (in bytes) of the string in the 1-byte ESL field of the NAM block.

Device and Directory Identifiers

The ENTER operation returns the device identifier for the target file in the 2-word DVI field of the NAM block and the directory identifier in the 3-word DID field of the NAM block.

File Specification Characteristics

The ERASE operation sets masks in the 1-word FNB field of the NAM block to show which file specification elements were present in the file string and default string. These masks and their meanings are:

NB$NOD	Node in file string or default string
NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$QUO	Quoted string in file string or default string
NB$NAM	File name in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Wildcarding

The ENTER operation clears the NB$WCH mask in the 1-word FNB field of the NAM block and the 1-byte RSL field of the NAM block; this shows that no wildcard context exists and that no resultant string was returned.

Completion Status and Value

The ENTER operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-15 lists control block fields that are input to the ENTER operation. Table 5-16 lists control block fields that are output by the ENTER operation.

Table 5-15: ENTER Input Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	FOP	File processing option mask
		FB$FID Use information in NAM block
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESA	Expanded string buffer address
NAM	ESS	Expanded string buffer size (bytes)
NAM	FID	File identifier

Table 5-16: ENTER Output Fields

Block	Field	Description
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESL	Expanded string length (bytes)
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
		NB$WCH Wildcard context established (cleared)
NAM	RSL	Resultant string length (bytes) (cleared)

5.8 $ERASE MACRO

The $ERASE macro calls the ERASE operation routine to erase a file and delete its directory entry. Note that erasing a file marks the file for deletion, but does not necessarily erase the file immediately; the file is erased when it has no accessing programs. The allocation for the file is released for use in other files.

FORMAT

The format for the $ERASE macro is:

```
$ERASE fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the ERASE operation.

If you supply a NAM block and specify wildcarding, the ERASE operation reads the address and length of the expanded string from NAM block fields; if you supply a NAM block and specify erase by NAM block, the ERASE operation reads NAM block fields to obtain identifiers for the target file.

To supply a NAM block for the ERASE operation, specify the address of the NAM block in the 1-word NAM field of the FAB.

OPTIONS

Erase by File Specification

If you want to erase a file by its file specification, clear the FB$FID mask in the 1-word FOP field of the FAB.

If the FB$FID mask is clear, the ERASE operation uses the fully qualified file specification to determine the target file.

Specify the address of the file string in the 1-word FNA field of the FAB; specify the size (in bytes) of the file string in the 1-byte FNS field of the FAB. If you specify 0 in the FNS field, the ERASE operation uses no file string.

Specify the address of the default string in the 1-word DNA field of the FAB; specify the size (in bytes) of the default string in the 1-byte DNS field of the FAB. If you specify 0 in the DNS field, the ERASE operation uses no default string.

Erase by NAM Block

If you want to identify the target file by its identifiers, set the FB$FID mask in the 1-word FOP field of the FAB, and supply a filled-in NAM block (one whose fields were written by a CREATE or OPEN operation for the file); this causes the ERASE operation to ignore the file specification fields (FNA, FNS, DNA, and DNS) and identify the file using identifiers in the NAM block: the 2-word DVI field of the NAM block and the 3-word FID field of the NAM block.

You can erase by NAM block without providing a file identifier; if the file identifier is 0, the device and directory identifiers (if nonzero) override in the normal manner (and the directory entry is removed).

Note that erasing a file by its file identifier does not remove the directory entry (if any) for the file.

Erase by Wildcard Specification

You can use the ERASE operation in a wildcarding program loop. (The NB$WCH mask in the 1-word FNB field of the NAM block will already have been set by an earlier PARSE operation.)

If you set the FB$FID mask in the 1-word FOP field of the FAB, the file found by a previous SEARCH operation and its directory entry are deleted, but all fields relevant to wildcard context are preserved (for possible subsequent SEARCH operations).

If you clear the FB$FID mask in the 1-word FOP field of the FAB, the ERASE operation first performs an implicit SEARCH operation. (The input and output fields for the SEARCH operation are not described here and are not included in the checklists at the end of this section.)

If the SEARCH operation finds a file that matches the wildcard file specification, the ERASE operation erases its contents and deletes its directory entry; if not, the ERASE operation does not erase the file contents or delete its directory entry, but instead passes control block data from the SEARCH operation (in particular, the ER$NMF completion status code and the cleared NB$WCH mask in the 1-word FNB field of the NAM block).

Expanded String Buffer

If you erase a file by its file specification, and if you want the ERASE operation to return the expanded string for the erased file, provide a buffer for the string. Specify the address of the expanded string buffer in the 1-word ESA field of the NAM block. Specify the size (in bytes) of the expanded string buffer in the 1-byte ESS field of the NAM block; if you specify 0 in the ESS field, the ERASE operation does not return the expanded string.

Private Buffer Pool

If you want the ERASE operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the ERASE operation uses the central buffer pool.

Logical Channel

Specify the logical channel for the ERASE operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

RETURNED VALUES

Expanded String

If you specify a buffer for the expanded string for the file (ESA and ESS fields in the NAM block), the ERASE operation writes the expanded string for the erased file in the buffer, and writes the length (in bytes) of the string in the 1-byte ESL field of the NAM block.

Device, Directory, and File Identifiers

The ERASE operation returns identifiers: the device identifier in the 2-word DVI field of the NAM block, the directory identifier in the 3-word DID field of the NAM block (unless you specified erase by file identifier), and the file identifier in the 3-word FID field of the NAM block.

Device Characteristics

The ERASE operation returns device characteristics as masks in the 1-byte DEV field of the FAB. The device characteristics are:

- Printer or terminal (indicated by the set FB$CCL mask in the 1-byte DEV field of the FAB and the set FB$REC mask in the 1-byte DEV field of the FAB; for a terminal, the FB$TRM mask in the 1-byte DEV field of the FAB is also set); RMS-11 treats a printer or terminal as a unit-record device.
- Disk, DECtape, or DECTAPE II (indicated by the set FB$MDI mask in the 1-byte DEV field of the FAB); RMS-11 treats a disk, DECtape, or DECTAPE II as a disk device.
- Unit-record device (indicated by the set FB$REC mask in the 1-byte DEV field of the FAB).
- Non-ANSI magtape or cassette tape (indicated by the set FB$SDI mask in the 1-byte DEV field of the FAB and the set FB$REC mask in the 1-byte DEV field of the FAB); RMS-11 treats a non-ANSI magtape or a cassette tape as a unit-record device.
- ANSI-format magtape (indicated by the set FB$SQD mask in the 1-byte DEV field of the FAB).

Wildcard Context

A nonwildcard ERASE operation clears the NB$WCH mask in the 1-word FNB field of the NAM block and the 1-byte RSL field of the NAM block; this shows that no wildcarding is in progress and that no resultant string was returned.

File Specification Characteristics

The ERASE operation sets masks in the 1-word FNB field of the NAM block to show which file specification elements were present in the file string and default string. These masks and their meanings are:

```
NB$NOD  Node in file string or default string
NB$DEV  Device in file string or default string
NB$DIR  Directory in file string or default string
NB$QUO  Quoted string in file string or default string
NB$NAM  File name in file string or default string
```

NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Completion Status and Value

The ERASE operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-17 lists control block fields that are input to the ERASE operation. Table 5-18 lists control block fields that are output by the ERASE operation.

Table 5-17: ERASE Input Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	FOP	File processing option mask
		FB$FID Use information in NAM block
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
NAM	DVI	Device identifier
NAM	ESA	Expanded string buffer address
NAM	ESS	Expanded string buffer size (bytes)
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$WCH Wildcard context established

Table 5-18: ERASE Output Fields

Block	Field	Description
FAB	DEV	Device characteristic mask
		FB$CCL Carriage-control device
		FB$MDI Multidirectory device
		FB$REC Record-oriented device
		FB$SDI Single-directory device
		FB$SQD Sequential device
		FB$TRM Terminal device
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESL	Expanded string length (bytes)
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
		NB$WCH Wildcard context established
NAM	FID	File identifier
NAM	RSL	Resultant string length (bytes)

5.9 $EXTEND MACRO

The $EXTEND macro calls the EXTEND operation routine to extend the allocation for an open file.

FORMAT

The format for the $EXTEND macro is:

```
$EXTEND fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the EXTEND operation.

For each ALL block that you supply, the EXTEND operation extends the corresponding area as described in the ALL block. You need not supply an ALL block for an area that you do not want to extend, but each supplied ALL block must correspond to an area in the file; this means that you can supply ALL blocks for areas other than area 0 only for an indexed file opened for record access.

To supply XABs (ALL, DAT, KEY, PRO, and SUM blocks) for the EXTEND operation, specify the address of the first XAB in the 1-word XAB field of the FAB; specify the address of the next XAB (if any) in the 1-word NXT field of each XAB; specify 0 in the NXT field of the last XAB.

All KEY blocks must be together in the chain of XABs, and must be in ascending order (by the index reference number in the 1-byte REF field of the KEY block); the index reference numbers need not be consecutive.

All ALL blocks must be together in the chain of XABs, and must be in ascending order (by the area identifier in the 1-byte AID field of the ALL block); the area identifiers need not be consecutive.

Multiple DAT, PRO, or SUM XABs are illegal.

OPTIONS

Internal File Identifier

The EXTEND operation reads the internal file identifier from the 1-word IFI field of the FAB. This is the value written by the CREATE or OPEN operation that opened the file.

Area 0 Extended by FAB

If you supply no ALL blocks, specify the size (in blocks) of the extension in the 2-word ALQ field of the FAB.

If you want the extension to be contiguous within itself (it will not necessarily be contiguous with the file), set the FB$CTG mask in the 1-word FOP field of the FAB; if you do not set this mask, the extension is not necessarily contiguous within itself.

Areas Extended by ALL Blocks

If you supply ALL blocks, the EXTEND operation ignores the ALQ field of the FAB, and extends each area specified in an ALL block. Specify each area to be extended by supplying an ALL block with the area number in the 1-byte AID field of the ALL block. Specify the size of the extension (in blocks) for the area in the 2-word ALQ field of the ALL block.

If you want the area extension to be contiguous within itself (it will not be contiguous with the previous area extent), set the XB$CTG mask in the 1-byte AOP field of the ALL block. If you do not set this mask, the extension will not necessarily be contiguous within itself.

If you want to place the extension at a specific location, specify an alignment mask in the 1-byte ALN field of the ALL block; if you specify 0, the EXTEND operation places the extension at any convenient location. The symbols for alignment masks are:

XB$CYL	Cylinder alignment
XB$LBN	Logical block alignment
XB$VBN	Virtual block alignment

Specify the number of the cylinder, logical block, or virtual block in the 2-word LOC field of the ALL block.

If you specify logical block alignment, and if you want the extension placed only at the logical block you specify, set the XB$HRD mask in the 1-byte AOP field of the ALL block. If you do not set this mask, the EXTEND operation selects an alternate location if the specified location is not available. If you do set this mask, the EXTEND operation returns an error completion if the specified location is not available.

STREAM CONTEXT

The EXTEND operation does not affect stream context.

RETURNED VALUES

Extension Sizes

The EXTEND operation returns the size (in blocks) of each extension it makes. If you extended only area 0 using FAB fields, the EXTEND operation writes the size of the extension in the 2-word ALQ field of the FAB. If you extended areas using ALL blocks, the EXTEND operation writes the size of each area extension in the 2-word ALQ field of the ALL block for the area.

Completion Status and Value

The EXTEND operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-19 lists control block fields that are input to the EXTEND operation. Table 5-20 lists control block fields that are output by the EXTEND operation.

Table 5-19: EXTEND Input Fields

Block	Field	Description
ALL	AID	Area number
ALL	ALN	Area extension alignment request
		XB$CYL Cylinder alignment
		XB$LBN Logical block alignment
		XB$VBN Virtual block alignment
ALL	ALQ	Area allocation extension request size (blocks)
ALL	AOP	Area option mask
		XB$CTG Contiguous area extension request
		XB$HRD Area extension hard location request
ALL	LOC	Area extension location request
ALL	NXT	Next XAB address
DAT	NXT	Next XAB address
FAB	ALQ	File allocation extension request size (blocks)
FAB	FOP	File processing option mask
		FB$CTG Contiguous file extension request
FAB	IFI	Internal FAB identifier
FAB	XAB	XAB address
KEY	NXT	Next XAB address
PRO	NXT	Next XAB address
KEY	REF	Index reference number
SUM	NXT	Next XAB address

Table 5-20: EXTEND Output Fields

Block	Field	Description
ALL	ALQ	Area allocation extension actual size (blocks)
FAB	ALQ	File allocation extension actual size (blocks)
FAB	STS	Completion status code
FAB	STV	Completion status value

5.10 $FIND MACRO (SEQUENTIAL ACCESS)

The $FIND macro calls the FIND operation routine to transfer a record (or part of a record) from a file to an I/O buffer. The FIND operation transfers the entire record if the file is relative or indexed, or if it has blocked records; it may transfer only part of the record if the record spans block boundaries. The FIND operation does not transfer the record to a user buffer.

The target of a sequential-access FIND operation is the next record (for an indexed file, the next record under the current index).

FORMAT

The format for the $FIND macro is:

```
$FIND rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the FIND operation.

OPTIONS

Internal Stream Identifier

The FIND operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Sequential Access

Specify the RB$SEQ code in the 1-byte RAC field of the RAB.

STREAM CONTEXT

The current-record context after a sequential access FIND operation is the found record; the next-record context is the record following the found record (for an indexed file, the next record under the current index). If the FIND operation returns an error completion, the current-record context is undefined, and the next-record context is unchanged.

RETURNED VALUES

RRN

For a relative file or for a sequential disk file with fixed-length records, a sequential-access FIND operation returns the relative record number (RRN) for the found record in the 2-word BKT field of the RAB.

RFA

The FIND operation returns the record file address (RFA) for the found record in the 3-word RFA field of the RAB.

Completion Status and Value

The FIND operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-21 lists control block fields that are input to the FIND operation. Table 5-22 lists control block fields that are output by the FIND operation.

Table 5-21: FIND (Sequential Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RAC	Record access code
		RB$SEQ Sequential access

Table 5-22: FIND (Sequential Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RFA	Record file address
RAB	STS	Completion status code
RAB	STV	Completion status value

5.11 $FIND MACRO (KEY ACCESS)

The $FIND macro calls the FIND operation routine to transfer a record (or part of a record) from a sequential disk file (with fixed-length records), a relative file, or an indexed file to an I/O buffer. The FIND operation transfers the entire record if the file is relative or indexed, or if it has blocked records; it may transfer only part of the record if the record spans block boundaries. The FIND operation does not transfer the record to a user buffer.

The target of a key-access FIND operation is the record having the specified key (under the specified match criterion). For a relative file or for a sequential disk file with fixed-length records, the key is a relative record number (RRN); for an indexed file, the key is an index key under the current index.

FORMAT

The format for the $FIND macro is:

```
$FIND rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the FIND operation.

OPTIONS

Internal Stream Identifier

The FIND operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Key Access

Specify the RB$KEY code in the 1-byte RAC field of the RAB.

Key of Reference (Indexed File)

Specify the key of reference in the 1-byte KRF field of the RAB. The key of reference is the reference number (REF field of KEY block) for the index you want to use for the FIND operation.

Key

Specify a buffer containing the key for the record to be found: specify the address of the key buffer in the 1-word KBF field of the RAB, and specify the size of the key in the 1-byte KSZ field of the RAB.

For a relative file, or for a sequential file with fixed-length records, specify a 4-byte binary relative record number (RRN) as the key, and specify the key size as 0 or 4.

For an indexed file, specify a key of the same type as the key for the current index, and specify a key size no greater than the key size for the current index. For a nonstring key, the specified key size must be the key size defined for the index (or, equivalently, 0); for a string key, if you specify a key size smaller than the key size for the index, the FIND operation searches for a record whose key begins with the specified partial key (under the specified key criterion).

Key Criterion

Specify a key-criterion mask in the 1-word ROP field of the RAB. The symbols for key-criterion masks are:

```
RB$KGE  Greater-than-or-equal key criterion
RB$KGT  Greater-than key criterion
```

If you specify the key-greater criterion, the FIND operation searches for the first record whose key is greater than the key you specify; if you specify the key-greater-or-equal criterion, the FIND operation searches for the first record whose key is greater than or equal to the key you specify; if you specify neither criterion, the FIND operation searches for a record whose key exactly matches the key you specify. (It is illegal to specify both criteria.)

STREAM CONTEXT

The current-record context after a key access FIND operation is the found record; the next-record context is unchanged. If the FIND operation returns an error completion, the current-record context is undefined, and the next-record context is unchanged.

RETURNED VALUES

RFA

The FIND operation returns the record file address (RFA) for the found record in the 3-word RFA field of the RAB.

RRN

For a relative file or for a sequential disk file with fixed-length records, the FIND operation returns the RRN of the found record in the 2-word BKT field of the RAB.

Completion Status and Value

The FIND operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-23 lists control block fields that are input to the FIND operation. Table 5-24 lists control block fields that are output by the FIND operation.

Table 5-23: FIND (Key Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	KBF	Key buffer address
RAB	KRF	Key of reference
RAB	KSZ	Key size (bytes)
RAB	RAC	Record access code
		RB$KEY Key access
RAB	ROP	Record processing option mask
		RB$KGE Greater-than-or-equal key criterion
		RB$KGT Greater-than key criterion

Table 5-24: FIND (Key Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RFA	Record file address
RAB	STS	Completion status code
RAB	STV	Completion status value

5.12 $FIND MACRO (RFA ACCESS)

The $FIND macro calls the FIND operation routine to transfer a record (or part of a record) from a file to an I/O buffer. The FIND operation transfers the entire file if the file is relative or indexed, or if it has blocked records; it may transfer only part of the record if the record spans block boundaries. The FIND operation does not transfer the record to a user buffer.

The target of an RFA-access FIND operation is the record having the record file address (RFA) you specify.

FORMAT

The format for the $FIND macro is:

```
$FIND rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the FIND operation.

OPTIONS

Internal Stream Identifier

The FIND operation reads the internal stream identifier from the 1-word ISI field of the RAB.

RFA Access

Specify the RB$RFA code in the 1-byte RAC field of the RAB.

RFA

Specify the RFA for the record to be found in the 3-word RFA field of the RAB.

STREAM CONTEXT

The current-record context after an RFA access FIND operation is the found record (for an indexed file, in the context of the primary index); the next-record context is unchanged. If the FIND operation returns an error completion, the current-record context is undefined, and the next-record context is unchanged.

RETURNED VALUES

RRN

For a relative file or for a sequential disk file with fixed-length records, the FIND operation returns the RRN of the found record in the 2-word BKT field of the RAB.

Completion Status and Value

The FIND operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-25 lists control block fields that are input to the FIND operation. Table 5-26 lists control block fields that are output by the FIND operation.

Table 5-25: FIND (RFA Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RAC	Record access code
		RB$RFA RFA access
RAB	RFA	Record file address

Table 5-26: FIND (RFA Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.13 $FLUSH MACRO

The $FLUSH macro calls the FLUSH operation routine to write any unwritten buffers for a stream. The FLUSH operation does not affect stream context, except that the current-record context is undefined for a following TRUNCATE or UPDATE operation.

Note one special case: if a file was opened for deferred writing (FB$DFW set in the FOP field of the FAB for the CREATE or OPEN operation), and was not opened for write sharing (FB$WRI cleared in the SHR field of the FAB), then a buffer may be controlled by a different stream, and it will not be written by the FLUSH operation.

FORMAT

The format for the $FLUSH macro is:

```
$FLUSH rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the FLUSH operation.

OPTIONS

Internal Stream Identifier

The FLUSH operation reads the internal stream identifier from the 1-word ISI field of the RAB.

STREAM CONTEXT

The FLUSH operation does not affect stream context, except that the current-record context is undefined for a following TRUNCATE or UPDATE operation.

RETURNED VALUES

Completion Status and Value

The FLUSH operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-27 lists control block fields that are input to the FLUSH operation. Table 5-28 lists control block fields that are output by the FLUSH operation.

Table 5-27: FLUSH Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier

Table 5-28: FLUSH Output Fields

Block	Field	Description
RAB	STS	Completion status code
RAB	STV	Completion status value

5.14 $FREE MACRO

The $FREE macro calls the FREE operation routine to free a locked bucket for a stream. The FREE operation does not affect stream context, except that the current-record context is undefined for a following DELETE, TRUNCATE, or UPDATE operation.

FORMAT

The format for the $FREE macro is:

```
$FREE rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the FREE operation.

OPTIONS

Internal Stream Identifier

The FREE operation reads the internal stream identifier from the 1-word ISI field of the RAB.

STREAM CONTEXT

The FREE operation does not affect stream context, except that the current-record context is undefined for a following DELETE, TRUNCATE, or UPDATE operation.

RETURNED VALUES

Completion Status and Value

The FREE operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-29 lists control block fields that are input to the FREE operation. Table 5-30 lists control block fields that are output by the FREE operation.

Table 5-29: FREE Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier

Table 5-30: FREE Output Fields

Block	Field	Description
RAB	STS	Completion status code
RAB	STV	Completion status value

5.15 $GET MACRO (SEQUENTIAL ACCESS)

The $GET macro calls the GET operation routine to transfer a record from a file to an I/O buffer and to a user buffer.

The target of a sequential-access GET operation depends on whether the previous operation was a FIND operation:

- If the previous operation was a FIND operation, the target of a sequential-access GET operation is the current record.
- If the previous operation was not a FIND operation, the target of a sequential-access GET operation is the next record (for an indexed file, the next record under the current index).

FORMAT

The format for the $GET macro is:

```
$GET rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the GET operation.

OPTIONS

Internal Stream Identifier

The GET operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Sequential Access

Specify the RB$SEQ code in the 1-byte RAC field of the RAB.

User Buffer

Specify a user buffer for the GET operation. The GET operation copies the retrieved record to this buffer if you do not specify locate mode (see next section, Locate Mode); the GET operation may copy the retrieved record to this buffer even if you specify locate mode.

Specify the address of the user buffer in the 1-word UBF field of the RAB, and specify the size (in bytes) of the user buffer in the 1-word USZ field of the RAB.

If the file is in VFC record format, specify the address of a buffer for the fixed-length portion of the record in the 1-word RHB field of the RAB.

Locate Mode

If you want the GET operation to use locate mode (in which the record may not be transferred to the user buffer), set the RB$LOC mask in the 1-word ROP field of the RAB; if you do not set this mask, the record is transferred to the user buffer.

STREAM CONTEXT

The current-record context after a sequential access GET operation is the retrieved record; the next-record context is the record following the retrieved record.

If the GET operation returns an error completion, the current-record context is undefined, and the next-record context is unchanged.

RETURNED VALUES

Record

The GET operation returns the address and size of the retrieved record in the 1-word RBF field of the RAB, and the size (in bytes) of the record in the 1-word RSZ field of the RAB.

If you did not specify locate mode for the GET operation, the record address returned in the RBF field is the address you specified in the UBF field; if you specified locate mode, the record address returned in the RBF field is either the address you specified in the UBF field, or the address of a location in an I/O buffer.

If the file is in VFC format, the GET operation writes the fixed-length portion of the record in the buffer you specified in the RHB field of the RAB.

RRN

For a relative file or for a sequential disk file with fixed-length records, a sequential-access GET operation returns the relative record number (RRN) for the retrieved record in the 2-word BKT field of the RAB.

RFA

The GET operation returns the record file address (RFA) for the retrieved record in the 3-word RFA field of the RAB.

Completion Status and Value

The GET operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-31 lists control block fields that are input to the GET operation. Table 5-32 lists control block fields that are output by the GET operation.

Table 5-31: GET (Sequential Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RAC	Record access code
		RB$SEQ Sequential access
RAB	RHB	VFC control buffer address
RAB	ROP	Record processing option mask
		RB$LOC Locate mode
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-32: GET (Sequential Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RBF	Record buffer address
RAB	RFA	Record file address
RAB	RSZ	Record size (bytes)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.16 $GET MACRO (KEY ACCESS)

The $GET macro calls the GET operation routine to transfer a record from a sequential disk file (with fixed-length records), a relative file, or an indexed file to an I/O buffer and to a user buffer.

The target of a key-access GET operation is the record having the specified key (under the specified match criterion). For a relative file or for a sequential disk file with fixed-length records, the key is a relative record number (RRN); for an indexed file, the key is an index key under the current index.

FORMAT

The format for the $GET macro is:

```
$GET rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the GET operation.

OPTIONS

Internal Stream Identifier

The GET operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Key Access

Specify the RB$KEY code in the 1-byte RAC field of the RAB.

Key of Reference (Indexed File)

Specify the key of reference in the 1-byte KRF field of the RAB. The key of reference is the reference number (REF field of KEY block) for the index you want to use for the GET operation.

Key

Specify a buffer containing the key for the record to be retrieved: specify the address of the key buffer in the 1-word KBF field of the RAB, and specify the size of the key in the 1-byte KSZ field of the RAB.

For a relative file or for a sequential file with fixed-length records, specify a 4-byte binary relative record number (RRN) as the key, and specify the key size as 0 or 4.

For an indexed file, specify a key of the same type as the key for the current index, and specify a key size no greater than the key size for the current index. For a nonstring key, the specified key size must be the key size defined for the index (or, equivalently, 0); for a string key, if you specify a key size smaller than the key size for

the index, the GET operation searches for a record whose key begins with the specified partial key (under the specified key criterion).

Key Criterion

Specify a key-criterion mask in the 1-word ROP field of the RAB. The symbols for key-criterion masks are:

```
RB$KGE  Greater-than-or-equal key criterion
RB$KGT  Greater-than key criterion
```

If you specify the key-greater criterion, the GET operation searches for the first record whose key is greater than the key you specify; if you specify the key-greater-or-equal criterion, the GET operation searches for the first record whose key is greater than or equal to the key you specify; if you specify neither criterion, the GET operation searches for a record whose key exactly matches the key you specify.

User Buffer

Specify a user buffer for the GET operation. The GET operation copies the retrieved record to this buffer if you do not specify locate mode (see next section, Locate Mode); the GET operation may copy the retrieved record to this buffer even if you specify locate mode.

Specify the address of the user buffer in the 1-word UBF field of the RAB, and specify the size (in bytes) of the user buffer in the 1-word USZ field of the RAB.

If the file is in VFC record format, specify the address of a buffer for the fixed-length portion of the record in the 1-word RHB field of the RAB.

Locate Mode

If you want the GET operation to use locate mode (in which the record may not be transferred to the user buffer), set the RB$LOC mask in the 1-word ROP field of the RAB; if you do not set this mask, the record is transferred to the user buffer.

STREAM CONTEXT

The current-record context after a key access GET operation is the retrieved record; the next-record context is the record following the retrieved record.

If the GET operation returns an error completion, the current-record context is undefined, and the next-record context is unchanged.

RETURNED VALUES

Record

The GET operation returns the address and size of the retrieved record in the 1-word RBF field of the RAB, and the size (in bytes) of the record in the 1-word RSZ field of the RAB.

If you did not specify locate mode for the GET operation, the record address returned in the RBF field is the address you specified in the UBF field. If you specified locate mode, the record address returned in the RBF field is either the address you specified in the UBF field, or the address of a location in an I/O buffer.

If the file is in VFC format, the GET operation writes the fixed-length portion of the record in the buffer you specified in the RHB field of the RAB.

RRN

For a relative file or for a sequential disk file with fixed-length records, a key-access GET operation returns the relative record number (RRN) for the retrieved record in the 2-word BKT field of the RAB.

RFA

The GET operation returns the record file address (RFA) for the retrieved record in the 3-word RFA field of the RAB.

Completion Status and Value

The GET operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-33 lists control block fields that are input to the GET operation. Table 5-34 lists control block fields that are output by the GET operation.

Table 5-33: GET (Key Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	KBF	Key buffer address
RAB	KRF	Key of reference
RAB	KSZ	Key size (bytes)
RAB	RAC	Record access code
		RB$KEY Key access
RAB	RHB	VFC control buffer address
RAB	ROP	Record processing option mask
		RB$KGE Greater-than-or-equal key criterion
		RB$KGT Greater-than key criterion
		RB$LOC Locate mode
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-34: GET (Key Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RBF	Record buffer address
RAB	RFA	Record file address
RAB	RSZ	Record size (bytes)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.17 $GET MACRO (RFA ACCESS)

The $GET macro calls the GET operation routine to transfer a record from a file to an I/O buffer and to a user buffer.

The target of an RFA-access GET operation is the record having the record file address (RFA) you specify.

FORMAT

The format for the $GET macro is:

```
$GET rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the GET operation.

OPTIONS

Internal Stream Identifier

The GET operation reads the internal stream identifier from the 1-word ISI field of the RAB.

RFA Access

Specify the RB$RFA code in the 1-byte RAC field of the RAB.

RFA

Specify the RFA for the record to be retrieved in the 3-word RFA field of the RAB.

User Buffer

Specify a user buffer for the GET operation. The GET operation copies the retrieved record to this buffer if you do not specify locate mode (see next section, Locate Mode); the GET operation may copy the retrieved record to this buffer even if you specify locate mode.

Specify the address of the user buffer in the 1-word UBF field of the RAB, and specify the size (in bytes) of the user buffer in the 1-word USZ field of the RAB.

If the file is in VFC record format, specify the address of a buffer for the fixed-length portion of the record in the 1-word RHB field of the RAB.

Locate Mode

If you want the GET operation to use locate mode (in which the record may not be transferred to the user buffer), set the RB$LOC mask in the 1-word ROP field of the RAB; if you do not set this mask, the record is transferred to the user buffer.

STREAM CONTEXT

The current-record context after an RFA access GET operation is the retrieved record (for an indexed file, in the context of the primary index); the next-record context is the record following the retrieved record. If the GET operation returns an error completion, the current-record context is undefined, and the next-record context is unchanged.

RETURNED VALUES

Record

The GET operation returns the address and size of the retrieved record in the 1-word RBF field of the RAB, and the size (in bytes) of the record in the 1-word RSZ field of the RAB.

If you did not specify locate mode for the GET operation, the record address returned in the RBF field is the address you specified in the UBF field. If you specified locate mode, the record address returned in the RBF field is either the address you specified in the UBF field, or the address of a location in an I/O buffer.

If the file is in VFC format, the GET operation writes the fixed-length portion of the record in the buffer you specified in the RHB field of the RAB.

RRN

For a relative file or for a sequential disk file with fixed-length records, an RFA-access GET operation returns the relative record number (RRN) for the retrieved record in the 2-word BKT field of the RAB.

Completion Status and Value

The GET operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-35 lists control block fields that are input to the GET operation. Table 5-36 lists control block fields that are output by the GET operation.

Table 5-35: GET (RFA Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RAC	Record access code
		RB$RFA RFA access
RAB	RFA	Record file address
RAB	RHB	VFC control buffer address
RAB	ROP	Record processing option mask
		RB$LOC Locate mode
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-36: GET (RFA Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RBF	Record buffer address
RAB	RSZ	Record size (bytes)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.18 $OPEN MACRO

The $OPEN macro calls the OPEN operation routine to open a file for processing by the calling task.

FORMAT

The format for the $OPEN macro is:

```
$OPEN fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the OPEN operation.

If you supply a NAM block and specify open by NAM block, the OPEN operation reads NAM block fields to obtain identifiers for the target file.

To supply a NAM block for the OPEN operation, specify the address of the NAM block in the 1-word NAM field of the FAB.

For each ALL block that you supply, the OPEN operation fills its fields with values describing the corresponding area (if any) of the file. You need not supply an ALL block for every area of the file. (If you are opening the file for block access, the OPEN operation writes information describing the file as a whole in the all block for area 0.)

For each KEY block that you supply, the OPEN operation fills its fields with values describing the corresponding index (if any) for the file. You need not supply a KEY block for every index of the file. (If you are opening the file for block access, the OPEN operation does not write in KEY blocks.)

If you supply a PRO block for a disk file, the OPEN operation fills its fields with values showing the owner and protection for the file.

If you supply a DAT block for a disk file, the OPEN operation fills its fields with values showing the creation date, expiration date, revision date, and revision number for the file.

If you supply a SUM block for a relative or indexed file, the OPEN operation fills its fields with values showing the number of areas and indexes for the file, and with its prologue version number. (If you are opening the file for block access, the OPEN operation returns the number of areas and number of keys as 0, and does not return the prologue version number.)

This information is especially useful if you do not know how many areas or keys an indexed file has when you open it. If you supply a SUM block for the OPEN operation, you can get the number of areas and number of indexes from its fields, and then supply the correct number of ALL blocks and KEY blocks for the DISPLAY operation.

To supply XABs (ALL, DAT, KEY, PRO, and SUM blocks) for the OPEN operation, specify the address of the first XAB in the 1-word XAB field of the FAB; specify the address of the next XAB (if any) in the 1-word NXT field of each XAB; specify 0 in the NXT field of the last XAB.

All KEY blocks must be together in the chain of XABs, and must be in ascending order (by the index reference number in the 1-byte REF field of the KEY block); the index reference numbers need not be consecutive.

All ALL blocks must be together in the chain of XABs, and must be in ascending order (by the area identifier in the 1-byte AID field of the ALL block); the area identifiers need not be consecutive.

Multiple DAT, PRO, or SUM XABs are illegal.

OPTIONS

Open by File Specification

If you want to open a file by its file specification, clear the FB$FID mask in the 1-word FOP field of the FAB; this causes the OPEN operation to use the fully qualified file specification to determine the target file.

Specify the address of the file string in the 1-word FNA field of the FAB; specify the size (in bytes) of the file string in the 1-byte FNS field of the FAB. If you specify 0 in the FNS field, the OPEN operation uses no file string.

Specify the address of the default string in the 1-word DNA field of the FAB; specify the size (in bytes) of the default string in the 1-byte DNS field of the FAB. If you specify 0 in the DNS field, the OPEN operation uses no default string.

Open by NAM Block

If you want to specify the target file by its identifiers, set the FB$FID mask in the 1-word FOP field of the FAB, and supply a filled-in NAM block (one whose fields were written by a CREATE or OPEN operation for the file); this causes the OPEN operation to use any nonzero identifiers in the NAM block to override corresponding elements in the file specification provided.

These identifiers are the device identifier (in the 2-word DVI field of the NAM block), the directory identifier (in the 3-word DID field of the NAM block), and the file identifier (in the 3-word FID field of the NAM block).

Open with Wildcard Context

If you want to open a file that was found by a wildcard SEARCH operation (using the FAB and NAM block that the SEARCH operation used), set the FB$FID mask in the 1-word FOP field of the FAB; this causes the OPEN operation to open the file without altering wildcard context.

Expanded String Buffer

If you want the OPEN operation to return the expanded string for the opened file, provide a buffer for the string. Specify the address of the expanded string buffer in the 1-word ESA field of the NAM block and its size (in bytes) in the 1-byte ESS field of the NAM block; if you specify 0 in the ESS field, the OPEN operation does not return the expanded string.

Key Name Buffer

If you want the key name string for an index returned to a buffer, supply a KEY block for the index. Specify the index reference number in the 1-byte REF field of the KEY block, and specify the address of a 32-byte buffer in the 1-word KNM field of the KEY block. If you do not supply a KEY block for an index, or if you specify 0 in its KNM field, the OPEN operation does not return the key name string.

While-Open Default Extension Sizes

If you want to override the default extension size for the file while it is open, specify the while-open default file extension size (in blocks) in the 1-word DEQ field of the FAB. If you specify 0, the OPEN operation does not establish a while-open default extension size for the file; instead, it uses the permanent default extension size.

The while-open default extension size for a file remains in force while the file is open, but does not change the file extension size established when the file was created.

Private Buffer Pool

If you want the OPEN operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the OPEN operation uses the central buffer pool.

The pool that the OPEN operation uses is also used by the DISPLAY and EXTEND operations, and by stream and record or block operations while the file is open.

Logical Channel

Specify the logical channel for the OPEN operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

The logical channel that the OPEN operation uses is also used by the DISPLAY and EXTEND operations, and by stream and record or block operations while the file is open.

Retrieval Pointers

Specify the number of retrieval pointers for the open file in the 1-byte RTV field of the FAB. If you specify 0, the OPEN operation uses the operating system default; if you specify -1, the CREATE operation maps as much of the file as possible with each retrieval pointer.

Requested-Access

Specify one or more requested-access masks in the 1-byte FAC field of the FAB. This mask determines the access that the opening program has while the file is open. If you specify no requested-access mask, find/get access is allowed (the OPEN operation uses the mask FB$GET).

The symbols for requested-access masks are:

FB$DEL	Request find/get/delete access
FB$GET	Request find/get access
FB$PUT	Request put access
FB$REA	Request block read access
FB$TRN	Request find/get/truncate access
FB$UPD	Request find/get/update access
FB$WRT	Request block write access

Note that FB$REA and FB$WRT override any record access requested.

Access Sharing

Specify the kinds of access that your program will share with other programs by setting an access-sharing mask in the 1-byte SHR field of the FAB. The symbols for access-sharing masks are:

FB$GET	Share find/get access
FB$NIL	No access sharing
FB$UPI	Share any access (user-provided interlock)
FB$WRI	Share find/get/put/update/delete access

The kinds of access sharing are:

- **Shared read access**

  Your program is willing to allow other programs to read the file, but not to write it.

- **Shared write access**

  Your program is willing to allow other programs to both read and write the file. Shared write access is not allowed for a sequential file unless the file has undefined record format and your program opens the file for block access; shared write access is also not allowed for a relative or indexed file that your program opens for block access. In such cases, RMS-11 automatically converts the shared write access specification to a shared read access specification internally.

- **No shared access**

  Your program is not willing to allow other programs to either read or write the file. RMS-11 does, however, allow other programs to read the file unless your program also requests some form of write access.

- **User-provided interlocking**

  Your program and other cooperating programs define and enforce their own access interlocking; RMS-11 does not check access sharing. User-provided interlocking is allowed only if your program has requested read access for a sequential file; otherwise, the FB$UPI mask is ignored (but other masks are honored).

Deferred Writing

If you want deferred buffer writing for the open file, set the FB$DFW mask in the 1-word FOP field of the FAB; This means that RMS-11 does not necessarily write its buffers during a write-type operation (DELETE, PUT, or UPDATE), but instead writes buffers only when it

needs them for other operations (or when your program executes the FLUSH operation for the stream).

If you do not set the FB$DFW mask, the DELETE, PUT, and UPDATE operations write buffers to the file immediately.

Note that record operations always use a form of deferred buffer writing for sequential files, and that block operations never use deferred buffer writing. Therefore you need only decide whether to use deferred writing for a record stream to a relative or indexed file.

File Locking

If you want the file to remain unlocked even if it is closed abnormally, set the FB$DLK mask in the l-word FOP field of the FAB; if you do not set this mask, the operating system locks the file if it is closed abnormally.

RETURNED VALUES

Internal File Identifier

The OPEN operation writes an internal file identifier in the l-word IFI field of the FAB. (The CLOSE operation clears the internal file identifier.)

The CLOSE, CONNECT, DISPLAY, and EXTEND operations read the internal file identifier; do not alter the IFI field while the file is open.

Device Characteristics

The OPEN operation returns device characteristics as masks in the l-byte DEV field of the FAB. The device characteristics are:

- Printer or terminal (indicated by the set FB$CCL mask in the l-byte DEV field of the FAB and the set FB$REC mask in the l-byte DEV field of the FAB; for a terminal, the FB$TRM mask in the l-byte DEV field of the FAB is also set); RMS-ll treats a printer or terminal as a unit-record device.

- Disk, DECtape, or DECTAPE II (indicated by the set FB$MDI mask in the l-byte DEV field of the FAB); RMS-ll treats a disk, DECtape, or DECTAPE II as a disk device.

- Unit-record device (indicated by the set FB$REC mask in the l-byte DEV field of the FAB).

- Non-ANSI magtape or cassette tape (indicated by the set FB$SDI mask in the l-byte DEV field of the FAB and the set FB$REC mask in the l-byte DEV field of the FAB); RMS-ll treats a non-ANSI magtape or a cassette tape as a unit-record device.

- ANSI-format magtape (indicated by the set FB$SQD mask in the l-byte DEV field of the FAB).

Device, Directory, and File Identifiers

If you supply a NAM block, the OPEN operation writes a device identifier in the 2-word DVI field of the NAM block, a directory identifier in the 3-word DID field of the NAM block, and a file identifier in the 3-word FID field of the NAM block.

Expanded String

If you specify a buffer for the expanded string for the file (ESA and ESS fields in the NAM block), the OPEN operation writes the expanded string for the opened file in this buffer, and writes the length (in bytes) of the string in the 1-byte ESL field of the NAM block.

File Allocation, Bucket Size, and Contiguity

The OPEN operation writes the file allocation size (in blocks) in the 2-word ALQ field of the FAB, and the file bucket size or largest area bucket size (in blocks) in the 1-byte BKS field of the FAB. If the file is contiguous, the OPEN operation sets the FB$CTG mask in the 1-word FOP field of the FAB.

Extension size

The OPEN operation writes the current default extension size for the open file in the 1-word DEQ field of the FAB.

File Organization

The OPEN operation writes the file organization code in the 1-byte ORG field of the FAB. The symbols for file organization codes are:

FB$IDX	Indexed file organization
FB$REL	Relative file organization
FB$SEQ	Sequential file organization

Record Format

The OPEN operation writes the record format code in the 1-byte RFM field of the FAB. The symbols for record format codes are:

FB$FIX	Fixed-length record format
FB$STM	Stream record format
FB$UDF	Undefined record format
FB$VAR	Variable-length record format
FB$VFC	VFC record format

If the record format is VFC, the OPEN operation writes the size (in bytes) of the VFC header field in the 1-byte FSZ field of the FAB; otherwise it writes 0 in the FSZ field.

Blocked Records (Sequential Disk File)

If the file was created specifying blocked records, the OPEN operation sets the FB$BLK mask in the 1-byte RAT field of the FAB. (The OPEN operation sets the mask if it was set when the file was created, even if the file is not a sequential file; preservation of this mask allows you to copy a sequential file to a file of a different organization and back without losing the blocked-record characteristic.)

Record-Output Handling

The OPEN operation writes the record-output mask in the 1-byte RAT field of the FAB. The symbols for record-output masks are:

FB$CR	Add CRLF to print record (LF-record-CR)
FB$FTN	FORTRAN-style carriage-control character in record
FB$PRN	VFC print record handling

Record Size

The OPEN operation writes the maximum permitted record size (in bytes) in the 1-word MRS field of the FAB.

Maximum Record Number

If the file is a relative file (FB$REL in the ORG field), the OPEN operation writes the maximum record number in the 2-word MRN field of the FAB (unless you are opening the file for block access).

Longest Record Length

The OPEN operation writes the length of the longest record in the file in the 1-word LRL field of the FAB; this value is meaningful only for sequential files.

Area Descriptions

For each ALL block that you supply, the OPEN operation writes a description in its fields of the corresponding area of the file (unless you are opening the file for block access). Area 0 is described in the ALL block containing 0 in its AID field, area 1 is described in the ALL block containing 1 in its AID field, and so forth.

The OPEN operation writes three sizes for a file area: the size (in blocks) of the unused portion of the area in the 2-word ALQ field of the ALL block, the default area extension size (in blocks) in the 1-word DEQ field of the ALL block, and the area bucket size (in blocks) in the 1-byte BKZ field of the ALL block. (If you are opening the file for block access, only the ALL block for area 0 is written, and the ALL block contains the current file allocation size, default file extension size, and file bucket size.)

The OPEN operation clears the 1-byte ALN field of the ALL block and the XB$HRD mask in the 1-byte AOP field of the ALL block. If you are opening a sequential or relative file for record access, the OPEN operation sets the XB$CTG mask in the 1-byte AOP field of the ALL block if the file is contiguous; otherwise it clears the entire 1-byte AOP field of the ALL block.

Key Descriptions

For each KEY block that you supply, the OPEN operation writes a description in its fields of the corresponding index of the file. (The OPEN operation does not write in KEY blocks if you are opening the file for block access.)

The primary index is described in the KEY block containing 0 in its REF field, the first alternate index is described in the KEY block containing 1 in its REF field, and so forth.

The OPEN operation writes the key data type code in the 1-byte DTP field of the KEY block. The symbols for key data type codes are:

XB$BN2	16-bit unsigned integer
XB$BN4	32-bit unsigned integer
XB$IN2	15-bit signed integer
XB$IN4	31-bit signed integer
XB$PAC	Packed decimal number
XB$STG	String

The OPEN operation writes the sizes of key segments in the 8-byte SIZ field of the KEY block. The size (in bytes) of the first key segment is in the first byte of the SIZ field, the size of the second segment is in the second byte of the SIZ field, and so forth. If the key has fewer than eight segments, the first byte containing 0 establishes the number of key segments.

The OPEN operation writes the positions of key segments in the 8-word POS field of the KEY block. The position (leftmost position is 0) of the first key segment is in the first word of the POS field, the position of the second segment is in the second word of the POS field, and so forth. If the key has fewer than eight segments, the remaining words of the POS field contain unpredictable values.

The OPEN operation writes a key flags mask in the 1-byte FLG field of the KEY block. The symbols for key flags masks are:

XB$CHG	Record key changes allowed on update
XB$DUP	Duplicate record keys allowed
XB$INI	No entries yet made in index
XB$NUL	Null record keys not indexed

The OPEN operation writes the null-key character in the 1-byte NUL field of the KEY block; this character is meaningful only if the XB$NUL mask in the FLG field is set and if the key is a string key (XB$STG in the DTP field).

The OPEN operation writes area numbers for the index: the area for the data level in the 1-byte DAN field of the KEY block, the area for the lowest index level in the 1-byte LAN field of the KEY block, and the area for higher index levels in the 1-byte IAN field of the KEY block.

The OPEN operation writes bucket fill numbers for the index areas: the fill number for the data area in the 1-word DFL field of the KEY block, and the fill number for the index areas in the 1-word IFL field of the KEY block.

The OPEN operation writes bucket sizes for index areas: the data area bucket size (in blocks) in the 1-byte DBS field of the KEY block, and the index area bucket size (in blocks) in the 1-byte IBS field of the KEY block.

The OPEN operation writes virtual block numbers for the index areas: the virtual block number for the first data bucket in the 2-word DVB field of the KEY block, and the virtual block number of the root index bucket in the 2-word RVB field of the KEY block.

The OPEN operation writes the number of levels in the index (not including the data level) in the 1-byte LVL field of the KEY block.

The OPEN operation writes the minimum size (in bytes) of a record that contains the key for the index in the 1-word MRL field of the KEY block.

The OPEN operation writes key segment information for the index: the number of key segments in the 1-byte NSG field of the KEY block, and the total key size (sum of segments, in bytes) in the 1-byte TKS field of the KEY block.

File Owner and Protection (Disk File)

If the file is a disk file, and if you supply a PRO block, the OPEN operation writes the project (or group) portion of the file owner code in the 1-word PRJ field of the PRO block, the programmer (or member) portion of the file owner code in the 1-word PRG field of the PRO block, and the file protection code in the 1-word PRO field of the PRO block.

File Dates

If you supply a DAT block, the OPEN operation writes four values in its fields: the creation date in the 4-word CDT field of the DAT block, the expiration date in the 4-word EDT field of the DAT block, the revision date in the 4-word RDT field of the DAT block, and the revision number (number of times the file has been opened for write access and then closed) in the 1-word RVN field of the DAT block.

File Summary Information

If you supply a SUM block and are opening an indexed file, the OPEN operation writes three values in its fields: the number of file areas in the 1-byte NOA field of the SUM block, the number of file indexes in the 1-byte NOK field of the SUM block, and the prologue version number (for a relative or indexed file) in the 1-word PVN field of the SUM block. (If you are opening the file for block access, the OPEN operation returns the number of areas and the number of keys as 0, and does not return the prologue version number.)

File Specification Characteristics

The OPEN operation sets masks in the l-word FNB field of the NAM block to show which file specification elements were present in the file string and default string. These masks and their meanings are:

NB$NOD	Node in file string or default string
NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$QUO	Quoted string in file string or default string
NB$NAM	File name in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Wildcard Context Information

If you cleared the FB$FID mask, the OPEN operation clears the NB$WCH mask in the l-word FNB field of the NAM block and the l-byte RSL field of the NAM block; this shows that no wildcard context information exists after the operation and that no resultant string was returned. If you set the FB$FID mask, the OPEN operation does not alter the NB$WCH mask, and (if the NB$WCH mask is set) does not alter the RSL field.

Completion Status and Value

The OPEN operation returns completion status in the l-word STS field of the FAB and returns a completion value in the l-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-39 lists control block fields that are input to the OPEN operation. Table 5-40 lists control block fields that are output by the OPEN operation.

Table 5-39: OPEN Input Fields

Block	Field	Description
ALL	AID	Area number
ALL	NXT	Next XAB address
DAT	NXT	Next XAB address
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DEQ	While-open file default extension size (blocks)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)

(continued on next page)

Table 5-39 (cont.): OPEN Input Fields

Block	Field	Description
FAB	FAC	Requested access mask
		FB$DEL Request find/get/delete access
		FB$GET Request find/get access
		FB$PUT Request put access
		FB$REA Request block read access
		FB$TRN Request find/get/truncate access
		FB$UPD Request find/get/update access
		FB$WRT Request block write access
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	FOP	File processing option mask
		FB$DFW Defer writing
		FB$DLK No file locking on abnormal close
		FB$FID Use information in NAM block
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
FAB	RTV	Retrieval pointer count
FAB	SHR	Shared access mask
		FB$GET Share find/get access
		FB$NIL No access sharing
		FB$UPI Share any access (user-provided interlock)
		FB$WRI Share find/get/put/update/delete access
FAB	XAB	XAB address
KEY	KNM	Key name buffer address
KEY	NXT	Next XAB address
KEY	REF	Index reference number
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESA	Expanded string buffer address
NAM	ESS	Expanded string buffer size (bytes)
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$WCH Wildcard context established
PRO	NXT	Next XAB address
SUM	NXT	Next XAB address

Table 5-40: OPEN Output Fields

Block	Field	Description
ALL	ALN	Area alignment mask
ALL	ALQ	Unused area allocation size (blocks)
ALL	AOP	Area option mask
		XB$CTG Contiguous area
		XB$HRD Hard area location (cleared)
ALL	BKZ	Area bucket size (blocks)
ALL	DEQ	Area default extension size (blocks)
DAT	CDT	File creation date
DAT	EDT	File expiration date
DAT	RDT	File revision date
DAT	RVN	File revision number
FAB	ALQ	Current file allocation (blocks)
FAB	BKS	File bucket size (blocks)
FAB	DEQ	Current file default extension size (blocks)
FAB	DEV	Device characteristic mask
		FB$CCL Carriage-control device
		FB$MDI Multidirectory device
		FB$REC Record-oriented device
		FB$SDI Single-directory device
		FB$SQD Sequential device
		FB$TRM Terminal device
FAB	FOP	File processing option mask
		FB$CTG Contiguous file
FAB	FSZ	Fixed control area size for VFC records (bytes)
FAB	IFI	Internal FAB identifier
FAB	LRL	Longest record length
FAB	MRN	Maximum record number
FAB	MRS	Maximum record size (bytes)
FAB	ORG	File organization code
		FB$SEQ Sequential file organization
		FB$REL Relative file organization
		FB$IDX Indexed file organization
FAB	RAT	Record handling mask
		FB$BLK Blocked records
		FB$CR Add CRLF to print record (LF-record-CR)
		FB$FTN FORTRAN-style carriage-control character in record
		FB$PRN VFC print record handling

(continued on next page)

Table 5-40 (cont.): OPEN Output Fields

Block	Field	Description
FAB	RFM	Record format code
		FB$UDF Undefined record format
		FB$FIX Fixed-length record format
		FB$VAR Variable-length record format
		FB$VFC VFC record format
		FB$STM Stream record format
FAB	STS	Completion status code
FAB	STV	Completion status value
KEY	DAN	Data area number
KEY	DBS	Data area bucket size (blocks)
KEY	DFL	Data bucket fill factor
KEY	DTP	Key data type code
		XB$BN2 16-bit unsigned integer
		XB$BN4 32-bit unsigned integer
		XB$IN2 15-bit signed integer
		XB$IN4 31-bit signed integer
		XB$PAC Packed decimal number
		XB$STG String
KEY	DVB	First data bucket virtual block number
KEY	FLG	Index option mask
		XB$CHG Record key changes allowed on update
		XB$DUP Duplicate record keys allowed
		XB$INI No entries yet made in index
		XB$NUL Null record keys not indexed
KEY	IAN	Higher level index area number
KEY	IBS	Index area bucket size (blocks)
KEY	IFL	Index bucket fill factor
KEY	LAN	Lowest index level area number
KEY	LVL	Number of index levels (not including data level)
KEY	MRL	Minimum length of record containing key (bytes)
KEY	NSG	Key segment count
KEY	NUL	Null key character
KEY	POS	Key segment positions
KEY	RVB	Root index bucket virtual block number
KEY	SIZ	Key segment sizes (bytes)
KEY	TKS	Total key size (sum of key segment sizes) (bytes)
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESL	Expanded string length (bytes)
NAM	FID	File identifier

(continued on next page)

Table 5-40 (cont.): OPEN Output Fields

Block	Field	Description
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
		NB$WCH Wildcard context established
PRO	PRG	Programmer or member portion of file owner code
PRO	PRJ	Project or group portion of file owner code
PRO	PRO	File protection code
SUM	NOA	Number of areas
SUM	NOK	Number of indexes
SUM	PVN	Prologue version number

5.19 $PARSE MACRO

The $PARSE macro calls the PARSE operation routine to analyze a file specification.

FORMAT

The format for the $PARSE macro is:

```
$PARSE fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the PARSE operation.

If you supply a NAM block for the PARSE operation, the operation routine writes file information in its fields. This information is suitable as input to subsequent wildcard SEARCH operations.

To supply a NAM block for the PARSE operation, specify the address of the NAM block in the 1-word NAM field of the FAB.

OPTIONS

File Specification

The PARSE operation constructs the merged string for the target file from the file string, the default string, RMS-11 defaults, and system defaults.

Specify the address of the file string in the 1-word FNA field of the FAB. Specify the size (in bytes) of the file string in the 1-byte FNS field of the FAB; if you specify 0 in the FNS field, the PARSE operation uses no file string.

Specify the address of the default string in the 1-word DNA field of the FAB. Specify the size (in bytes) of the default string in the 1-byte DNS field of the FAB; if you specify 0 in the DNS field, the PARSE operation uses no default string.

Expanded String Buffer

If you want the PARSE operation to return the expanded string for the file, provide a buffer for the string. If you want subsequent wildcard SEARCH operations to use the results of the PARSE operation, you must provide an expanded string buffer.

Specify the address of the expanded string buffer in the 1-word ESA field of the NAM block. Specify the size (in bytes) of the expanded string buffer in the 1-byte ESS field of the NAM block; if you specify 0 in the ESS field, the PARSE operation does not return the expanded string.

Private Buffer Pool

If you want the PARSE operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the PARSE operation uses the central buffer pool.

Logical Channel

Specify the logical channel for the PARSE operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

RETURNED VALUES

Wildcard Initialization

If you supplied a NAM block to be initialized for wildcard SEARCH operations, the PARSE operation clears several fields: the 3-word DID field of the NAM block, the 1-byte RSL field of the NAM block, the 1-word WCC field of the NAM block, and the 1-word WDI field of the NAM block. These cleared fields are part of the initialization for subsequent wildcard SEARCH operations.

The PARSE operation writes a match-pattern (for subsequent wildcard SEARCH operations) in the expanded string buffer, and writes the length (in bytes) of the expanded string in the 1-byte ESL field of the NAM block.

The PARSE operation sets the NB$WCH mask in the 1-word FNB field of the NAM block, showing that wildcard information in the NAM block is initialized.

File Specification Characteristics

The PARSE operation sets masks in the 1-word FNB field of the NAM block to show which file specification elements were present in the file string and default string. These masks and their meanings are:

NB$NOD	Node in file string or default string
NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$QUO	Quoted string in file string or default string
NB$NAM	File name in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Expanded String

If you supply a NAM block, and if the input file specification string does not contain wildcard characters, the PARSE operation writes the expanded string in the expanded string buffer; this string is a fully qualified file specification except that the file version number (if any) from the input file specification is unchanged.

Completion Status and Value

The PARSE operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-41 lists control block fields that are input to the PARSE operation. Table 5-42 lists control block fields that are output by the PARSE operation.

Table 5-41: PARSE Input Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
NAM	ESA	Expanded string buffer address
NAM	ESS	Expanded string buffer size (bytes)

Table 5-42: PARSE Output Fields

Block	Field	Description
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier (cleared)
NAM	ESL	Expanded string length (bytes)
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WCH Wildcard context established
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
NAM	RSL	Resultant string length (bytes) (cleared)
NAM	WCC	Wildcard context (cleared)
NAM	WDI	Wildcard directory context (cleared)

5.20 $PUT MACRO (SEQUENTIAL ACCESS)

The $PUT macro calls the PUT operation routine to transfer a record from a user buffer to an I/O buffer and to a file.

The target of a sequential-access PUT operation depends on the file organization:

- For a sequential file, the target of a sequential-access PUT operation is the end-of-file, and the next-record context must be the end-of-file.
- For a relative file, the target of a sequential-access PUT operation is the next cell (as determined by the next-record context or by the context of an immediately preceding sequential access PUT operation).
- For an indexed file, a sequential-access PUT operation has no target; the PUT operation inserts the record and updates indexes. If the immediately preceding operation was also a sequential access PUT operation, the primary key value in your record must be greater than or equal to the primary key value of the preceding record.

FORMAT

The format for the $PUT macro is:

```
$PUT rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the PUT operation.

OPTIONS

Internal Stream Identifier

The PUT operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Sequential Access

Specify the RB$SEQ code in the 1-byte RAC field of the RAB.

Record

Specify the address of the record to be transferred in the 1-word RBF field of the RAB, and the size (in bytes) of the record in the 1-word RSZ field of the RAB.

If the record is in VFC format, specify the address of the fixed-length portion of the record in the 1-word RHB field of the RAB. If you specify 0 in this field, the record header will be null-filled.

Locate Mode

For a sequential file, if you want the PUT operation to use locate mode, specify the address of the user buffer in the 1-word UBF field of the RAB, specify the maximum size of the record for the **next** PUT operation in the 1-word USZ field of the RAB, and set the RB$LOC mask in the 1-word ROP field of the RAB.

The PUT operation returns (in the RBF field) the address of a location where your program can build the next record for output. The **maximum next record size** that you specify in the USZ field determines whether the next record can fit into an I/O buffer.

Bucket Fill Number Honoring

If you want the PUT operation to honor bucket fill numbers for the file and its areas, set the RB$LOA mask in the 1-word ROP field of the RAB. If you do not set this mask, the PUT operation fills buckets without regard to bucket fill numbers.

Update Existing Record (Relative File)

If you want to transfer the record to a cell in a relative file even if the cell contains a record, set the RB$UIF mask in the 1-word ROP field of the RAB. If you do not set this mask, and if the cell already contains a record, the PUT operation returns an error completion and does not transfer the record.

Mass Insertion (Indexed File)

For an indexed file, using mass-insertion mode for a series of PUT operations speeds up the insertion of a series of records. To use mass-insertion mode for a series of records, set the RB$MAS mask in the 1-word ROP field of the RAB for each PUT operation in the series.

STREAM CONTEXT

The current-record and next-record contexts after a sequential access PUT operation are undefined.

RETURNED VALUES

Next Record Buffer

If you specified locate mode for the PUT operation, the PUT operation returns the address of a location where your program can build the next record for output in the 1-word RBF field of the RAB. This address gives a location in the I/O buffer (if there is room for another record there), or the location of your user buffer (if not).

RRN

For a relative file or for a sequential disk file with fixed-length records, a sequential-access PUT operation returns the relative record number (RRN) for the inserted record in the 2-word BKT field of the RAB.

RFA

The PUT operation returns the record file address (RFA) for the inserted record in the 3-word RFA field of the RAB.

Completion Status and Value

The PUT operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-43 lists control block fields that are input to the PUT operation. Table 5-44 lists control block fields that are output by the PUT operation.

Table 5-43: PUT (Sequential Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RAC	Record access code
		RB$SEQ Sequential access
RAB	RBF	Record buffer address
RAB	RHB	VFC control buffer address
RAB	ROP	Record processing option mask
		RB$LOA Honor bucket fill numbers
		RB$LOC Locate mode
		RB$MAS Mass insert
		RB$UIF Update if record exists
RAB	RSZ	Record size (bytes)
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-44: PUT (Sequential Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RFA	Record file address
RAB	RBF	Record buffer address
RAB	STS	Completion status code
RAB	STV	Completion status value

5.21 $PUT MACRO (KEY ACCESS)

The $PUT macro calls the PUT operation routine to transfer a record from a user buffer to an I/O buffer and to a sequential disk file (with fixed-length records), a relative file, or an indexed file.

The target of a key-access PUT operation depends on the file organization:

- For a sequential disk file (with fixed-length records) or a relative file, the key is a relative record number (RRN), and the target of a key-access PUT operation is the cell specified by the RRN.
- For an indexed file, a key-access PUT operation has no target; the PUT operation inserts the record and updates indexes.

FORMAT

The format for the $PUT macro is:

```
$PUT rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the PUT operation.

OPTIONS

Internal Stream Identifier

The PUT operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Key Access

Specify the RB$KEY code in the 1-byte RAC field of the RAB.

Record

Specify the address of the record to be transferred in the 1-word RBF field of the RAB, and the size (in bytes) of the record in the 1-word RSZ field of the RAB.

If the record is in VFC format, specify the address of the fixed-length portion of the record in the 1-word RHB field of the RAB. If you specify 0 in this field, the record header will be null-filled.

Record Buffer

Specify a record buffer for the PUT operation; specify the address of the record buffer in the 1-word UBF field of the RAB; specify the size (in bytes) of the record buffer in the 1-word USZ field of the RAB.

Note that the value in the UBF field will be used (copied to the RBF field) only if you specify locate mode. A request for locate mode is otherwise ignored for a key access PUT operation.

RRN

For a relative file or for a sequential disk file with fixed-length records, specify a 4-byte relative record number (RRN) in the 1-word KBF field of the RAB, and specify 0 or 4 in the 1-byte KSZ field of the RAB.

Bucket Fill Number Honoring

If you want the PUT operation to honor bucket fill numbers for the file and its areas, set the RB$LOA mask in the 1-word ROP field of the RAB. If you do not set this mask, the PUT operation fills buckets without regard to bucket fill numbers.

Update Existing Record (Relative File)

If you want to transfer the record to a cell in a relative file even if the cell contains a record, set the RB$UIF mask in the 1-word ROP field of the RAB. If you do not set this mask, and if the cell already contains a record, the PUT operation returns an error completion and does not transfer the record.

STREAM CONTEXT

The current-record context after a key access PUT operation is undefined; the next-record context is unchanged.

RETURNED VALUES

RRN

For a relative file or for a sequential disk file with fixed-length records, a key-access PUT operation returns the relative record number (RRN) for the inserted record in the 2-word BKT field of the RAB.

RFA

The PUT operation returns the record file address (RFA) for the inserted record in the 3-word RFA field of the RAB.

Completion Status and Value

The PUT operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-45 lists control block fields that are input to the PUT operation. Table 5-46 lists control block fields that are output by the PUT operation.

Table 5-45: PUT (Key Access) Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	KBF	Key buffer address
RAB	KSZ	Key size (bytes)
RAB	RAC	Record access code
		RB$KEY Key access
RAB	RBF	Record buffer address
RAB	RHB	VFC control buffer address
RAB	ROP	Record processing option mask
		RB$LOA Honor bucket fill numbers
		RB$LOC Locate mode
		RB$UIF Update if record exists
RAB	RSZ	Record size (bytes)
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-46: PUT (Key Access) Output Fields

Block	Field	Description
RAB	BKT	Relative record number (RRN)
RAB	RBF	Record buffer address
RAB	RFA	Record file address
RAB	STS	Completion status code
RAB	STV	Completion status value

5.22 $READ MACRO (SEQUENTIAL ACCESS)

The $READ macro calls the READ operation routine to transfer blocks from a file to an I/O buffer. The target of a sequential-access READ operation is the readable block (and, for a multiblock READ operation, following blocks).

FORMAT

The format for the $READ macro is:

```
$READ rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the READ operation.

OPTIONS

Internal Stream Identifier

The READ operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Block Specification

For a sequential-access READ operation, specify 0 in the 2-word BKT field of the RAB.

User Buffer

Specify the address of the user buffer in the 1-word UBF field of the RAB, and specify the size (in bytes) of the user buffer in the 1-word USZ field of the RAB.

The READ operation reads enough blocks from the file to fill the buffer (unless it reached the end-of-file before the buffer is fille d).

STREAM CONTEXT

The readable-block context after a READ operation is the block following the last-read block; the writable-block context is the first-read block.

RETURNED VALUES

Data Blocks

The READ operation returns the address and length of the data read from the file. The value in the 1-word RBF field of the RAB is the address of the data read; the value in the 1-word RSZ field of the RAB is the length (in bytes) of the data read.

The READ operation normally will not read beyond the logical end-of-file. For sequential files with undefined (UDF) record format, however, the READ operation will respect the logical end-of-file marker only in either of the following situations:

- The file is not accessed for writing and no write-sharing is allowed.
- The file is accessed for writing and the FB$NIL mask in the 1-byte SHR field of the FAB is set.

If you do not specify no write-sharing, RMS-11 will ignore the logical end-of-file marker and will stop only at the physical end-of-file on the disk.

Record File Address (RFA)

The READ operation returns the virtual block number of the first-read block in the first two words of the 3-word RFA field of the RAB (it clears the third word).

Completion Status and Value

The READ operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-47 lists control block fields that are input to the READ operation. Table 5-48 lists control block fields that are output by the READ operation.

Table 5-47: READ (Sequential Access) Input Fields

Block	Field	Description
RAB	BKT	Virtual block number (VBN)
RAB	ISI	Internal stream identifier
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-48: READ (Sequential Access) Output Fields

Block	Field	Description
RAB	RBF	Record buffer address
RAB	RFA	Virtual block number (2 words)
RAB	RSZ	Record size (bytes)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.23 $READ MACRO (VBN ACCESS)

The $READ macro calls the READ operation routine to transfer blocks from a file to an I/O buffer. The target of a VBN-access READ operation is a specified block (and, for a multiblock READ operation, following blocks).

FORMAT

The format for the $READ macro is:

```
$READ rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the READ operation.

OPTIONS

Internal Stream Identifier

The READ operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Block Specification

Specify the virtual block number of the first block to be read in the 2-word BKT field of the RAB.

User Buffer

Specify the address of the user buffer in the 1-word UBF field of the RAB, and specify the size (in bytes) of the user buffer in the 1-word USZ field of the RAB.

The READ operation reads enough blocks from the file to fill the buffer (unless it reached the end-of-file before the buffer is fille d).

STREAM CONTEXT

The readable-block context after a READ operation is the block following the last-read block; the writable-block context is the first-read block.

RETURNED VALUES

Data Blocks

The READ operation returns the address and length of the data read from the file. The value in the 1-word RBF field of the RAB is the address of the data read; the value in the 1-word RSZ field of the RAB is the length (in bytes) of the data read.

The READ operation normally will not read beyond the logical end-of-file. For sequential files with undefined (UDF) record format, however, the READ operation will respect the logical end-of-file marker only in either of the following situations:

- The file is not accessed for writing and no write-sharing is allowed.
- The file is accessed for writing and the FB$NIL mask in the 1-byte SHR field of the FAB is set.

If you do not specify no write-sharing, RMS-11 will ignore the logical end-of-file marker and will stop only at the physical end-of-file on the disk.

Record File Address (RFA)

The READ operation returns the virtual block number of the first-read block in the first two words of the 3-word RFA field of the RAB (it clears the third word).

Completion Status and Value

The READ operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-49 lists control block fields that are input to the READ operation. Table 5-50 lists control block fields that are output by the READ operation.

Table 5-49: READ (VBN Access) Input Fields

Block	Field	Description
RAB	BKT	Virtual block number (VBN)
RAB	ISI	Internal stream identifier
RAB	UBF	User buffer address
RAB	USZ	User buffer size (bytes)

Table 5-50: READ (VBN Access) Output Fields

Block	Field	Description
RAB	RBF	Record buffer address
RAB	RFA	Virtual block number (2 words)
RAB	RSZ	Record size (bytes)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.24 $REMOVE MACRO

The $REMOVE macro calls the REMOVE operation routine to remove the directory entry for a file.

FORMAT

The format for the $REMOVE macro is:

```
$REMOVE fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the REMOVE operation.

If you supply a NAM block and specify remove by NAM block, the REMOVE operation reads NAM block fields to obtain identifiers for the target file.

To supply a NAM block for the REMOVE operation, specify the address of the NAM block in the l-word NAM field of the FAB.

OPTIONS

Remove by File Specification

If you want to remove a directory entry by its file specification, clear the FB$FID mask in the l-word FOP field of the FAB.

If the FB$FID mask is clear, the REMOVE operation uses the fully qualified file specification to identify the target file.

Specify the address of the file string in the l-word FNA field of the FAB, and specify the size (in bytes) of the file string in the l-byte FNS field of the FAB. If you specify 0 in the FNS field, the REMOVE operation uses no file string.

Specify the address of the default string in the l-word DNA field of the FAB, and specify the size (in bytes) of the default string in the l-byte DNS field of the FAB. If you specify 0 in the DNS field, the REMOVE operation uses no default string.

Remove by NAM Block

If you want to identify the target file by its identifiers, set the FB$FID mask in the l-word FOP field of the FAB, and supply a filled-in NAM block (one whose fields were written by a CREATE, OPEN, or SEARCH operation).

The REMOVE operation reads the 2-word DVI field of the NAM block and the 3-word DID field of the NAM block (if nonzero) to override the corresponding file specification elements in the merged string.

Remove by Wildcard Specification

You can use the REMOVE operation in a wildcarding program loop. (The NB$WCH mask in the 1-word FNB field of the NAM block will already have been set by an earlier PARSE operation.)

If you set the FB$FID mask in the 1-word FOP field of the FAB, the file found by a previous SEARCH operation is removed without affecting fields that are used as context for subsequent SEARCH operations.

If you clear the FB$FID mask in the 1-word FOP field of the FAB, the REMOVE operation first performs an implicit SEARCH operation. (The input and output fields for the SEARCH operation are not described here and are not included in the checklists at the end of this section.)

If the SEARCH operation finds a file that matches the wildcard file specification, the REMOVE operation removes its directory entry; if not, the REMOVE operation does not remove a directory entry, but instead passes control block data from the SEARCH operation (in particular, the ER$NMF completion status code and the cleared NB$WCH mask in the 1-word FNB field of the NAM block).

Expanded String Buffer

If you want the REMOVE operation to return the expanded string for the file whose directory entry was removed, provide a buffer for the string. Specify the address of the expanded string buffer in the 1-word ESA field of the NAM block. Specify the size (in bytes) of the expanded string buffer in the 1-byte ESS field of the NAM block; if you specify 0 in the ESS field, the REMOVE operation does not return the expanded string.

Private Buffer Pool

If you want the REMOVE operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the REMOVE operation uses the central buffer pool.

Logical Channel

Specify the logical channel for the REMOVE operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

RETURNED VALUES

Device, Directory, and File Identifiers

If you supply a NAM block, the REMOVE operation writes a device identifier in the 2-word DVI field of the NAM block, a directory identifier in the 3-word DID field of the NAM block, and a file identifier in the 3-word FID field of the NAM block.

Expanded String

If you specify a buffer for the expanded string for the file (ESA and ESS fields in the NAM block), the REMOVE operation writes the expanded string for the target file in this buffer, and writes the length (in bytes) of the string in the 1-byte ESL field of the NAM block.

File Specification Characteristics

The REMOVE operation sets the masks in the 1-word FNB field of the NAM block to show which file specification elements were present in the file string and default string.

These masks and their meaning are:

NB$NOD	Node in file string or default string
NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$QUO	Quoted string in file string or default string
NB$NAM	File name in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Wildcarding

The REMOVE operation clears the NB$WCH mask in the 1-word FNB field of the NAM block; this shows that no wildcard context exists after the REMOVE operation. It also clears the 1-byte RSL field of the NAM block to show that no resultant string was returned.

Completion Status and Value

The REMOVE operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-53 lists control block fields that are input to the REMOVE operation. Table 5-54 lists control block fields that are output by the REMOVE operation.

Table 5-53: REMOVE Input Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	FOP	File processing option mask
		FB$FID Use information in NAM block
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
NAM	DVI	Device identifier
NAM	ESA	Expanded string buffer address
NAM	ESS	Expanded string buffer size (bytes)
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$WCH Wildcard context established

Table 5-54: REMOVE Output Fields

Block	Field	Description
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESL	Expanded string length (bytes)
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
		NB$WCH Wildcard context established (cleared)
NAM	RSL	Resultant string length (bytes)

5.25 $RENAME MACRO

The $RENAME macro calls the RENAME operation routine to change the directory entry for a file.

The old and new entries (file specifications) must have the same device specification.

FORMAT

The format for the $RENAME macro is:

```
$RENAME oldfabaddr,[erraddr],[sucaddr],newfabaddr
```

where **oldfabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; **sucaddr** is the address of the success handler for the operation; and **newfabaddr** is the address of the FAB giving the new file specification.

CONTROL BLOCKS

You must supply two FABs for the RENAME operation: an "old" FAB containing the current specification for the file, and a "new" FAB containing the new specification for the file.

If you supply a NAM block for the old FAB (old NAM block) and specify either rename by NAM block or wildcarding, the RENAME operation reads its fields to obtain identifiers for the old file specification. If you supply a NAM block for the new FAB (new NAM block) and specify rename by NAM block, the RENAME operation reads its fields to obtain identifiers for the new file specification.

To supply a NAM block for the RENAME operation, specify the address of the NAM block in the 1-word NAM field of the FAB.

OPTIONS

Old File Specification (Nonwildcard RENAME Operation)

If the NB$WCH mask in the 1-word FNB field of the NAM block is cleared (nonwildcard RENAME operation), the RENAME operation uses the merged string to identify the target file.

Specify the old file string in the old FAB: specify the address of the old file string in the 1-word FNA field of the FAB, and specify the size (in bytes) of the old file string in the 1-byte FNS field of the FAB. If you specify 0 in the FNS field, the RENAME operation uses no old file string.

Specify the old default string in the old FAB: specify the address of the old default string in the 1-word DNA field of the FAB, and specify the size (in bytes) of the old default string in the 1-byte DNS field of the FAB. If you specify 0 in the DNS field, the RENAME operation uses no old default string.

If you set the FB$FID mask in the 1-word FOP field of the FAB and supply a NAM block, the RENAME operation reads the device identifier from the 2-word DVI field of the NAM block; if this value is nonzero, the specified device overrides the device in the merged string.

In the same circumstance, the RENAME operation reads the directory identifier from the 3-word DID field of the NAM block; if this value is nonzero, the specified directory overrides the directory in the merged string.

Old File Specification (Wildcard RENAME Operation)

You can use the RENAME operation in a wildcarding program loop. (The NB$WCH mask in the 1-word FNB field of the NAM block will already have been set by an earlier PARSE operation.)

If you set the FB$FID mask in the 1-word FOP field of the FAB, the file found by a previous SEARCH operation is renamed without affecting fields that are used as context for subsequent SEARCH operations.

If you clear the FB$FID mask in the 1-word FOP field of the FAB, the RENAME operation first performs an implicit SEARCH operation. (The input and output fields for the SEARCH operation are not described here and are not included in the checklists at the end of this section.)

If the SEARCH operation finds a file that matches the wildcard file specification, the RENAME operation replaces its directory entry; if not, the RENAME operation does not replace a directory entry, but instead passes control block data from the SEARCH operation (in particular, the ER$NMF completion status code and the cleared NB$WCH mask in the 1-word FNB field of the NAM block).

New File Specification

The RENAME operation uses the new merged string to identify the target file.

Specify the address of the new file string in the 1-word FNA field of the FAB and specify the size (in bytes) of the new file string in the 1-byte FNS field of the FAB. If you specify 0 in the FNS field, the RENAME operation uses no new file string.

Specify the address of the new default string in the 1-word DNA field of the FAB and specify the size (in bytes) of the new default string in the 1-byte DNS field of the FAB. If you specify 0 in the DNS field, the RENAME operation uses no new default string.

If you set the FB$FID mask in the 1-word FOP field of the FAB and supply a NAM block, the RENAME operation reads the device identifier from the 2-word DVI field of the NAM block; if this value is nonzero, the specified device overrides the device in the merged string.

In the same circumstance, the RENAME operation reads the directory identifier from the 3-word DID field of the NAM block; if this value is nonzero, the specified directory overrides the directory in the merged string.

Private Buffer Pool

If you want the RENAME operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the RENAME operation uses the central buffer pool.

Logical Channel

Specify the logical channel for the RENAME operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

Expanded String Buffers

If you want the expanded string for the file given by a FAB returned to a buffer, supply a NAM block for the FAB. Specify the address of the buffer in the 1-word ESA field of the NAM block, and the size (in bytes) of the buffer in the 1-byte ESS field of the NAM block. If you do not supply a NAM block for a FAB, or if you specify 0 in the ESS field, the RENAME operation does not return the expanded string.

RETURNED VALUES

Expanded Strings

If you specify a buffer for the expanded string for a FAB (ESA and ESS fields in the NAM block), the RENAME operation writes the expanded string in the buffer, and writes the length (in bytes) of the string in the 1-byte ESL field of the NAM block.

Device, Directory, and File Identifiers

If you supply a NAM block, the RENAME operation writes a device identifier in the 2-word DVI field of the NAM block, a directory identifier in the 3-word DID field of the NAM block, and a file identifier in the 3-word FID field of the NAM block.

File Specification Characteristics

The RENAME operation sets masks in the 1-word FNB field of the NAM block to show which file specification elements were present in the file string and default string.

These masks and their meaning are:

NB$NOD	Node in file string or default string
NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$QUO	Quoted string in file string or default string
NB$NAM	File name in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

Wildcarding

The RENAME operation clears the NB$WCH mask in the 1-word FNB field of the NAM block; this shows that no wildcard context exists after the RENAME operation. It also clears the 1-byte RSL field of the NAM block to show that no resultant string was returned.

Completion Status and Value

The RENAME operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-55 lists control block fields that are input to the RENAME operation. Table 5-56 lists control block fields that are output by the RENAME operation.

Table 5-55: RENAME Input Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	DNA	Default string address
FAB	DNS	Default string size (bytes)
FAB	FNA	File string address
FAB	FNS	File string size (bytes)
FAB	FOP	File processing option mask
		FB$FID Use information in NAM block
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESA	Expanded string buffer address
NAM	ESS	Expanded string buffer size (bytes)
NAM	FNB	File specification mask
		NB$WCH Wildcard context established

Table 5-56: RENAME Output Fields

Block	Field	Description
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESL	Expanded string length (bytes)
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$NOD Node in file string or default string
		NB$DEV Device in file string or default string
		NB$DIR Directory in file string or default string
		NB$QUO Quoted string in file string or default string
		NB$NAM File name in file string or default string
		NB$TYP File type in file string or default string
		NB$VER File version in file string or default string
		NB$WDI Wildcard directory in file string or default string
		NB$WNA Wildcard file name in file string or default string
		NB$WTY Wildcard file type in file string or default string
		NB$WVE Wildcard file version in file string or default string
		NB$WCH Wildcard context established (cleared)
NAM	RSL	Resultant string length (bytes)

5.26 $REWIND MACRO

The $REWIND macro calls the REWIND operation routine to reset the context for a stream to the beginning-of-file. The file can have any organization.

FORMAT

The format for the $REWIND macro is:

```
$REWIND rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the REWIND operation.

OPTIONS

Internal Stream Identifier

The REWIND operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Key of Reference

For an indexed file, you must specify the index that the stream will use in accessing records. Specify this key of reference in the 1-byte KRF field of the RAB. This value matches the value in the file's KEY block for the index: 0 for the primary index, 1 for the first alternate index, and so forth.

STREAM CONTEXT

For a record access file, the current context after a REWIND operation is undefined and the next-record context is the first record in the file; for an indexed file, this first record is defined by the specified index.

For a block access file, both the readable-block and writable-block contexts after a REWIND operation are the first block in the file.

RETURNED VALUES

Completion Status and Value

The REWIND operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-57 lists control block fields that are input to the REWIND operation. Table 5-58 lists control block fields that are output by the REWIND operation.

Table 5-57: REWIND Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	KRF	Key of reference

Table 5-58: REWIND Output Fields

Block	Field	Description
RAB	STS	Completion status code
RAB	STV	Completion status value

5.27 $SEARCH MACRO

The $SEARCH macro calls the SEARCH operation routine to scan a directory and return a file specification and identifiers in NAM block fields. You should precede the SEARCH operation by a PARSE operation, which initializes the NAM block fields for the SEARCH operation.

The SEARCH operation finds a file specification that matches the match-pattern initialized (in the expanded string buffer) by the PARSE operation; a series of wildcard SEARCH operations returns successive matching file specifications.

FORMAT

The format for the $SEARCH macro is:

```
$SEARCH fabaddr[,erraddr[,sucaddr]]
```

where **fabaddr** is the address of the FAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a FAB for the SEARCH operation.

You must supply a NAM block for the SEARCH operation.

To supply a NAM block for the SEARCH operation, specify the address of the NAM block in the 1-word NAM field of the FAB.

OPTIONS

Wildcard Context Information

The SEARCH operation reads NAM block fields that are initialized, written, or preserved by a preceding PARSE or wildcard SEARCH operation: the 3-word DID field of the NAM block, the 2-word DVI field of the NAM block, the 1-word ESA field of the NAM block, the 1-byte ESL field of the NAM block, the NB$WCH mask in the 1-word FNB field of the NAM block, the 1-word RSA field of the NAM block, the 1-byte RSL field of the NAM block, the 1-byte RSS field of the NAM block, the 1-word WCC field of the NAM block, and the 1-word WDI field of the NAM block.

The SEARCH operation also uses the expanded string in the expanded string buffer.

You must preserve these fields between a PARSE and a SEARCH operation and between successive wildcard SEARCH operations.

Private Buffer Pool

If you want the SEARCH operation to use a private buffer pool instead of the central buffer pool, specify the address of the (word-aligned) private buffer pool in the 1-word BPA field of the FAB, and its size (in bytes) in the 1-word BPS field of the FAB; this size must be a multiple of 4.

If you specify 0 in either the BPA field or the BPS field, the SEARCH operation uses the central buffer pool.

Logical Channel

Specify the logical channel for the SEARCH operation in the 1-byte LCH field of the FAB. The logical channel number must not be the same as the logical channel number for any already-open file, and must not be 0.

RETURNED VALUES

Resultant String

The SEARCH operation writes the full file specification for the found file in the resultant string buffer, and writes the length of the string in the 1-byte RSL field of the NAM block.

Directory and File Identifiers

If the SEARCH operation finds a file that matches the wildcard pattern, it writes the directory identifier for the found file in the 3-word DID field of the NAM block, and the file identifier in the 3-word FID field of the NAM block.

Wildcard Context Information

The SEARCH operation writes the wildcard context in the 1-word WCC field of the NAM block, and the wildcard directory context in the 1-word WDI field of the NAM block.

If the SEARCH operation did not find a matching file, it clears the NB$WCH mask in the 1-word FNB field of the NAM block; this shows that no further wildcarding is possible using the current wildcard information.

Completion Status and Value

The SEARCH operation returns completion status in the 1-word STS field of the FAB and returns a completion value in the 1-word STV field of the FAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-59 lists control block fields that are input to the SEARCH operation. Table 5-60 lists control block fields that are output by the SEARCH operation.

Table 5-59: SEARCH Input Fields

Block	Field	Description
FAB	BPA	Private buffer pool address
FAB	BPS	Private buffer pool size (bytes)
FAB	LCH	Logical channel number
FAB	NAM	NAM block address
NAM	DID	Directory identifier
NAM	DVI	Device identifier
NAM	ESA	Expanded string buffer address
NAM	ESL	Expanded string length (bytes)
NAM	FNB	File specification mask
		NB$WCH Wildcard context established
NAM	RSA	Resultant string buffer address
NAM	RSL	Resultant string length (bytes)
NAM	RSS	Resultant string buffer size (bytes)
NAM	WCC	Wildcard context
NAM	WDI	Wildcard directory context

Table 5-60: SEARCH Output Fields

Block	Field	Description
FAB	STS	Completion status code
FAB	STV	Completion status value
NAM	DID	Directory identifier
NAM	FID	File identifier
NAM	FNB	File specification mask
		NB$WCH Wildcard context established
NAM	RSL	Resultant string length (bytes)
NAM	WCC	Wildcard context
NAM	WDI	Wildcard directory context

5.28 $TRUNCATE MACRO

The $TRUNCATE macro calls the TRUNCATE operation routine to remove records from the latter part of a sequential file; records are removed inclusively from the current record through the end-of-file. If the file cannot be truncated, the TRUNCATE operation returns an error completion and leaves the current-record context undefined and the next-record context unchanged.

FORMAT

The format for the $TRUNCATE macro is:

```
$TRUNCATE rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the TRUNCATE operation.

OPTIONS

Internal Stream Identifier

The TRUNCATE operation reads the internal stream identifier from the 1-word ISI field of the RAB.

STREAM CONTEXT

The TRUNCATE operation destroys the current-record context; the next-record context after the TRUNCATE operation is the end-of-file.

RETURNED VALUES

Completion Status and Value

The TRUNCATE operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-63 lists control block fields that are input to the TRUNCATE operation. Table 5-64 lists control block fields that are output by the TRUNCATE operation.

Table 5-63: TRUNCATE Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier

Table 5-64: TRUNCATE Output Fields

Block	Field	Description
RAB	STS	Completion status code
RAB	STV	Completion status value

5.29 $UPDATE MACRO

The $UPDATE macro calls the UPDATE operation routine to transfer a record from a user buffer to a file (overwriting the existing record). The target of the UPDATE operation is the current record, which is overwritten.

If no record (as specified in the RAB) can be transferred, the UPDATE operation returns an error completion.

FORMAT

The format for the $UPDATE macro is:

```
$UPDATE rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the UPDATE operation.

OPTIONS

Internal Stream Identifier

The UPDATE operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Record Buffer

Specify the address of the record buffer in the 1-word RBF field of the RAB, and specify the size (in bytes) of the record buffer in the 1-word RSZ field of the RAB. For sequential files and for indexed files in which duplicate primary key values are permitted, the size of the buffer must be the same as the size of the existing record.

If the file has VFC format, specify the address of the buffer for the VFC header in the 1-word RHB field of the RAB; if you specify zero in this field, the existing record header will remain unchanged.

STREAM CONTEXT

The UPDATE operation destroys the current-record context; the next-record context after the UPDATE operation is unchanged.

RETURNED VALUES

Completion Status and Value

The UPDATE operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-65 lists control block fields that are input to the UPDATE operation. Table 5-66 lists control block fields that are output by the UPDATE operation.

Table 5-65: UPDATE Input Fields

Block	Field	Description
RAB	ISI	Internal stream identifier
RAB	RBF	Record buffer address
RAB	RHB	VFC control buffer address
RAB	RSZ	Record size (bytes)

Table 5-66: UPDATE Output Fields

Block	Field	Description
RAB	STS	Completion status code
RAB	STV	Completion status value

5.30 $WRITE MACRO (SEQUENTIAL ACCESS)

The $WRITE macro calls the WRITE operation routine to write blocks to a file. The target of a sequential-access WRITE operation is the writable block (and, for a multiblock WRITE operation, following blocks).

FORMAT

The format for the $WRITE macro is:

```
$WRITE rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the WRITE operation.

OPTIONS

Internal Stream Identifier

The WRITE operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Block Identification

For a sequential-access WRITE operation, specify 0 in the 2-word BKT field of the RAB.

Record Buffer

Specify the address of the record buffer in the 1-word RBF field of the RAB, and specify the size (in bytes) of the record buffer in the 1-word RSZ field of the RAB. You must specify a record buffer for the WRITE operation; the WRITE operation transfers data from this buffer to the file.

The WRITE operation normally updates the logical end-of-file marker, when appropriate, and automatically extends the file's allocation, when necessary. For sequential files with undefined (UDF) record format, however, the WRITE operation updates the logical end-of-file marker and performs automatic file extensions only if the FB$NIL mask in the 1-byte SHR field of the FAB is set.

Record File Address (RFA)

The WRITE operation returns the virtual block number of the first-written block in the first two words of the 3-word RFA field of the RAB (it clears the third word).

STREAM CONTEXT

The readable-block context after a WRITE operation is the block following the last-written block; the writable-block context after a WRITE operation is the block following the last-written block.

RETURNED VALUES

Completion Status and Value

The WRITE operation returns completion status in the l-word STS field of the RAB and returns a completion value in the l-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-68 lists control block fields that are input to the WRITE operation. Table 5-69 lists control block fields that are output by the WRITE operation.

Table 5-68: WRITE (Sequential Access) Input Fields

Block	Field	Description
RAB	BKT	Virtual block number (VBN)
RAB	ISI	Internal stream identifier
RAB	RBF	Record buffer address
RAB	RSZ	Record size (bytes)

Table 5-69: WRITE (Sequential Access) Output Fields

Block	Field	Description
RAB	RFA	Virtual block number (2 words)
RAB	STS	Completion status code
RAB	STV	Completion status value

5.31 $WRITE MACRO (VBN ACCESS)

The $WRITE macro calls the WRITE operation routine to write blocks to a file. The target of a VBN-access WRITE operation is the writable block (and, for a multiblock WRITE operation, following blocks).

FORMAT

The format for the $WRITE macro is:

```
$WRITE rabaddr[,erraddr[,sucaddr]]
```

where **rabaddr** is the address of the RAB for the operation; **erraddr** is the address of the error handler for the operation; and **sucaddr** is the address of the success handler for the operation.

CONTROL BLOCKS

You must supply a RAB for the WRITE operation.

OPTIONS

Internal Stream Identifier

The WRITE operation reads the internal stream identifier from the 1-word ISI field of the RAB.

Block Identification

Specify the virtual block number of the first block to be written in the 2-word BKT field of the RAB.

Record Buffer

Specify the address of the record buffer in the 1-word RBF field of the RAB, and specify the size (in bytes) of the record buffer in the 1-word RSZ field of the RAB. You must specify a record buffer for the WRITE operation; the WRITE operation transfers data from this buffer to the file.

The WRITE operation normally updates the logical end-of-file marker, when appropriate, and automatically extends the file's allocation, when necessary. For sequential files with undefined (UDF) record format, however, the WRITE operation updates the logical end-of-file marker and performs automatic file extensions only if the FB$NIL mask in the 1-byte SHR field of the FAB is set.

Record File Address (RFA)

The WRITE operation returns the virtual block number of the first-written block in the first two words of the 3-word RFA field of the RAB (it clears the third word).

STREAM CONTEXT

The readable-block context after a WRITE operation is the block following the last-written block; the writable-block context after a WRITE operation is the block following the last-written block.

RETURNED VALUES

Completion Status and Value

The WRITE operation returns completion status in the 1-word STS field of the RAB and returns a completion value in the 1-word STV field of the RAB. Appendix A lists completion status symbols and values.

CHECKLISTS

Table 5-70 lists control block fields that are input to the WRITE operation. Table 5-71 lists control block fields that are output by the WRITE operation.

Table 5-70: WRITE (VBN Access) Input Fields

Block	Field	Description
RAB	BKT	Virtual block number (VBN)
RAB	ISI	Internal stream identifier
RAB	RBF	Record buffer address
RAB	RSZ	Record size (bytes)

Table 5-71: WRITE (VBN Access) Output Fields

Block	Field	Description
RAB	RFA	Virtual block number (2 words)
RAB	STS	Completion status code
RAB	STV	Completion status value

CHAPTER 6

CONTROL BLOCK FIELDS

Each major section of this chapter describes an RMS-11 control block, and includes:

- Block summary table

  A table summarizes the entire control block. The table shows the offset, offset symbol, field size, and a brief description of each field in the block; for each mask or code for a field, the table shows the value, symbol, and a brief description of the mask or code.

- Field summaries

  Each subsection following the block summary table is a description of one field in the block. A field that has masks that are very different in purpose (such as the FOP field in the FAB) is described as a number of separate "fields" (such as FOP FB$FID, FOP FB$RWO, and so forth).

  The description of each field includes the following:

  USE: a summary of the purpose of the field

  SIZE: the size of the field

  INIT: the format of the field-initialization macro (if any)

  ACCESS: the formats of field-access macros to access the field

  MASKS or **CODES:** (if any) each mask or code symbol and a brief description

  INPUT: the operations that read values from the field, and the meanings of those values

  OUTPUT: the operations that store values in the field, and the meanings of those values

Fields described as "Reserved" and undefined bits in masks should (and in some cases must) be 0.

6.1 ALL BLOCK SUMMARY

This section summarizes the ALL block and its fields. Table 6-1 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-1: ALL Block Summary

Offset	Offset Symbol	Field Size	Description
000	O$COD	1 byte	ALL block identifier code
			000004 XB$ALL ALL block identifier
001	O$BLN	1 byte	ALL block length (bytes)
			000034 XB$LAL ALL block length (bytes)
002	O$NXT	1 word	Next XAB address
004	O$AID	1 byte	Area number
005	O$BKZ	1 byte	Area bucket size (blocks)
006	O$VOL	1 word	Reserved
010	O$ALN	1 byte	Area alignment mask
			000001 XB$CYL Cylinder alignment 000002 XB$LBN Logical block alignment 000004 XB$VBN Virtual block alignment
011	O$AOP	1 byte	Area option mask
			000001 XB$HRD Hard area location 000002 XB$CTG Contiguous area
012	O$ALQ	2 words	Area allocation size (blocks)
012	O$ALQ0	1 word	ALQ field low word
014	O$ALQ1	1 word	ALQ field high word
016	O$DEQ	1 word	Area default extension size (blocks)
020		1 word	Reserved
022	O$LOC	2 words	Area location
022	O$LOC0	1 word	LOC field low word
024	O$LOC1	1 word	LOC field high word

6.1.1 AID Field in ALL Block

USE Contains the area identifier for the area described by the ALL block.

INIT X$AID number

SIZE 1 byte

ACCESS

```
$FETCH dst,AID,reg      ;AID field to 1-byte dst
$STORE src,AID,reg      ;1-byte src to AID field
$COMPARE src,AID,reg    ;1-byte src with AID field
```

INPUT

CLOSE	Area number
CREATE	Area number
DISPLAY	Area number
EXTEND	Area number
OPEN	Area number

6.1.2 ALN Field in ALL Block

USE Indicates alignment for the area described by the ALL block.

INIT X$ALN mask

SIZE 1 byte

ACCESS

```
$SET mask,ALN,reg       ;Mask bits on in ALN field
$OFF mask,ALN,reg       ;Mask bits off in ALN field
$TESTBITS mask,ALN,reg  ;Test mask bits in ALN field
$FETCH dst,ALN,reg      ;ALN field to 1-byte dst
$STORE src,ALN,reg      ;1-byte src to ALN field
$COMPARE src,ALN,reg    ;1-byte src with ALN field
```

MASKS

```
XB$CYL  Cylinder alignment
XB$LBN  Logical block alignment
XB$VBN  Virtual block alignment
```

INPUT

```
CREATE      Initial area alignment request
EXTEND      Area extension alignment request
```

OUTPUT

```
DISPLAY     Area alignment mask (cleared)
OPEN        Area alignment mask (cleared)
```

6.1.3 ALQ Field in ALL Block

USE Contains the allocation size for the area described by the ALL block.

INIT X$ALQ number

SIZE 2 words

ACCESS

```
$FETCH dst,ALQ,reg      ;ALQ field to 2-word dst
$STORE src,ALQ,reg      ;2-word src to ALQ field
$FETCH dst,ALQn,reg     ;ALQ word n to 1-word dst
$STORE src,ALQn,reg     ;1-word src to ALQ word n
$COMPARE src,ALQn,reg   ;1-word src with ALQ word n
```

INPUT

CREATE	Initial area allocation request size (blocks)
EXTEND	Area allocation extension request size (blocks)

OUTPUT

DISPLAY	Unused area allocation size (blocks)
EXTEND	Area allocation extension actual size (blocks)
OPEN	Unused area allocation size (blocks)

6.1.4 AOP Field in ALL Block (XB$CTG Mask)

USE Indicates contiguity for the area described by the ALL block.

INIT X$AOP mask

SIZE 1 byte

ACCESS

```
$SET mask,AOP,reg        ;Mask bits on in AOP field
$OFF mask,AOP,reg        ;Mask bits off in AOP field
$TESTBITS mask,AOP,reg   ;Test mask bits in AOP field
$FETCH dst,AOP,reg       ;AOP field to 1-byte dst
$STORE src,AOP,reg       ;1-byte src to AOP field
$COMPARE src,AOP,reg     ;1-byte src with AOP field
```

INPUT

CREATE	Contiguous area request
EXTEND	Contiguous area extension request

OUTPUT

DISPLAY	Contiguous area (cleared)
OPEN	Contiguous area (cleared)

6.1.5 AOP Field in ALL Block (XB$HRD Mask)

USE Indicates a demand for the requested location.

INIT X$AOP mask

SIZE 1 byte

ACCESS

```
$SET mask,AOP,reg       ;Mask bits on in AOP field
$OFF mask,AOP,reg       ;Mask bits off in AOP field
$TESTBITS mask,AOP,reg  ;Test mask bits in AOP field
$FETCH dst,AOP,reg      ;AOP field to 1-byte dst
$STORE src,AOP,reg      ;1-byte src to AOP field
$COMPARE src,AOP,reg    ;1-byte src with AOP field
```

INPUT

CREATE	Area hard location request
EXTEND	Area extension hard location request

OUTPUT

DISPLAY	Hard area location (cleared)
OPEN	Hard area location (cleared)

6.1.6 BKZ Field in ALL Block

USE Contains the bucket size for the area described by the ALL block.

INIT X$BKZ number

SIZE 1 byte

ACCESS

```
$FETCH dst,BKZ,reg      ;BKZ field to 1-byte dst
$STORE src,BKZ,reg      ;1-byte src to BKZ field
$COMPARE src,BKZ,reg    ;1-byte src with BKZ field
```

INPUT

CREATE	Area bucket size (blocks)

OUTPUT

DISPLAY	Area bucket size (blocks)
OPEN	Area bucket size (blocks)

6.1.7 BLN Field in ALL Block (XB$LAL Code)

USE Contains the length of the ALL block.

INIT None

SIZE 1 byte

ACCESS

```
$FETCH dst,BLN,reg      ;BLN field to 1-byte dst
$COMPARE src,BLN,reg    ;1-byte src with BLN field
```

6.1.8 COD Field in ALL Block (XB$ALL Code)

USE	Contains the identifier for the ALL block.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,COD,reg ;COD field to 1-byte dst $COMPARE src,COD,reg ;1-byte src with COD field

6.1.9 DEQ Field in ALL Block

USE Contains the default extension size for the area described by the ALL block.

INIT X$DEQ number

SIZE 1 word

ACCESS

```
$FETCH dst,DEQ,reg      ;DEQ field to 1-word dst
$STORE src,DEQ,reg      ;1-word src to DEQ field
$COMPARE src,DEQ,reg    ;1-word src with DEQ field
```

INPUT

CREATE	Area default extension size (blocks)

OUTPUT

DISPLAY	Area default extension size (blocks)
OPEN	Area default extension size (blocks)

6.1.10 LOC Field in ALL Block

USE Contains the location of the area described by the ALL block.

The meaning of the location depends on the ALN mask: a cylinder number (if XB$CYL), a logical block number (if XB$LBN), or a virtual block number (if XB$VBN).

INIT X$LOC number

SIZE 2 words

ACCESS

```
$FETCH dst,LOC,reg      ;LOC field to 2-word dst
$STORE src,LOC,reg      ;2-word src to LOC field
$FETCH dst,LOCn,reg     ;LOC word n to 1-word dst
$STORE src,LOCn,reg     ;1-word src to LOC word n
$COMPARE src,LOCn,reg   ;1-word src with LOC word n
```

INPUT

CREATE	Initial area location request
EXTEND	Area extension location request

6.1.11 NXT Field in ALL Block

USE Contains the address of the next XAB (ALL, DAT, KEY, PRO, or SUM block) in a chain of XABs.

INIT X$NXT address

SIZE 1 word

ACCESS

```
$FETCH dst,NXT,reg      ;NXT field to 1-word dst
$STORE src,NXT,reg      ;1-word src to NXT field
$COMPARE src,NXT,reg    ;1-word src with NXT field
```

INPUT

CLOSE	Next XAB address
CREATE	Next XAB address
DISPLAY	Next XAB address
ENTER	Next XAB address
ERASE	Next XAB address
EXTEND	Next XAB address
OPEN	Next XAB address
PARSE	Next XAB address
REMOVE	Next XAB address
RENAME	Next XAB address
SEARCH	Next XAB address

6.2 DAT BLOCK SUMMARY

This section summarizes the DAT block and its fields. Table 6-2 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-2: DAT Block Summary

Offset	Offset Symbol	Field Size	Description
000	O$COD	1 byte	DAT block identifier code
			000003 XB$DAT DAT block identifier
001	O$BLN	1 byte	DAT block length (bytes)
			000046 XB$DTL DAT block length (bytes)
002	O$NXT	1 word	Next XAB address
006	O$RDT	4 words	File revision date
016	O$CDT	4 words	File creation date
036	O$BDT	4 words	Reserved

6.2.1 BLN Field in DAT Block (XB$DTL Code)

USE	Contains the length of the DAT block.	
INIT	None	
SIZE	1 byte	
ACCESS	$FETCH dst,BLN,reg	;BLN field to 1-byte dst
	$COMPARE src,BLN,reg	;1-byte src with BLN field

6.2.2 CDT Field in DAT Block

USE	Contains the binary creation date for the file.
INIT	None
SIZE	4 words
ACCESS	$FETCH dst,CDT,reg ;CDT field to 4-word dst
OUTPUT	DISPLAY File creation date OPEN File creation date

6.2.3 COD Field in DAT Block (XB$DAT Code)

USE	Contains the identifier for the DAT block.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,COD,reg ;COD field to 1-byte dst $COMPARE src,COD,reg ;1-byte src with COD field

6.2.4 EDT Field in DAT Block

USE	Contains the expiration date for the file.
INIT	None
SIZE	4 words
ACCESS	$FETCH dst,EDT,reg ;EDT field to 4-word dst
OUTPUT	OPEN File expiration date DISPLAY File expiration date

6.2.5 NXT Field in DAT Block

USE Contains the address of the next XAB (ALL, DAT, KEY, PRO, or SUM block) in a chain of XABs.

INIT X$NXT address

SIZE 1 word

ACCESS

```
$FETCH dst,NXT,reg      ;NXT field to 1-word dst
$STORE src,NXT,reg      ;1-word src to NXT field
$COMPARE src,NXT,reg    ;1-word src with NXT field
```

INPUT

CLOSE	Next XAB address
CREATE	Next XAB address
DISPLAY	Next XAB address
ENTER	Next XAB address
ERASE	Next XAB address
EXTEND	Next XAB address
OPEN	Next XAB address
PARSE	Next XAB address
REMOVE	Next XAB address
RENAME	Next XAB address
SEARCH	Next XAB address

6.2.6 RDT Field in DAT Block

USE	Contains the binary revision date for the file.
INIT	None
SIZE	4 words
ACCESS	$FETCH dst,RDT,reg ;RDT field to 4-word dst
OUTPUT	DISPLAY File revision date OPEN File revision date

6.2.7 RVN Field in DAT Block

USE Contains the revision number (number of times closed) for the file.

INIT None

SIZE 1 word

ACCESS
```
$FETCH dst,RVN,reg      ;RVN field to 1-word dst
$COMPARE src,RVN,reg    ;1-word src with RVN field
```

OUTPUT
```
DISPLAY     File revision number
OPEN        File revision number
```

6.3 FAB SUMMARY

This section summarizes the FAB and its fields. Table 6-3 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-3: FAB Summary

Offset	Offset Symbol	Field Size	Description
000	O$BID	1 byte	FAB identifier
			000003 FB$BID FAB identification code
001	O$BLN	1 byte	FAB length (bytes)
			000120 FB$BLN FAB length (bytes)
002	O$CTX	1 word	User context
004	O$IFI	1 word	Internal FAB identifier
006	O$STS	1 word	Completion status code
010	O$STV	1 word	Completion status value
012	O$ALQ	2 words	File allocation size (blocks)
016	O$DEQ	1 word	File default extension size (blocks)
020	O$FAC	1 byte	Requested access mask
			000001 FB$PUT Request put access
			000002 FB$GET Request find/get access
			000004 FB$DEL Request find/get/delete access
			000010 FB$UPD Request find/get/update access
			000020 FB$TRN Request find/get/truncate access
			000041 FB$WRT Request block write access
			000042 FB$REA Request block read access
021	O$SHR	1 byte	Shared access mask
			000002 FB$GET Share find/get access
			000015 FB$WRI Share find/get/put/update/delete access
			000040 FB$UPI Share any access (user-provided interlock)
			000100 FB$NIL No access sharing

(continued on next page)

Table 6-3 (cont.): FAB Summary

Offset	Offset Symbol	Field Size	Description
022	O$FOP	1 word	File processing option mask
			000020 FB$DLK No file locking on abnormal close
			000200 FB$CTG Contiguous file
			000400 FB$SUP Supersede existing file
			002000 FB$TMP Temporary file
			004000 FB$MKD Mark file for deletion
			006000 FB$TMD Temporary file, mark for deletion
			010000 FB$FID Use information in NAM block
			020000 FB$DFW Defer writing
024	O$RTV	1 byte	Retrieval pointer count
025	O$ORG	1 byte	File organization code
			000000 FB$SEQ Sequential file organization
			000020 FB$REL Relative file organization
			000040 FB$IDX Indexed file organization
026	O$RAT	1 byte	Record handling mask
			000001 FB$FTN FORTRAN-style carriage-control character in record
			000002 FB$CR Add CRLF to print record (LF-record-CR)
			000004 FB$PRN VFC print record handling
			000010 FB$BLK Blocked records
027	O$RFM	1 byte	Record format code
			000000 FB$UDF Undefined record format
			000001 FB$FIX Fixed-length record format
			000002 FB$VAR Variable-length record format
			000003 FB$VFC VFC record format
			000004 FB$STM Stream record format
030	O$XAB	1 word	XAB address
032	O$BPA	1 word	Private buffer pool address
034	O$BPS	1 word	Private buffer pool size (bytes)
036	O$MRS	1 word	Maximum record size (bytes)
040	O$MRN	2 words	Maximum record number
044	O$LRL	1 word	Longest record length
046	O$NAM	1 word	NAM block address
050	O$FNA	1 word	File string address
052	O$DNA	1 word	Default string address
054	O$FNS	1 byte	File string size (bytes)

(continued on next page)

Table 6-3 (cont.): FAB Summary

Offset	Offset Symbol	Field Size	Description
055	O$DNS	1 byte	Default string size (bytes)
056	O$BLS	1 word	Magtape block size (characters)
060	O$FSZ	1 byte	Fixed control area size for VFC records (bytes)
061	O$BKS	1 byte	File bucket size (blocks)
062	O$DEV	1 byte	Device characteristic mask
			000001 FB$REC Record-oriented device
			000002 FB$CCL Carriage-control device
			000004 FB$TRM Terminal device
			000010 FB$MDI Multidirectory device
			000020 FB$SDI Single-directory device
			000040 FB$SQD Sequential device
063	O$LCH	1 byte	Logical channel number

6.3.1 ALQ Field in FAB

USE Contains the allocation size for the file.

INIT F$ALQ number

SIZE 2 words

ACCESS

```
$FETCH dst,ALQ,reg      ;ALQ field to 2-word dst
$STORE src,ALQ,reg      ;2-word src to ALQ field
$FETCH dst,ALQn,reg     ;ALQ word n to 1-word dst
$STORE src,ALQn,reg     ;1-word src to ALQ word n
$COMPARE src,ALQn,reg   ;1-word src with ALQ word n
```

INPUT

CREATE	Initial file allocation request size (blocks)
EXTEND	File allocation extension request size (blocks)

OUTPUT

EXTEND	File allocation extension actual size (blocks)
OPEN	Current file allocation (blocks)

6.3.2 BID Field in FAB (FB$BID Code)

USE	Contains the identifier for the FAB.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,BID,reg ;BID field to 1-byte dst $COMPARE src,BID,reg ;1-byte src with BID field

6.3.3 BKS Field in FAB

USE Contains the bucket size for the file.

INIT F$BKS number

SIZE 1 byte

ACCESS

```
$FETCH dst,BKS,reg      ;BKS field to 1-byte dst
$STORE src,BKS,reg      ;1-byte src to BKS field
$COMPARE src,BKS,reg    ;1-byte src with BKS field
```

INPUT CREATE File bucket size (blocks)

OUTPUT OPEN File bucket size (blocks)

6.3.4 BLN Field in FAB (FB$BLN Code)

USE	Contains the length of the FAB.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,BLN,reg ;BLN field to 1-byte dst $COMPARE src,BLN,reg ;1-byte src with BLN field

6.3.5 BPA Field in FAB

USE Contains the address of the private buffer pool for the operation.

INIT F$BPA address

SIZE 1 word

ACCESS

```
$FETCH dst,BPA,reg      ;BPA field to 1-word dst
$STORE src,BPA,reg      ;1-word src to BPA field
$COMPARE src,BPA,reg    ;1-word src with BPA field
```

INPUT

CREATE	Private buffer pool address
ENTER	Private buffer pool address
ERASE	Private buffer pool address
OPEN	Private buffer pool address
PARSE	Private buffer pool address
REMOVE	Private buffer pool address
RENAME	Private buffer pool address
SEARCH	Private buffer pool address

OUTPUT

CLOSE	Private buffer pool address

6.3.6 BPS Field in FAB

USE Contains the size of the private buffer pool for the operation.

INIT F$BPS number

SIZE 1 word

ACCESS

```
$FETCH dst,BPS,reg      ;BPS field to 1-word dst
$STORE src,BPS,reg      ;1-word src to BPS field
$COMPARE src,BPS,reg    ;1-word src with BPS field
```

INPUT

CREATE	Private buffer pool size (bytes)
ENTER	Private buffer pool size (bytes)
ERASE	Private buffer pool size (bytes)
OPEN	Private buffer pool size (bytes)
PARSE	Private buffer pool size (bytes)
REMOVE	Private buffer pool size (bytes)
RENAME	Private buffer pool size (bytes)
SEARCH	Private buffer pool size (bytes)

OUTPUT

CLOSE	Private buffer pool size (bytes)

6.3.7 CTX Field in FAB

USE Contains any information you may want to associate with the file at run time.

INIT F$CTX number

SIZE 1 word

ACCESS

```
$FETCH dst,CTX,reg      ;CTX field to 1-word dst
$STORE src,CTX,reg      ;1-word src to CTX field
$COMPARE src,CTX,reg    ;1-word src with CTX field
```

6.3.8 DEQ Field in FAB

USE	Contains the default extension size for the file.	
INIT	F$DEQ number	
SIZE	1 word	
ACCESS	$FETCH dst,DEQ,reg	;DEQ field to 1-word dst
	$STORE src,DEQ,reg	;1-word src to DEQ field
	$COMPARE src,DEQ,reg	;1-word src with DEQ field
INPUT	CREATE	Permanent file default extension size (blocks)
	OPEN	While-open file default extension size (blocks)
OUTPUT	OPEN	Current file default extension size (blocks)

6.3.9 DEV Field in FAB

USE Indicates device characteristics for the file.

INIT None

SIZE 1 byte

ACCESS

```
$TESTBITS mask,DEV,reg ;Test mask bits in DEV field
$FETCH dst,DEV,reg     ;DEV field to 1-byte dst
$COMPARE src,DEV,reg   ;1-byte src with DEV field
```

MASKS

FB$CCL	Carriage-control device
FB$MDI	Multidirectory device
FB$REC	Record-oriented device
FB$SDI	Single-directory device
FB$SQD	Sequential device
FB$TRM	Terminal device

OUTPUT

CREATE	Device characteristic mask
ERASE	Device characteristic mask
OPEN	Device characteristic mask

6.3.10 DNA Field in FAB

USE Contains the address of the default string for the operation.

INIT F$DNA address

SIZE 1 word

ACCESS

```
$FETCH dst,DNA,reg      ;DNA field to 1-word dst
$STORE src,DNA,reg      ;1-word src to DNA field
$COMPARE src,DNA,reg    ;1-word src with DNA field
```

INPUT

CREATE	Default string address
ENTER	Default string address
ERASE	Default string address
OPEN	Default string address
PARSE	Default string address
REMOVE	Default string address
RENAME	Default string address

6.3.11 DNS Field in FAB

USE Contains the size of the default string for the operation.

INIT F$DNS number

SIZE 1 byte

ACCESS

```
$FETCH dst,DNS,reg      ;DNS field to 1-byte dst
$STORE src,DNS,reg      ;1-byte src to DNS field
$COMPARE src,DNS,reg    ;1-byte src with DNS field
```

INPUT

CREATE	Default string size (bytes)
ENTER	Default string size (bytes)
ERASE	Default string size (bytes)
OPEN	Default string size (bytes)
PARSE	Default string size (bytes)
REMOVE	Default string size (bytes)
RENAME	Default string size (bytes)

6.3.12 FAC Field in FAB

USE Indicates the requested access for the file.

INIT F$FAC mask

SIZE 1 byte

ACCESS

```
$SET mask,FAC,reg       ;Mask bits on in FAC field
$OFF mask,FAC,reg       ;Mask bits off in FAC field
$TESTBITS mask,FAC,reg  ;Test mask bits in FAC field
$FETCH dst,FAC,reg      ;FAC field to 1-byte dst
$STORE src,FAC,reg      ;1-byte src to FAC field
$COMPARE src,FAC,reg    ;1-byte src with FAC field
```

MASKS

```
FB$DEL  Request find/get/delete access
FB$GET  Request find/get access
FB$PUT  Request put access
FB$REA  Request block read access
FB$TRN  Request find/get/truncate access
FB$UPD  Request find/get/update access
FB$WRT  Request block write access
```

INPUT

```
CREATE      Requested access mask
OPEN        Requested access mask
```

6.3.13 FNA Field in FAB

USE Contains the address of the file string for the file.

INIT F$FNA address

SIZE 1 word

ACCESS

```
$FETCH dst,FNA,reg      ;FNA field to 1-word dst
$STORE src,FNA,reg      ;1-word src to FNA field
$COMPARE src,FNA,reg    ;1-word src with FNA field
```

INPUT

CREATE	File string address
ENTER	File string address
ERASE	File string address
OPEN	File string address
PARSE	File string address
REMOVE	File string address
RENAME	File string address

6.3.14 FNS Field in FAB

USE Contains the size of the file string for the file.

INIT F$FNS number

SIZE 1 byte

ACCESS

```
$FETCH dst,FNS,reg      ;FNS field to 1-byte dst
$STORE src,FNS,reg      ;1-byte src to FNS field
$COMPARE src,FNS,reg    ;1-byte src with FNS field
```

INPUT

CREATE	File string size (bytes)
ENTER	File string size (bytes)
ERASE	File string size (bytes)
OPEN	File string size (bytes)
PARSE	File string size (bytes)
REMOVE	File string size (bytes)
RENAME	File string size (bytes)

6.3.15 FOP Field in FAB (FB$CTG Mask)

USE Indicates file contiguity.

INIT F$FOP mask

SIZE 1 word

ACCESS

```
$SET mask,FOP,reg        ;Mask bits on in FOP field
$OFF mask,FOP,reg        ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg   ;Test mask bits in FOP field
$FETCH dst,FOP,reg       ;FOP field to 1-word dst
$STORE src,FOP,reg       ;1-word src to FOP field
$COMPARE src,FOP,reg     ;1-word src with FOP field
```

INPUT

CREATE	Contiguous file request
EXTEND	Contiguous file extension request

OUTPUT

OPEN	Contiguous file

6.3.16 FOP Field in FAB (FB$DFW Mask)

USE Requests deferred writing for the file.

INIT F$FOP mask

SIZE 1 word

ACCESS
```
$SET mask,FOP,reg       ;Mask bits on in FOP field
$OFF mask,FOP,reg       ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg  ;Test mask bits in FOP field
$FETCH dst,FOP,reg      ;FOP field to 1-word dst
$STORE src,FOP,reg      ;1-word src to FOP field
$COMPARE src,FOP,reg    ;1-word src with FOP field
```

INPUT
```
CREATE      Defer writing
OPEN        Defer writing
```

6.3.17 FOP Field in FAB (FB$DLK Mask)

USE Requests no file locking if the file is closed abnormally.

INIT F$FOP mask

SIZE 1 word

ACCESS

```
$SET mask,FOP,reg        ;Mask bits on in FOP field
$OFF mask,FOP,reg        ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg   ;Test mask bits in FOP field
$FETCH dst,FOP,reg       ;FOP field to 1-word dst
$STORE src,FOP,reg       ;1-word src to FOP field
$COMPARE src,FOP,reg     ;1-word src with FOP field
```

INPUT

CREATE	No file locking on abnormal close
OPEN	No file locking on abnormal close

6.3.18 FOP Field in FAB (FB$FID Mask)

USE Requests that NAM block information be used to identify the file.

INIT F$FOP mask

SIZE 1 word

ACCESS

```
$SET mask,FOP,reg       ;Mask bits on in FOP field
$OFF mask,FOP,reg       ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg  ;Test mask bits in FOP field
$FETCH dst,FOP,reg      ;FOP field to 1-word dst
$STORE src,FOP,reg      ;1-word src to FOP field
$COMPARE src,FOP,reg    ;1-word src with FOP field
```

INPUT

CREATE	Use information in NAM block
ENTER	Use information in NAM block
ERASE	Use information in NAM block
OPEN	Use information in NAM block
REMOVE	Use information in NAM block
RENAME	Use information in NAM block

6.3.19 FOP Field in FAB (FB$MKD Mask)

USE Requests that the file be marked for deletion.

INIT F$FOP mask

SIZE 1 word

ACCESS

```
$SET mask,FOP,reg       ;Mask bits on in FOP field
$OFF mask,FOP,reg       ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg  ;Test mask bits in FOP field
$FETCH dst,FOP,reg      ;FOP field to 1-word dst
$STORE src,FOP,reg      ;1-word src to FOP field
$COMPARE src,FOP,reg    ;1-word src with FOP field
```

INPUT

CREATE	Mark file for deletion

6.3.20 FOP Field in FAB (FB$SUP Mask)

USE Requests that the created file supersede the old file with the same specification (if one exists).

INIT F$FOP mask

SIZE 1 word

ACCESS
```
$SET mask,FOP,reg       ;Mask bits on in FOP field
$OFF mask,FOP,reg       ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg  ;Test mask bits in FOP field
$FETCH dst,FOP,reg      ;FOP field to 1-word dst
$STORE src,FOP,reg      ;1-word src to FOP field
$COMPARE src,FOP,reg    ;1-word src with FOP field
```

INPUT CREATE Supersede existing file

6.3.21 FOP Field in FAB (FB$TMP Mask)

USE — Requests that the created file be a temporary file (one with no directory entry).

INIT — F$FOP mask

SIZE — 1 word

ACCESS

```
$SET mask,FOP,reg        ;Mask bits on in FOP field
$OFF mask,FOP,reg        ;Mask bits off in FOP field
$TESTBITS mask,FOP,reg ;Test mask bits in FOP field
$FETCH dst,FOP,reg       ;FOP field to 1-word dst
$STORE src,FOP,reg       ;1-word src to FOP field
$COMPARE src,FOP,reg     ;1-word src with FOP field
```

INPUT — CREATE Temporary file

6.3.22 FSZ Field in FAB

USE	Contains the size of the fixed control area for VFC records.	
INIT	F$FSZ number	
SIZE	1 byte	
ACCESS	$FETCH dst,FSZ,reg	;FSZ field to 1-byte dst
	$STORE src,FSZ,reg	;1-byte src to FSZ field
	$COMPARE src,FSZ,reg	;1-byte src with FSZ field
INPUT	CREATE	Fixed control area size for VFC records (bytes)
OUTPUT	OPEN	Fixed control area size for VFC records (bytes)

6.3.23 IFI Field in FAB

USE	Contains the internal file identifier for the file.	
INIT	None	
SIZE	1 word	
ACCESS	$FETCH dst,IFI,reg	;IFI field to 1-word dst
	$COMPARE src,IFI,reg	;1-word src with IFI field
INPUT	CLOSE	Internal FAB identifier
	CONNECT	Internal FAB identifier
	DISPLAY	Internal FAB identifier
	EXTEND	Internal FAB identifier
OUTPUT	CLOSE	Internal FAB identifier
	CREATE	Internal FAB identifier
	OPEN	Internal FAB identifier

6.3.24 LCH Field in FAB

USE — Contains the logical channel number for the operation.

INIT — F$LCH number

SIZE — 1 byte

ACCESS

```
$FETCH dst,LCH,reg      ;LCH field to 1-byte dst
$STORE src,LCH,reg      ;1-byte src to LCH field
$COMPARE src,LCH,reg    ;1-byte src with LCH field
```

INPUT

CREATE	Logical channel number
ENTER	Logical channel number
ERASE	Logical channel number
OPEN	Logical channel number
PARSE	Logical channel number
REMOVE	Logical channel number
RENAME	Logical channel number
SEARCH	Logical channel number

6.3.25 LRL Field in FAB

USE	Contains the length of the longest record in a sequential file.	
INIT	None	
SIZE	1 word	
ACCESS	$FETCH dst,LRL,reg $COMPARE src,LRL,reg	;LRL field to 1-word dst ;1-word src with LRL field
INPUT	CREATE	Longest record length (block access to sequential files only)
OUTPUT	OPEN	Longest record length

6.3.26 MRN Field in FAB

USE Contains the maximum record number allowed in a relative file.

INIT F$MRN number

SIZE 2 words

ACCESS

```
$FETCH dst,MRN,reg      ;MRN field to 2-word dst
$STORE src,MRN,reg      ;2-word src to MRN field
$FETCH dst,MRNn,reg     ;MRN word n to 1-word dst
$STORE src,MRNn,reg     ;1-word src to MRN word n
$COMPARE src,MRNn,reg   ;1-word src with MRN word n
```

INPUT CREATE Maximum record number

OUTPUT OPEN Maximum record number

6.3.27 MRS Field in FAB

USE	Contains the record size or maximum record size for the file.
INIT	F$MRS number
SIZE	1 word
ACCESS	$FETCH dst,MRS,reg ;MRS field to 1-word dst $STORE src,MRS,reg ;1-word src to MRS field $COMPARE src,MRS,reg ;1-word src with MRS field
INPUT	CREATE Maximum record size (bytes)
OUTPUT	OPEN Maximum record size (bytes)

6.3.28 NAM Field in FAB

USE Contains the address of the NAM block for the operation.

INIT F$NAM address

SIZE 1 word

ACCESS

```
$FETCH dst,NAM,reg      ;NAM field to 1-word dst
$STORE src,NAM,reg      ;1-word src to NAM field
$COMPARE src,NAM,reg    ;1-word src with NAM field
```

INPUT

CREATE	NAM block address
ENTER	NAM block address
ERASE	NAM block address
OPEN	NAM block address
PARSE	NAM block address
REMOVE	NAM block address
RENAME	NAM block address
SEARCH	NAM block address

6.3.29 ORG Field in FAB

USE Contains the file organization code.

INIT F$ORG code

SIZE 1 byte

ACCESS

```
$FETCH dst,ORG,reg      ;ORG field to 1-byte dst
$STORE src,ORG,reg      ;1-byte src to ORG field
$COMPARE src,ORG,reg    ;1-byte src with ORG field
```

CODES

```
FB$IDX  Indexed file organization
FB$REL  Relative file organization
FB$SEQ  Sequential file organization
```

INPUT CREATE File organization code

OUTPUT OPEN File organization code

6.3.30 RAT Field in FAB

USE Indicates the record-output characteristic for the file. (The RAT field also contains the record-blocking characteristic, which is described in the next section.)

INIT F$RAT mask

SIZE 1 byte

ACCESS

```
$SET mask,RAT,reg       ;Mask bits on in RAT field
$OFF mask,RAT,reg       ;Mask bits off in RAT field
$TESTBITS mask,RAT,reg  ;Test mask bits in RAT field
$FETCH dst,RAT,reg      ;RAT field to 1-byte dst
$STORE src,RAT,reg      ;1-byte src to RAT field
$COMPARE src,RAT,reg    ;1-byte src with RAT field
```

MASKS

FB$CR	Add CRLF to print record (LF-record-CR)
FB$FTN	FORTRAN-style carriage-control character in record
FB$PRN	VFC print record handling

INPUT CREATE Record handling mask

OUTPUT OPEN Record handling mask

6.3.31 RAT Field in FAB (FB$BLK Mask)

USE Indicates whether the file has blocked records. (The RAT field also contains the record-output characteristic, which is described in the previous section.)

INIT F$RAT mask

SIZE 1 byte

ACCESS

```
$SET mask,RAT,reg       ;Mask bits on in RAT field
$OFF mask,RAT,reg       ;Mask bits off in RAT field
$TESTBITS mask,RAT,reg  ;Test mask bits in RAT field
$FETCH dst,RAT,reg      ;RAT field to 1-byte dst
$STORE src,RAT,reg      ;1-byte src to RAT field
$COMPARE src,RAT,reg    ;1-byte src with RAT field
```

INPUT	CREATE	Blocked records
OUTPUT	OPEN	Blocked records

6.3.32 RFM Field in FAB

USE	Contains the record format code for the file.
INIT	F$RFM code
SIZE	1 byte
ACCESS	$FETCH dst,RFM,reg ;RFM field to 1-byte dst $STORE src,RFM,reg ;1-byte src to RFM field $COMPARE src,RFM,reg ;1-byte src with RFM field
CODES	FB$FIX Fixed-length record format FB$STM Stream record format FB$UDF Undefined record format FB$VAR Variable-length record format FB$VFC VFC record format
INPUT	CREATE Record format code
OUTPUT	OPEN Record format code

6.3.33 RTV Field in FAB

USE Contains the retrieval pointer count for the file.

INIT F$RTV number

SIZE 1 byte

ACCESS

```
$FETCH dst,RTV,reg      ;RTV field to 1-byte dst
$STORE src,RTV,reg      ;1-byte src to RTV field
$COMPARE src,RTV,reg    ;1-byte src with RTV field
```

INPUT

```
CREATE      Retrieval pointer count
OPEN        Retrieval pointer count
```

6.3.34 SHR Field in FAB

USE Indicates requested access sharing for the file.

INIT F$SHR mask

SIZE 1 byte

ACCESS

```
$SET mask,SHR,reg       ;Mask bits on in SHR field
$OFF mask,SHR,reg       ;Mask bits off in SHR field
$TESTBITS mask,SHR,reg  ;Test mask bits in SHR field
$FETCH dst,SHR,reg      ;SHR field to 1-byte dst
$STORE src,SHR,reg      ;1-byte src to SHR field
$COMPARE src,SHR,reg    ;1-byte src with SHR field
```

MASKS

```
FB$GET  Share find/get access
FB$NIL  No access sharing
FB$UPI  Share any access (user-provided interlock)
FB$WRI  Share find/get/put/update/delete access
```

INPUT

```
CREATE      Shared access mask
OPEN        Shared access mask
```

6.3.35 STS Field in FAB

USE Contains the completion status code for the operation.

INIT None

SIZE 1 word

ACCESS

```
$FETCH dst,STS,reg      ;STS field to 1-word dst
$COMPARE src,STS,reg    ;1-word src with STS field
```

OUTPUT

CLOSE	Completion status code
CREATE	Completion status code
DISPLAY	Completion status code
ENTER	Completion status code
ERASE	Completion status code
EXTEND	Completion status code
OPEN	Completion status code
PARSE	Completion status code
REMOVE	Completion status code
RENAME	Completion status code
SEARCH	Completion status code

6.3.36 STV Field in FAB

USE — Contains the completion status value for the operation.

INIT — None

SIZE — 1 word

ACCESS

```
$FETCH dst,STV,reg      ;STV field to 1-word dst
$COMPARE src,STV,reg    ;1-word src with STV field
```

OUTPUT

CLOSE	Completion status value
CREATE	Completion status value
DISPLAY	Completion status value
ENTER	Completion status value
ERASE	Completion status value
EXTEND	Completion status value
OPEN	Completion status value
PARSE	Completion status value
REMOVE	Completion status value
RENAME	Completion status value
SEARCH	Completion status value

6.3.37 XAB Field in FAB

USE Contains the address of the first XAB (ALL, DAT, KEY, PRO, or SUM block) in a chain of XABs.

INIT F$XAB address

SIZE 1 word

ACCESS

```
$FETCH dst,XAB,reg      ;XAB field to 1-word dst
$STORE src,XAB,reg      ;1-word src to XAB field
$COMPARE src,XAB,reg    ;1-word src with XAB field
```

INPUT

CLOSE	XAB address
CREATE	XAB address
DISPLAY	XAB address
EXTEND	XAB address
OPEN	XAB address

6.4 KEY BLOCK SUMMARY

This section summarizes the KEY block and its fields. Table 6-4 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-4: KEY Block Summary

Offset	Offset Symbol	Field Size	Description
000	O$COD	1 byte	KEY block identifier code
			000001 XB$KEY KEY block identifier
001	O$BLN	1 byte	KEY block length (bytes)
			000070 XB$KYL KEY block length (bytes)
002	O$NXT	1 word	Next XAB address
004	O$REF	1 byte	Index reference number
005	O$LVL	1 byte	Number of index levels (not including data level)
006	O$IFL	1 word	Index bucket fill factor
010	O$DFL	1 word	Data bucket fill factor
012	O$NUL	1 byte	Null key character
013	O$IAN	1 byte	Higher level index area number
014	O$LAN	1 byte	Lowest index level area number
015	O$DAN	1 byte	Data area number
016	O$FLG	1 byte	Index option mask
			000001 XB$DUP Duplicate record keys allowed
			000002 XB$CHG Record key changes allowed on update
			000020 XB$INI No entries yet made in index
			000004 XB$NUL Null record keys not indexed
017	O$DTP	1 byte	Key data type code
			000000 XB$STG String
			000001 XB$IN2 15-bit signed integer
			000002 XB$BN2 16-bit unsigned integer
			000003 XB$IN4 31-bit signed integer
			000004 XB$BN4 32-bit unsigned integer
			000005 XB$PAC Packed decimal number

(continued on next page)

Table 6-4 (cont.): KEY Block Summary

Offset	Offset Symbol	Field Size	Description
020	O$KNM	1 word	Key name buffer address
022	O$POS	8 words	Key segment positions
022	O$POS0	1 word	Key segment 0 position
024	O$POS1	1 word	Key segment 1 position
026	O$POS2	1 word	Key segment 2 position
030	O$POS3	1 word	Key segment 3 position
032	O$POS4	1 word	Key segment 4 position
034	O$POS5	1 word	Key segment 5 position
036	O$POS6	1 word	Key segment 6 position
040	O$POS7	1 word	Key segment 7 position
042	O$SIZ	8 bytes	Key segment sizes (bytes)
042	O$SIZ0	1 byte	Key segment 0 size (bytes)
043	O$SIZ1	1 byte	Key segment 1 size (bytes)
044	O$SIZ2	1 byte	Key segment 2 size (bytes)
045	O$SIZ3	1 byte	Key segment 3 size (bytes)
046	O$SIZ4	1 byte	Key segment 4 size (bytes)
047	O$SIZ5	1 byte	Key segment 5 size (bytes)
050	O$SIZ6	1 byte	Key segment 6 size (bytes)
051	O$SIZ7	1 byte	Key segment 7 size (bytes)
052	O$RVB	2 words	Root index bucket virtual block number
056	O$DVB	2 words	First data bucket virtual block number
062	O$IBS	1 byte	Index area bucket size (blocks)
063	O$DBS	1 byte	Data area bucket size (blocks)
064	O$NSG	1 byte	Key segment count
065	O$TKS	1 byte	Total key size (sum of key segment sizes) (bytes)
066	O$MRL	1 word	Minimum length of record containing key (bytes)

6.4.1 BLN Field in KEY Block (XB$KYL Code)

USE	Contains the length of the KEY block.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,BLN,reg ;BLN field to 1-byte dst $COMPARE src,BLN,reg ;1-byte src with BLN field

6.4.2 COD Field in KEY Block (XB$KEY Code)

USE Contains the identifier for the KEY block.

INIT None

SIZE 1 byte

ACCESS

```
$FETCH dst,COD,reg      ;COD field to 1-byte dst
$COMPARE src,COD,reg    ;1-byte src with COD field
```

6.4.3 DAN Field in KEY Block

USE Contains the area number of the data area for the index described by the KEY block.

INIT X$DAN number

SIZE 1 byte

ACCESS

```
$FETCH dst,DAN,reg      ;DAN field to 1-byte dst
$STORE src,DAN,reg      ;1-byte src to DAN field
$COMPARE src,DAN,reg    ;1-byte src with DAN field
```

INPUT

CREATE	Data area number

OUTPUT

DISPLAY	Data area number
OPEN	Data area number

6.4.4 DBS Field in KEY Block

USE	Contains the bucket size for the data area for the index described by the KEY block.
INIT	None
SIZE	1 byte
ACCESS	`$FETCH dst,DBS,reg ;DBS field to 1-byte dst` `$COMPARE src,DBS,reg ;1-byte src with DBS field`
OUTPUT	DISPLAY Data area bucket size (blocks) OPEN Data area bucket size (blocks)

6.4.5 DFL Field in KEY Block

USE Contains the bucket fill number for the data area for the index described by the KEY block.

INIT X$DFL number

SIZE 1 word

ACCESS

```
$FETCH dst,DFL,reg      ;DFL field to 1-word dst
$STORE src,DFL,reg      ;1-word src to DFL field
$COMPARE src,DFL,reg    ;1-word src with DFL field
```

INPUT

CREATE	Data bucket fill factor

OUTPUT

DISPLAY	Data bucket fill factor
OPEN	Data bucket fill factor

6.4.6 DTP Field in KEY Block

USE Contains the key data type code for the index described by the KEY block.

INIT X$DTP code

SIZE 1 byte

ACCESS

```
$FETCH dst,DTP,reg      ;DTP field to 1-byte dst
$STORE src,DTP,reg      ;1-byte src to DTP field
$COMPARE src,DTP,reg    ;1-byte src with DTP field
```

CODES

```
XB$BN2   16-bit unsigned integer
XB$BN4   32-bit unsigned integer
XB$IN2   15-bit signed integer
XB$IN4   31-bit signed integer
XB$PAC   Packed decimal number
XB$STG   String
```

INPUT

```
CREATE      Key data type code
```

OUTPUT

```
DISPLAY     Key data type code
OPEN        Key data type code
```

6.4.7 DVB Field in KEY Block

USE	Contains the virtual block number of the first bucket in the data area for the index described by the KEY block.
INIT	None
SIZE	2 words
ACCESS	$FETCH dst,DVB,reg ;DVB field to 2-word dst
OUTPUT	DISPLAY First data bucket virtual block number OPEN First data bucket virtual block number

6.4.8 FLG Field in KEY Block (XB$CHG Mask)

USE Specifies that a record key (for an alternate index) is allowed to change when the record is updated.

INIT X$FLG mask

SIZE 1 byte

ACCESS

```
$SET mask,FLG,reg       ;Mask bits on in FLG field
$OFF mask,FLG,reg       ;Mask bits off in FLG field
$TESTBITS mask,FLG,reg  ;Test mask bits in FLG field
$FETCH dst,FLG,reg      ;FLG field to 1-byte dst
$STORE src,FLG,reg      ;1-byte src to FLG field
$COMPARE src,FLG,reg    ;1-byte src with FLG field
```

INPUT

CREATE	Record key changes allowed on update

OUTPUT

DISPLAY	Record key changes allowed on update
OPEN	Record key changes allowed on update

6.4.9 FLG Field in KEY Block (XB$DUP Mask)

USE Indicates that duplicate record keys are allowed for the index described by the KEY block; duplicate record keys are not allowed in the primary index.

INIT X$FLG mask

SIZE 1 byte

ACCESS

```
$SET mask,FLG,reg       ;Mask bits on in FLG field
$OFF mask,FLG,reg       ;Mask bits off in FLG field
$TESTBITS mask,FLG,reg  ;Test mask bits in FLG field
$FETCH dst,FLG,reg      ;FLG field to 1-byte dst
$STORE src,FLG,reg      ;1-byte src to FLG field
$COMPARE src,FLG,reg    ;1-byte src with FLG field
```

INPUT

CREATE	Duplicate record keys allowed

OUTPUT

DISPLAY	Duplicate record keys allowed
OPEN	Duplicate record keys allowed

6.4.10 FLG Field in KEY Block (XB$NUL Mask)

USE Indicates that records containing only null characters are not contained in the index described by the KEY block. (The null character is specified in the NUL field of the KEY block.)

INIT X$FLG mask

SIZE 1 byte

ACCESS

```
$SET mask,FLG,reg       ;Mask bits on in FLG field
$OFF mask,FLG,reg       ;Mask bits off in FLG field
$TESTBITS mask,FLG,reg  ;Test mask bits in FLG field
$FETCH dst,FLG,reg      ;FLG field to 1-byte dst
$STORE src,FLG,reg      ;1-byte src to FLG field
$COMPARE src,FLG,reg    ;1-byte src with FLG field
```

INPUT

CREATE	Null record keys not indexed

OUTPUT

DISPLAY	Null record keys not indexed
OPEN	Null record keys not indexed

6.4.11 IAN Field in KEY Block

USE Contains the area number of the area containing the higher index levels (all except the lowest level) for the index described by the KEY block.

INIT X$IAN number

SIZE 1 byte

ACCESS

```
$FETCH dst,IAN,reg      ;IAN field to 1-byte dst
$STORE src,IAN,reg      ;1-byte src to IAN field
$COMPARE src,IAN,reg    ;1-byte src with IAN field
```

INPUT

CREATE	Higher level index area number

OUTPUT

DISPLAY	Higher level index area number
OPEN	Higher level index area number

6.4.12 IBS Field in KEY Block

USE Contains the bucket size of the area containing the index described by the KEY block.

INIT None

SIZE 1 byte

ACCESS

```
$FETCH dst,IBS,reg      ;IBS field to 1-byte dst
$COMPARE src,IBS,reg    ;1-byte src with IBS field
```

OUTPUT

```
DISPLAY     Index area bucket size (blocks)
OPEN        Index area bucket size (blocks)
```

6.4.13 IFL Field in KEY Block

USE	Contains the bucket fill number for the area containing the index described by the KEY block.
INIT	X$IFL number
SIZE	1 word
ACCESS	$FETCH dst,IFL,reg ;IFL field to 1-word dst $STORE src,IFL,reg ;1-word src to IFL field $COMPARE src,IFL,reg ;1-word src with IFL field
INPUT	CREATE Index bucket fill factor
OUTPUT	DISPLAY Index bucket fill factor OPEN Index bucket fill factor

6.4.14 KNM Field in KEY Block

USE Contains the address of the 32-byte key name buffer for the index described by the KEY block.

INIT X$KNM address

SIZE 1 word

ACCESS

```
$FETCH dst,KNM,reg      ;KNM field to 1-word dst
$STORE src,KNM,reg      ;1-word src to KNM field
$COMPARE src,KNM,reg    ;1-word src with KNM field
```

INPUT

CREATE	Key name buffer address
DISPLAY	Key name buffer address
OPEN	Key name buffer address

6.4.15 LAN Field in KEY Block

USE Contains the area number of the area containing the lowest level of the index described by the KEY block.

INIT X$LAN number

SIZE 1 byte

ACCESS

```
$FETCH dst,LAN,reg      ;LAN field to 1-byte dst
$STORE src,LAN,reg      ;1-byte src to LAN field
$COMPARE src,LAN,reg    ;1-byte src with LAN field
```

INPUT

CREATE	Lowest index level area number

OUTPUT

DISPLAY	Lowest index level area number
OPEN	Lowest index level area number

6.4.16 LVL Field in KEY Block

USE	Contains the number of levels (not including the data level) for the index described by the KEY block.	
INIT	None	
SIZE	1 byte	
ACCESS	`$FETCH dst,LVL,reg`	`;LVL field to 1-byte dst`
	`$COMPARE src,LVL,reg`	`;1-byte src with LVL field`
OUTPUT	DISPLAY	Number of index levels (not including data level)
	OPEN	Number of index levels (not including data level)

6.4.17 MRL Field in KEY Block

USE	Contains the length of the smallest record that is long enough to completely contain a record key for the index described by the KEY block.
INIT	None
SIZE	1 word
ACCESS	$FETCH dst,MRL,reg ;MRL field to 1-word dst $COMPARE src,MRL,reg ;1-word src with MRL field
OUTPUT	DISPLAY Minimum length of record containing key (bytes) OPEN Minimum length of record containing key (bytes)

6.4.18 NSG Field in KEY Block

USE Contains the number of key segments in the key for the index described by the KEY block.

INIT None

SIZE 1 byte

ACCESS
```
$FETCH dst,NSG,reg      ;NSG field to 1-byte dst
$COMPARE src,NSG,reg    ;1-byte src with NSG field
```

OUTPUT
```
DISPLAY     Key segment count
OPEN        Key segment count
```

6.4.19 NUL Field in KEY Block

USE Contains the null character for the (alternate) index described by the KEY block. For a string key (XB$STG in the DTP field of the KEY block), the NUL field contains an ASCII character; for any other key data type, the NUL field is unused (nonstring keys use 0 as the null value when the XB$NUL mask is set).

INIT X$NUL number

SIZE 1 byte

ACCESS

```
$FETCH dst,NUL,reg      ;NUL field to 1-byte dst
$STORE src,NUL,reg      ;1-byte src to NUL field
$COMPARE src,NUL,reg    ;1-byte src with NUL field
```

INPUT

CREATE	Null key character

OUTPUT

DISPLAY	Null key character
OPEN	Null key character

6.4.20 NXT Field in KEY Block

USE — Contains the address of the next XAB (ALL, DAT, KEY, PRO, or SUM block) in a chain of XABs.

INIT — X$NXT address

SIZE — 1 word

ACCESS

```
$FETCH dst,NXT,reg      ;NXT field to 1-word dst
$STORE src,NXT,reg      ;1-word src to NXT field
$COMPARE src,NXT,reg    ;1-word src with NXT field
```

INPUT

CLOSE	Next XAB address
CREATE	Next XAB address
DISPLAY	Next XAB address
ENTER	Next XAB address
ERASE	Next XAB address
EXTEND	Next XAB address
OPEN	Next XAB address
PARSE	Next XAB address
REMOVE	Next XAB address
RENAME	Next XAB address
SEARCH	Next XAB address

6.4.21 POS Field in KEY Block

USE Contains the positions of segments for the record keys in the index described by the KEY block. (The first key position is position 0.)

INIT X$POS <number[,number]...>

SIZE 8 words

ACCESS

```
$FETCH dst,POS,reg      ;POS field to 8-word dst
$STORE src,POS,reg      ;8-word src to POS field
$FETCH dst,POSn,reg     ;POS word n to 1-word dst
$STORE src,POSn,reg     ;1-word src to POS word n
$COMPARE src,POSn,reg   ;1-word src with POS word n
```

INPUT

CREATE	Key segment positions

OUTPUT

DISPLAY	Key segment positions
OPEN	Key segment positions

6.4.22 REF Field in KEY Block

USE Contains the reference number for the index described by the KEY block.

INIT X$REF number

SIZE 1 byte

ACCESS

```
$FETCH dst,REF,reg      ;REF field to 1-byte dst
$STORE src,REF,reg      ;1-byte src to REF field
$COMPARE src,REF,reg    ;1-byte src with REF field
```

INPUT

CLOSE	Index reference number
CREATE	Index reference number
DISPLAY	Index reference number
EXTEND	Index reference number
OPEN	Index reference number

6.4.23 RVB Field in KEY Block

USE	Contains the virtual block number of the first block of the root bucket of the index described by the KEY block.
INIT	None
SIZE	2 words
ACCESS	$FETCH dst,RVB,reg ;RVB field to 2-word dst
OUTPUT	DISPLAY Root index bucket virtual block number OPEN Root index bucket virtual block number

6.4.24 SIZ Field in KEY Block

USE Contains the sizes of segments for the record keys in the index described by the KEY block.

INIT X$SIZ <number[,number]...>

SIZE 8 bytes

ACCESS

```
$FETCH dst,SIZ,reg      ;SIZ field to 8-byte dst
$STORE src,SIZ,reg      ;8-byte src to SIZ field
$FETCH dst,SIZn,reg     ;SIZ byte n to 1-byte dst
$STORE src,SIZn,reg     ;1-byte src to SIZ byte n
$COMPARE src,SIZn,reg   ;1-byte src with SIZ byte n
```

INPUT	CREATE	Key segment sizes (bytes)
OUTPUT	DISPLAY	Key segment sizes (bytes)
	OPEN	Key segment sizes (bytes)

6.4.25 TKS Field in KEY Block

USE	Contains the total key size (sum of the segment sizes) of a record key for the index described by the KEY block.	
INIT	None	
SIZE	1 byte	
ACCESS	$FETCH dst,TKS,reg	;TKS field to 1-byte dst
	$COMPARE src,TKS,reg	;1-byte src with TKS field
OUTPUT	DISPLAY	Total key size (sum of key segment sizes) (bytes)
	OPEN	Total key size (sum of key segment sizes) (bytes)

6.5 NAM BLOCK SUMMARY

This section summarizes the NAM block and its fields. Table 6-5 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-5: NAM Block Summary

Offset	Offset Symbol	Field Size	Description		
000	O$RLF	1 word	Reserved		
002	O$RSA	1 word	Resultant string buffer address		
004	O$RSS	1 byte	Resultant string buffer size (bytes)		
005	O$RSL	1 byte	Resultant string length (bytes)		
006	O$DVI	2 words	Device identifier		
012	O$WDI	1 word	Wildcard directory context		
014	O$FID	3 words	File identifier		
022	O$DID	3 words	Directory identifier		
030	O$FNB	1 word	File specification mask		
			000001	NB$VER	File version in file string or default string
			000002	NB$TYP	File type in file string or default string
			000004	NB$NAM	File name in file string or default string
			000010	NB$WVE	Wildcard file version in file string or default string
			000020	NB$WTY	Wildcard file type in file string or default string
			000040	NB$WNA	Wildcard file name in file string or default string
			000100	NB$DIR	Directory in file string or default string
			000200	NB$DEV	Device in file string or default string
			000400	NB$NOD	Node in file string or default string
			001000	NB$WDI	Wildcard directory in file string or default string
			002000	NB$QUO	Quoted string in file string or default string
			004000	NB$WCH	Wildcard context established

(continued on next page)

Table 6-5 (cont.): NAM Block Summary

Offset	Offset Symbol	Field Size	Description
032	O$ESA	1 word	Expanded string buffer address
034	O$ESS	1 byte	Expanded string buffer size (bytes)
035	O$ESL	1 byte	Expanded string length (bytes)
036	O$WCC	1 word	Wildcard context

The first word of the NAM block is currently reserved, as noted above, and must contain the value 0. If the NAM block is extended in the future, the first byte will contain an identifier and the second byte will contain the (new) block length.

6.5.1 DID Field in NAM Block

USE	Contains the directory identifier for the target file.	
INIT	None	
SIZE	3 words	
ACCESS	$FETCH dst,DID,reg ;DID field to 3-word dst	
INPUT	CREATE	Directory identifier
	ENTER	Directory identifier
	ERASE	Directory identifier
	OPEN	Directory identifier
	REMOVE	Directory identifier
	RENAME	Directory identifier
	SEARCH	Directory identifier
OUTPUT	CREATE	Directory identifier
	ENTER	Directory identifier
	ERASE	Directory identifier
	OPEN	Directory identifier
	REMOVE	Directory identifier
	RENAME	Directory identifier
	SEARCH	Directory identifier

6.5.2 DVI Field in NAM Block

USE	Contains the device identifier for the target file.	
INIT	None	
SIZE	2 words	
ACCESS	$FETCH dst,DVI,reg	;DVI field to 2-word dst
INPUT	CREATE	Device identifier
	ENTER	Device identifier
	ERASE	Device identifier
	OPEN	Device identifier
	REMOVE	Device identifier
	RENAME	Device identifier
	SEARCH	Device identifier
OUTPUT	CREATE	Device identifier
	ENTER	Device identifier
	ERASE	Device identifier
	OPEN	Device identifier
	PARSE	Device identifier
	REMOVE	Device identifier
	RENAME	Device identifier

6.5.3 ESA Field in NAM Block

USE Contains the address of the expanded string buffer.

INIT N$ESA address

SIZE 1 word

ACCESS

```
$FETCH dst,ESA,reg      ;ESA field to 1-word dst
$STORE src,ESA,reg      ;1-word src to ESA field
$COMPARE src,ESA,reg    ;1-word src with ESA field
```

INPUT

CREATE	Expanded string buffer address
ENTER	Expanded string buffer address
ERASE	Expanded string buffer address
OPEN	Expanded string buffer address
PARSE	Expanded string buffer address
REMOVE	Expanded string buffer address
RENAME	Expanded string buffer address
SEARCH	Expanded string buffer address

6.5.4 ESL Field in NAM Block

USE Contains the length of the expanded string.

INIT None

SIZE 1 byte

ACCESS

```
$FETCH dst,ESL,reg      ;ESL field to 1-byte dst
$COMPARE src,ESL,reg    ;1-byte src with ESL field
```

INPUT

SEARCH	Expanded string length (bytes)

OUTPUT

CREATE	Expanded string length (bytes)
ENTER	Expanded string length (bytes)
ERASE	Expanded string length (bytes)
OPEN	Expanded string length (bytes)
PARSE	Expanded string length (bytes)
REMOVE	Expanded string length (bytes)
RENAME	Expanded string length (bytes)

6.5.5 ESS Field in NAM Block

USE Contains the size of the expanded string buffer.

INIT N$ESS number

SIZE 1 byte

ACCESS

```
$FETCH dst,ESS,reg      ;ESS field to 1-byte dst
$STORE src,ESS,reg      ;1-byte src to ESS field
$COMPARE src,ESS,reg    ;1-byte src with ESS field
```

INPUT

CREATE	Expanded string buffer size (bytes)
ENTER	Expanded string buffer size (bytes)
ERASE	Expanded string buffer size (bytes)
OPEN	Expanded string buffer size (bytes)
PARSE	Expanded string buffer size (bytes)
REMOVE	Expanded string buffer size (bytes)
RENAME	Expanded string buffer size (bytes)

6.5.6 FID Field in NAM Block

USE	Contains the file identifier for the target file.	
INIT	None	
SIZE	3 words	
ACCESS	$FETCH dst,FID,reg	;FID field to 3-word dst
INPUT	ENTER	File identifier
	ERASE	File identifier
	OPEN	File identifier
OUTPUT	CREATE	File identifier
	ERASE	File identifier
	OPEN	File identifier
	REMOVE	File identifier
	RENAME	File identifier
	SEARCH	File identifier

6.5.7 FNB Field in NAM Block

USE Indicates which parts of the merged string were taken from the file string or the default string. (The masks in this section do not include the NB$WCH mask, which has its own description in the next section.)

INIT None

SIZE 1 word

ACCESS

```
$TESTBITS mask,FNB,reg ;Test mask bits in FNB field
$FETCH dst,FNB,reg     ;FNB field to 1-word dst
$COMPARE src,FNB,reg   ;1-word src with FNB field
```

MASKS

NB$DEV	Device in file string or default string
NB$DIR	Directory in file string or default string
NB$NAM	File name in file string or default string
NB$NOD	Node in file string or default string
NB$QUO	Quoted string in file string or default string
NB$TYP	File type in file string or default string
NB$VER	File version in file string or default string
NB$WDI	Wildcard directory in file string or default string
NB$WNA	Wildcard file name in file string or default string
NB$WTY	Wildcard file type in file string or default string
NB$WVE	Wildcard file version in file string or default string

OUTPUT

CREATE	File specification mask
ENTER	File specification mask
ERASE	File specification mask
OPEN	File specification mask
PARSE	File specification mask
REMOVE	File specification mask
RENAME	File specification mask

6.5.8 FNB Field in NAM Block (NB$WCH Mask)

USE Indicates that a valid wildcard context exists. (Masks for the FNB field other than the NB$WCH mask are described in the previous section.)

INIT None

SIZE 1 word

ACCESS

```
$TESTBITS mask,FNB,reg ;Test mask bits in FNB field
$FETCH dst,FNB,reg     ;FNB field to 1-word dst
$COMPARE src,FNB,reg   ;1-word src with FNB field
```

INPUT

ERASE	Wildcard context established
REMOVE	Wildcard context established
RENAME	Wildcard context established
SEARCH	Wildcard context established

OUTPUT

CREATE	Wildcard context established
ENTER	Wildcard context established
OPEN	Wildcard context established
PARSE	Wildcard context established
SEARCH	Wildcard context established

6.5.9 RSA Field in NAM Block

USE	Contains the address of the resultant string buffer.
INIT	N$RSA address
SIZE	1 word
ACCESS	$FETCH dst,RSA,reg ;RSA field to 1-word dst $STORE src,RSA,reg ;1-word src to RSA field $COMPARE src,RSA,reg ;1-word src with RSA field
INPUT	SEARCH Resultant string buffer address

6.5.10 RSL Field in NAM Block

USE	Contains the length of the resultant string.
INIT	None
SIZE	1 byte
ACCESS	`$FETCH dst,RSL,reg ;RSL field to 1-byte dst` `$COMPARE src,RSL,reg ;1-byte src with RSL field`
INPUT	SEARCH Resultant string length (bytes)
OUTPUT	SEARCH Resultant string length (bytes)

6.5.11 RSS Field in NAM Block

USE Contains the size of the resultant string buffer.

INIT N$RSS number

SIZE 1 byte

ACCESS

```
$FETCH dst,RSS,reg      ;RSS field to 1-byte dst
$STORE src,RSS,reg      ;1-byte src to RSS field
$COMPARE src,RSS,reg    ;1-byte src with RSS field
```

INPUT SEARCH Resultant string buffer size (bytes)

6.5.12 WCC Field in NAM Block

USE	Contains wildcard context information.	
INIT	None	
SIZE	1 word	
ACCESS	$FETCH dst,WCC,reg	;WCC field to 1-word dst
	$COMPARE src,WCC,reg	;1-word src with WCC field
INPUT	SEARCH	Wildcard context
OUTPUT	PARSE	Wildcard context
	SEARCH	Wildcard context

6.5.13 WDI Field in NAM Block

USE	Contains wildcard directory context information.	
INIT	None	
SIZE	1 word	
ACCESS	$FETCH dst,WDI,reg	;WDI field to 1-word dst
	$COMPARE src,WDI,reg	;1-word src with WDI field
INPUT	SEARCH	Wildcard directory context
OUTPUT	PARSE	Wildcard directory context
	SEARCH	Wildcard directory context

6.6 PRO BLOCK SUMMARY

This section summarizes the PRO block and its fields. Table 6-6 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-6: PRO Block Summary

Offset	Offset Symbol	Field Size	Description
000	O$COD	1 byte	PRO block identifier
			000003 XB$PRO PRO block identifier code
001	O$BLN	1 byte	PRO block length (bytes)
			000012 XB$PRL PRO block length (bytes)
002	O$NXT	1 word	Next XAB address
004	O$PRG	1 word	Programmer or member portion of file owner code
006	O$PRJ	1 word	Project or group portion of file owner code
010	O$PRO	1 word	File protection code

6.6.1 BLN Field in PRO Block (XB$PRL Code)

USE Contains the length of the PRO block.

INIT None

SIZE 1 byte

ACCESS

```
$FETCH dst,BLN,reg      ;BLN field to 1-byte dst
$COMPARE src,BLN,reg    ;1-byte src with BLN field
```

6.6.2 COD Field in PRO Block (XB$PRO Code)

USE	Contains the identifier for the PRO block.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,COD,reg ;COD field to 1-byte dst $COMPARE src,COD,reg ;1-byte src with COD field

6.6.3 NXT Field in PRO Block

USE Contains the address of the next XAB (ALL, DAT, KEY, PRO, or SUM block) in a chain of XABs.

INIT X$NXT address

SIZE 1 word

ACCESS

```
$FETCH dst,NXT,reg      ;NXT field to 1-word dst
$STORE src,NXT,reg      ;1-word src to NXT field
$COMPARE src,NXT,reg    ;1-word src with NXT field
```

INPUT

CLOSE	Next XAB address
CREATE	Next XAB address
DISPLAY	Next XAB address
ENTER	Next XAB address
ERASE	Next XAB address
EXTEND	Next XAB address
OPEN	Next XAB address
PARSE	Next XAB address
REMOVE	Next XAB address
RENAME	Next XAB address
SEARCH	Next XAB address

6.6.4 PRG Field in PRO Block

USE Contains the member or programmer portion of the file owner code.

INIT X$PRG number

SIZE 1 word

ACCESS

```
$FETCH dst,PRG,reg      ;PRG field to 1-word dst
$STORE src,PRG,reg      ;1-word src to PRG field
$COMPARE src,PRG,reg    ;1-word src with PRG field
```

INPUT	CLOSE	Programmer or member portion of file owner code
OUTPUT	DISPLAY	Programmer or member portion of file owner code
	OPEN	Programmer or member portion of file owner code

6.6.5 PRJ Field in PRO Block

USE	Contains the group or project portion of the file owner code.	
INIT	X$PRJ number	
SIZE	1 word	
ACCESS	$FETCH dst,PRJ,reg	;PRJ field to 1-word dst
	$STORE src,PRJ,reg	;1-word src to PRJ field
	$COMPARE src,PRJ,reg	;1-word src with PRJ field
INPUT	CLOSE	Project or group portion of file owner code
OUTPUT	DISPLAY	Project or group portion of file owner code
	OPEN	Project or group portion of file owner code

6.6.6 PRO Field in PRO Block

USE Contains the protection code for the file.

INIT X$PRO number

SIZE 1 word

ACCESS

```
$FETCH dst,PRO,reg      ;PRO field to 1-word dst
$STORE src,PRO,reg      ;1-word src to PRO field
$COMPARE src,PRO,reg    ;1-word src with PRO field
```

INPUT

CLOSE	File protection code
CREATE	File protection code
DISPLAY	File protection code
OPEN	File protection code

6.7 RAB SUMMARY

This section summarizes the RAB and its fields. Table 6-7 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-7: RAB Summary

Offset	Offset Symbol	Field Size	Description
000	O$BID	1 byte	RAB identifier code
			000001 RB$BID RAB identifier
001	O$BLN	1 byte	RAB length (bytes)
			000120 RB$BLN RAB length (bytes)
002	O$CTX	1 word	User context
004	O$ISI	1 word	Internal stream identifier
006	O$STS	1 word	Completion status code
010	O$STV	1 word	Completion status value
012	O$RFA	3 words	Record file address
020	O$RAC	1 byte	Record access code
			000000 RB$SEQ Sequential access 000001 RB$KEY Key access 000002 RB$RFA RFA access
021	O$KSZ	1 byte	Key size (bytes)
022	O$ROP	1 word	Record processing option mask
			000001 RB$EOF Position to end-of-file 000002 RB$MAS Mass insert 000020 RB$LOA Honor bucket fill numbers 000100 RB$LOC Locate mode 002000 RB$KGE Greater-than-or-equal key criterion 004000 RB$KGT Greater-than key criterion 010000 RB$FDL Fast deletion

(continued on next page)

Table 6-7 (cont.): RAB Summary

Offset	Offset Symbol	Field Size	Description
	020000	RB$UIF	Update if record exists
024	O$USZ	1 word	User buffer size (bytes)
026	O$UBF	1 word	User buffer address
030	O$RSZ	1 word	Record size (bytes)
032	O$RBF	1 word	Record buffer address
034	O$KBF	1 word	Key buffer address
036	O$KRF	1 byte	Key of reference
037	O$MBF	1 byte	Multibuffer count
040	O$MBC	1 byte	Multiblock count
041	O$RT1A	1 byte	Reserved
042	O$RHB	1 word	VFC control buffer address
044	O$FAB	1 word	FAB address
046	O$BKT	2 words	Virtual block number (VBN) or relative record number (RRN)

6.7.1 BID Field in RAB (RB$BID Code)

USE	Contains the identifier for the RAB.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,BID,reg ;BID field to 1-byte dst $COMPARE src,BID,reg ;1-byte src with BID field

6.7.2 BKT Field in RAB

USE Contains a virtual block number or relative record number for a target record.

INIT R$BKT number

SIZE 2 words

ACCESS

```
$FETCH dst,BKT,reg      ;BKT field to 2-word dst
$STORE src,BKT,reg      ;2-word src to BKT field
$FETCH dst,BKTn,reg     ;BKT word n to 1-word dst
$STORE src,BKTn,reg     ;1-word src to BKT word n
$COMPARE src,BKTn,reg   ;1-word src with BKT word n
```

INPUT

READ	Virtual block number (VBN)
SPACE	Virtual block number (VBN) increment
WRITE	Virtual block number (VBN)

OUTPUT

FIND	Relative record number (RRN)
GET	Relative record number (RRN)
PUT	Relative record number (RRN)

6.7.3 BLN Field in RAB

USE	Contains the length of the RAB.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,BLN,reg ;BLN field to 1-byte dst $COMPARE src,BLN,reg ;1-byte src with BLN field
CODES	RB$BLN RAB length (bytes)

6.7.4 CTX Field in RAB

USE	Contains any information you may want to associate with the stream at run time.
INIT	R$CTX number
SIZE	1 word
ACCESS	$FETCH dst,CTX,reg ;CTX field to 1-word dst $STORE src,CTX,reg ;1-word src to CTX field $COMPARE src,CTX,reg ;1-word src with CTX field

6.7.5 FAB Field in RAB

USE Contains the address of the FAB for the target file.

INIT R$FAB address

SIZE 1 word

ACCESS

```
$FETCH dst,FAB,reg      ;FAB field to 1-word dst
$STORE src,FAB,reg      ;1-word src to FAB field
$COMPARE src,FAB,reg    ;1-word src with FAB field
```

INPUT CONNECT FAB address

6.7.6 ISI Field in RAB

USE Contains the internal stream identifier for the target file.

INIT None

SIZE 1 word

ACCESS

```
$FETCH dst,ISI,reg      ;ISI field to 1-word dst
$COMPARE src,ISI,reg    ;1-word src with ISI field
```

INPUT

DELETE	Internal stream identifier
DISCONNECT	Internal stream identifier
FIND	Internal stream identifier
FLUSH	Internal stream identifier
FREE	Internal stream identifier
GET	Internal stream identifier
PUT	Internal stream identifier
READ	Internal stream identifier
REWIND	Internal stream identifier
TRUNCATE	Internal stream identifier
UPDATE	Internal stream identifier
WAIT	Internal stream identifier
WRITE	Internal stream identifier

OUTPUT

CONNECT	Internal stream identifier
DISCONNECT	Internal stream identifier

6.7.7 KBF Field in RAB

USE Contains the address of the key buffer for the target record.

INIT R$KBF address

SIZE 1 word

ACCESS

```
$FETCH dst,KBF,reg      ;KBF field to 1-word dst
$STORE src,KBF,reg      ;1-word src to KBF field
$COMPARE src,KBF,reg    ;1-word src with KBF field
```

INPUT

FIND	Key buffer address
GET	Key buffer address
PUT	Key buffer address

6.7.8 KRF Field in RAB

USE Contains the index reference number of the index for the operation.

INIT R$KRF number

SIZE 1 byte

ACCESS

```
$FETCH dst,KRF,reg      ;KRF field to 1-byte dst
$STORE src,KRF,reg      ;1-byte src to KRF field
$COMPARE src,KRF,reg    ;1-byte src with KRF field
```

INPUT

CONNECT	Key of reference
FIND	Key of reference
GET	Key of reference
REWIND	Key of reference

6.7.9 KSZ Field in RAB

USE Contains the size of the record key for the operation.

INIT R$KSZ number

SIZE 1 byte

ACCESS

```
$FETCH dst,KSZ,reg      ;KSZ field to 1-byte dst
$STORE src,KSZ,reg      ;1-byte src to KSZ field
$COMPARE src,KSZ,reg    ;1-byte src with KSZ field
```

INPUT

```
FIND        Key size (bytes)
GET         Key size (bytes)
PUT         Key size (bytes)
```

6.7.10 MBC Field in RAB

USE Contains the multiblock count for the stream.

INIT R$MBC number

SIZE 1 byte

ACCESS

```
$FETCH dst,MBC,reg      ;MBC field to 1-byte dst
$STORE src,MBC,reg      ;1-byte src to MBC field
$COMPARE src,MBC,reg    ;1-byte src with MBC field
```

INPUT CONNECT Multiblock count

6.7.11 MBF Field in RAB

USE	Contains the multibuffer count for the stream.
INIT	R$MBF number
SIZE	1 byte
ACCESS	$FETCH dst,MBF,reg ;MBF field to 1-byte dst $STORE src,MBF,reg ;1-byte src to MBF field $COMPARE src,MBF,reg ;1-byte src with MBF field
INPUT	CONNECT Multibuffer count

6.7.12 RAC Field in RAB

USE Contains the access mode code for the operation.

INIT R$RAC code

SIZE 1 byte

ACCESS

```
$FETCH dst,RAC,reg      ;RAC field to 1-byte dst
$STORE src,RAC,reg      ;1-byte src to RAC field
$COMPARE src,RAC,reg    ;1-byte src with RAC field
```

CODES

```
RB$KEY  Key access
RB$RFA  RFA access
RB$SEQ  Sequential access
```

INPUT

```
FIND        Record access code
GET         Record access code
PUT         Record access code
```

6.7.13 RBF Field in RAB

USE Contains the address of the record buffer for the operation.

INIT R$RBF address

SIZE 1 word

ACCESS

```
$FETCH dst,RBF,reg      ;RBF field to 1-word dst
$STORE src,RBF,reg      ;1-word src to RBF field
$COMPARE src,RBF,reg    ;1-word src with RBF field
```

INPUT

PUT	Record buffer address
UPDATE	Record buffer address
WRITE	Record buffer address

OUTPUT

CONNECT	Record buffer address
GET	Record buffer address
PUT	Record buffer address
READ	Record buffer address

6.7.14 RFA Field in RAB

USE	Contains the record file address for the target record.	
INIT	None	
SIZE	3 words	
ACCESS	$FETCH dst,RFA,reg ;RFA field to 3-word dst	
INPUT	FIND	Record file address
	GET	Record file address
OUTPUT	CONNECT	End-of-file address
	FIND	Record file address
	GET	Record file address
	PUT	Record file address
	READ	Virtual block number (2 words)
	WRITE	Virtual block number (2 words)

6.7.15 RHB Field in RAB

USE Contains the address of the VFC fixed control area buffer for the target record.

INIT R$RHB address

SIZE 1 word

ACCESS

```
$FETCH dst,RHB,reg      ;RHB field to 1-word dst
$STORE src,RHB,reg      ;1-word src to RHB field
$COMPARE src,RHB,reg    ;1-word src with RHB field
```

INPUT

```
GET         VFC control buffer address
PUT         VFC control buffer address
UPDATE      VFC control buffer address
```

6.7.16 ROP Field in RAB (RB$EOF Mask)

USE	Requests initial stream context at end-of-file.
INIT	R$ROP mask
SIZE	1 word
ACCESS	$SET mask,ROP,reg ;Mask bits on in ROP field $OFF mask,ROP,reg ;Mask bits off in ROP field $TESTBITS mask,ROP,reg ;Test mask bits in ROP field $FETCH dst,ROP,reg ;ROP field to 1-word dst $STORE src,ROP,reg ;1-word src to ROP field $COMPARE src,ROP,reg ;1-word src with ROP field
INPUT	CONNECT Position to end-of-file

6.7.17 ROP Field in RAB (RB$FDL Mask)

USE Requests fast deletion.

INIT R$ROP mask

SIZE 1 word

ACCESS
```
$SET mask,ROP,reg        ;Mask bits on in ROP field
$OFF mask,ROP,reg        ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg ;Test mask bits in ROP field
$FETCH dst,ROP,reg       ;ROP field to 1-word dst
$STORE src,ROP,reg       ;1-word src to ROP field
$COMPARE src,ROP,reg     ;1-word src with ROP field
```

INPUT DELETE Fast deletion

6.7.18 ROP Field in RAB (RB$KGE Mask)

USE Requests greater-than-or-equal key match criterion.

INIT R$ROP mask

SIZE 1 word

ACCESS

```
$SET mask,ROP,reg       ;Mask bits on in ROP field
$OFF mask,ROP,reg       ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg  ;Test mask bits in ROP field
$FETCH dst,ROP,reg      ;ROP field to 1-word dst
$STORE src,ROP,reg      ;1-word src to ROP field
$COMPARE src,ROP,reg    ;1-word src with ROP field
```

INPUT

FIND	Greater-than-or-equal key criterion
GET	Greater-than-or-equal key criterion

6.7.19 ROP Field in RAB (RB$KGT Mask)

USE — Requests greater-then key match criterion.

INIT — R$ROP mask

SIZE — 1 word

ACCESS

```
$SET mask,ROP,reg        ;Mask bits on in ROP field
$OFF mask,ROP,reg        ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg   ;Test mask bits in ROP field
$FETCH dst,ROP,reg       ;ROP field to 1-word dst
$STORE src,ROP,reg       ;1-word src to ROP field
$COMPARE src,ROP,reg     ;1-word src with ROP field
```

INPUT

FIND	Greater-than key criterion
GET	Greater-than key criterion

6.7.20 ROP Field in RAB (RB$LOA Mask)

USE Requests bucket fill number honoring.

INIT R$ROP mask

SIZE 1 word

ACCESS
```
$SET mask,ROP,reg       ;Mask bits on in ROP field
$OFF mask,ROP,reg       ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg  ;Test mask bits in ROP field
$FETCH dst,ROP,reg      ;ROP field to 1-word dst
$STORE src,ROP,reg      ;1-word src to ROP field
$COMPARE src,ROP,reg    ;1-word src with ROP field
```

INPUT
```
PUT         Honor bucket fill numbers
UPDATE      Honor bucket fill numbers
```

6.7.21 ROP Field in RAB (RB$LOC Mask)

USE Requests locate mode operation.

INIT R$ROP mask

SIZE 1 word

ACCESS

```
$SET mask,ROP,reg       ;Mask bits on in ROP field
$OFF mask,ROP,reg       ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg  ;Test mask bits in ROP field
$FETCH dst,ROP,reg      ;ROP field to 1-word dst
$STORE src,ROP,reg      ;1-word src to ROP field
$COMPARE src,ROP,reg    ;1-word src with ROP field
```

INPUT

CONNECT	Locate mode
GET	Locate mode
PUT	Locate mode

6.7.22 ROP Field in RAB (RB$MAS Mask)

USE Requests mass insertion.

INIT R$ROP mask

SIZE 1 word

ACCESS
```
$SET mask,ROP,reg       ;Mask bits on in ROP field
$OFF mask,ROP,reg       ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg  ;Test mask bits in ROP field
$FETCH dst,ROP,reg      ;ROP field to 1-word dst
$STORE src,ROP,reg      ;1-word src to ROP field
$COMPARE src,ROP,reg    ;1-word src with ROP field
```

INPUT PUT Mass insert

6.7.23 ROP Field in RAB (RB$UIF Mask)

USE Requests update if target record already exists.

INIT R$ROP mask

SIZE 1 word

ACCESS
```
$SET mask,ROP,reg       ;Mask bits on in ROP field
$OFF mask,ROP,reg       ;Mask bits off in ROP field
$TESTBITS mask,ROP,reg  ;Test mask bits in ROP field
$FETCH dst,ROP,reg      ;ROP field to 1-word dst
$STORE src,ROP,reg      ;1-word src to ROP field
$COMPARE src,ROP,reg    ;1-word src with ROP field
```

INPUT PUT Update if record exists

6.7.24 RSZ Field in RAB

USE	Contains the size of the target record.	
INIT	R$RSZ number	
SIZE	1 word	
ACCESS	$FETCH dst,RSZ,reg	;RSZ field to 1-word dst
	$STORE src,RSZ,reg	;1-word src to RSZ field
	$COMPARE src,RSZ,reg	;1-word src with RSZ field
INPUT	PUT	Record size (bytes)
	UPDATE	Record size (bytes)
	WRITE	Record size (bytes)
OUTPUT	GET	Record size (bytes)
	READ	Record size (bytes)

6.7.25 STS Field in RAB

USE Contains the completion status code for the operation.

INIT None

SIZE 1 word

ACCESS

```
$FETCH dst,STS,reg      ;STS field to 1-word dst
$COMPARE src,STS,reg    ;1-word src with STS field
```

OUTPUT

CONNECT	Completion status code
DELETE	Completion status code
DISCONNECT	Completion status code
FIND	Completion status code
FLUSH	Completion status code
FREE	Completion status code
GET	Completion status code
PUT	Completion status code
READ	Completion status code
REWIND	Completion status code
TRUNCATE	Completion status code
UPDATE	Completion status code
WRITE	Completion status code

6.7.26 STV Field in RAB

USE Contains the completion status value for the operation.

INIT None

SIZE 1 word

ACCESS

```
$FETCH dst,STV,reg      ;STV field to 1-word dst
$COMPARE src,STV,reg    ;1-word src with STV field
```

OUTPUT

CONNECT	Completion status value
DELETE	Completion status value
DISCONNECT	Completion status value
FIND	Completion status value
FLUSH	Completion status value
FREE	Completion status value
GET	Completion status value
PUT	Completion status value
READ	Completion status value
REWIND	Completion status value
TRUNCATE	Completion status value
UPDATE	Completion status value
WRITE	Completion status value

6.7.27 UBF Field in RAB

USE Contains the address of the user buffer for the operation.

INIT R$UBF address

SIZE 1 word

ACCESS
```
$FETCH dst,UBF,reg      ;UBF field to 1-word dst
$STORE src,UBF,reg      ;1-word src to UBF field
$COMPARE src,UBF,reg    ;1-word src with UBF field
```

INPUT

CONNECT	User buffer address
GET	User buffer address
PUT	User buffer address
READ	User buffer address

6.7.28 USZ Field in RAB

USE	Contains the size of the user buffer for the operation.
INIT	R$USZ number
SIZE	1 word
ACCESS	$FETCH dst,USZ,reg ;USZ field to 1-word dst $STORE src,USZ,reg ;1-word src to USZ field $COMPARE src,USZ,reg ;1-word src with USZ field
INPUT	CONNECT User buffer size (bytes) GET User buffer size (bytes) PUT User buffer size (bytes) READ User buffer size (bytes)

6.8 SUM BLOCK SUMMARY

This section summarizes the SUM block and its fields. Table 6-8 summarizes the entire block, giving the offset, offset symbol, size, and a brief description for each field; for a field that has mask or code symbols, the table also gives the value, symbol, and a brief description for each mask or code.

Table 6-8: SUM Block Summary

Offset	Offset Symbol	Field Size	Description
000	O$COD	1 byte	SUM block identifier
			000005 XB$SUM SUM block identifier code
001	O$BLN	1 byte	SUM block length (bytes)
			000012 XB$SML SUM block length (bytes)
002	O$NXT	1 word	Next XAB address
004	O$NOK	1 byte	Number of indexes
005	O$NOA	1 byte	Number of areas
006	O$NOR	1 byte	Reserved
007		1 byte	Reserved
010	O$PVN	1 word	Prologue version number

6.8.1 BLN Field in SUM Block (XB$SML Code)

USE	Contains the length of the SUM block.	
INIT	None	
SIZE	1 byte	
ACCESS	$FETCH dst,BLN,reg	;BLN field to 1-byte dst
	$COMPARE src,BLN,reg	;1-byte src with BLN field

6.8.2 COD Field in SUM Block (XB$SUM Code)

USE	Contains the identifier for the SUM block.
INIT	None
SIZE	1 byte
ACCESS	$FETCH dst,COD,reg ;COD field to 1-byte dst $COMPARE src,COD,reg ;1-byte src with COD field

6.8.3 NOA Field in SUM Block

USE	Contains the number of areas in the file.
INIT	None
SIZE	1 byte
ACCESS	`$FETCH dst,NOA,reg ;NOA field to 1-byte dst` `$COMPARE src,NOA,reg ;1-byte src with NOA field`
OUTPUT	DISPLAY Number of areas OPEN Number of areas

6.8.4 NOK Field in SUM Block

USE Contains the number of indexes in the file.

INIT None

SIZE 1 byte

ACCESS
```
$FETCH dst,NOK,reg      ;NOK field to 1-byte dst
$COMPARE src,NOK,reg    ;1-byte src with NOK field
```

OUTPUT
```
DISPLAY     Number of indexes
OPEN        Number of indexes
```

6.8.5 NXT Field in SUM Block

USE Contains the address of the next XAB (ALL, DAT, KEY, PRO, or SUM block) in a chain of XABs.

INIT X$NXT address

SIZE 1 word

ACCESS

```
$FETCH dst,NXT,reg      ;NXT field to 1-word dst
$STORE src,NXT,reg      ;1-word src to NXT field
$COMPARE src,NXT,reg    ;1-word src with NXT field
```

INPUT

CLOSE	Next XAB address
CREATE	Next XAB address
DISPLAY	Next XAB address
ENTER	Next XAB address
ERASE	Next XAB address
EXTEND	Next XAB address
OPEN	Next XAB address
PARSE	Next XAB address
REMOVE	Next XAB address
RENAME	Next XAB address
SEARCH	Next XAB address

6.8.6 PVN Field in SUM Block

USE	Contains the prologue version number for the file.
INIT	None
SIZE	1 word
ACCESS	$FETCH dst,PVN,reg ;PVN field to 1-word dst $COMPARE src,PVN,reg ;1-word src with PVN field
OUTPUT	DISPLAY Prologue version number OPEN Prologue version number

CHAPTER 7

EXAMPLE PROGRAMS

This chapter contains example programs; the titles of the programs are:

- PARSE - $PARSE TEST
- SEARCH - $SEARCH TEST
- ERASE - $ERASE TEST
- RENAME - $RENAME TEST
- OPEN1 - $OPEN BY NAME/FID TEST
- OPEN2 - $OPEN BY FID WITH WILDCARDS TEST
- OPEN3 - $OPEN WITH IMPLICIT WILDCARDS (ILLEGAL)
- GSA - CORE SPACE ALLOCATOR

Example 7-1: PARSE - $PARSE Test

```
        .TITLE  PARSE - $PARSE TEST
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $PARSE,$STORE,$FETCH,$COMPARE
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $PARSE function.
;

;
; RMS-11 Data Structures
;

        GSA$    GSA

FAB::   FAB$B                   ; Argument FAB
         F$NAM  NAM             ; Link to NAM
         F$LCH  2.              ; Channel #2
        FAB$E

NAM::   NAM$B                   ; NAM definition
         N$ESA  EXPSTR          ; EXP STR address
         N$ESS  128.            ; EXP STR length
        NAM$E


        .PSECT  $CODE$,RO,I

PARSE::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB,R0         ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #NAM,R3         ; Map the target NAM
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES1,Q.IOPL(R5)
        MOV     #QUES1L,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the DNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF1,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,DNS,R0    ; Set the default length
        $STORE  #BUFF1,DNA,R0   ; Set the default address
        MOV     #QUES2,Q.IOPL(R5)
        MOV     #QUES2L,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the DNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF2,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
```

```
        $STORE  IOLEN,FNS,R0    ; Set the default length
        $STORE  #BUFF2,FNA,R0   ; Set the default address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
        $PARSE  R0              ; Parse the strings
        $COMPARE #0,STS,R0      ; An error?
        BLT     ERROR           ; Yes if MI; display it
        CLR     (R2)            ; Init the length
        $FETCH  (R2),ESL,R3     ; Get the string length
        TST     (R2)+           ; Advance
        $FETCH  (R2)+,ESA,R3    ; Get the string address
        $FETCH  (R2),FNB,R3     ; Get the file name bits
        MOV     #ESSSTR,R1      ; Select the format string
        CALL    PRINT           ; Display the file
        CALL    BITS            ; Do the FNB bit disply
        BR      PARSE           ; And let's try another

EXIT:
        EXIT$S                  ; Task exit

ERROR:
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
        CALL    PRINT           ; Go edit and print the message
        BR      PARSE           ; Let's try this again

BITS:
        MOV     #EDBLK,R2       ; Init EDBLK address
        $FETCH  R0,FNB,R3       ; Get the FNB bits
        BIT     #2000,R0        ; Quoted string?
        BEQ     2$              ; No if EQ
        MOV     #QUO,(R2)+      ; Set Quoted string
2$:     BIT     #1000,R0        ; Wild directory?
        BEQ     4$              ; No if EQ
        MOV     #WDI,(R2)+      ; Set wild directory
4$:     BIT     #400,R0         ; Node spec?
        BEQ     6$              ; No if EQ
        MOV     #NOD,(R2)+      ; Set nodespec
6$:     BIT     #100,R0         ; Directory spec?
        BEQ     8$              ; No if EQ
        MOV     #DIR,(R2)+      ; Set directory
8$:     BIT     #40,R0          ; Wild name?
        BEQ     10$             ; No if eq
        MOV     #WNA,(R2)+      ; Set wild name
10$:    BIT     #20,R0          ; Wild type?
        BEQ     12$             ; No if EQ
        MOV     #WTY,(R2)+      ; Set wild type
12$:    BIT     #10,R0          ; Wild version?
        BEQ     14$             ; No if EQ
        MOV     #WVE,(R2)+      ; Set wild version
14$:    BIT     #4,R0           ; Name?
        BEQ     16$             ; No if EQ
        MOV     #NME,(R2)+      ; Set name
16$:    BIT     #2,R0           ; Type?
        BEQ     18$             ; No if EQ
        MOV     #TYP,(R2)+      ; Set type
18$:    BIT     #1,R0           ; Version?
        BEQ     20$             ; No if EQ
        MOV     #VER,(R2)+      ; Set version
20$:    MOV     #END,(R2)       ; End with a null...
        MOV     #DEV,R1         ; Set the default (dev)
        CALL    PRINT           ; Edit and print
        RETURN                  ; And exit
PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
```

```
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,Q.IOPL(R5)
        MOV     R1,Q.IOPL+2(R5)
        DIR$    R5              ; Send to the terminal
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

QUES1:  .Ascii  <15><12>"Enter the default name string: "
        QUES1L = . - QUES1
QUES2:  .Ascii  <15><12>"Enter the primary name string: "
        QUES2L = . - QUES2
ERRSTR: .Asciz  "$PARSE error -- STS=%P, STV=%P"
ESSSTR: .Ascii  "$PARSE expanded string is %VA%N"
        .Asciz  "       File name bits (FNB) are %P"
DEV:    .Asciz  "       (DEV%I"
NOD:    .Asciz  ", NOD%I"
DIR:    .Asciz  ", DIR%I"
NME:    .Asciz  ", NAM%I"
QUO:    .Asciz  ", QUO%I"
TYP:    .Asciz  ", TYP%I"
VER:    .Asciz  ", VER%I"
WDI:    .Asciz  ", WDI%I"
WNA:    .Asciz  ", WNA%I"
WTY:    .Asciz  ", WTY%I"
WVE:    .Asciz  ", WVE%I"
END:    .Asciz  ")"

        .EVEN
EDBLK:  .BLKW   16.
BUFFER:
BUFF1:  .Blkb   64.
BUFF2:  .Blkb   64
EXPSTR: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    PARSE
```

Example 7-2: SEARCH - $SEARCH Test

```
        .TITLE  SEARCH - $SEARCH TEST
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $PARSE,$SEARCH,$STORE,$FETCH,$COMPARE
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $SEARCH function.
;

;
; RMS-11 Data Structures
;

        GSA$    GSA

FAB::   FAB$B                   ; Argument FAB
         F$DNA  SYDSKA          ; If no device, SY:
         F$DNS  SYDSKL          ; should be used
         F$NAM  NAM             ; Link to NAM
         F$LCH  2.              ; Channel #2
        FAB$E

NAM::   NAM$B                   ; NAM definition
         N$ESA  EXPSTR          ; EXP STR address
         N$ESS  128.            ; EXP STR length
         N$RSA  RESSTR          ; RES STR address
         N$RSS  128.            ; RES STR length
        NAM$E


        .PSECT  $CODE$,RO,I

SEARCH::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB,R0         ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #NAM,R3         ; Map the target NAM
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES,Q.IOPL(R5)
        MOV     #QUESL,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the DNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,FNS,R0    ; Set the string length
        $STORE  #BUFF,FNA,R0    ; Set the string address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
        CLR     FILCNT          ; Init count of matches
        $PARSE  R0              ; Parse the strings
        $COMPARE #0,STS,R0      ; An error?
        BLT     ERROR           ; Yes if MI; display it
GETFIL:
```

```
        MOV     #EDBLK,R2       ; Reset the edit block addr
        $SEARCH R0              ; Get a matching file
        $COMPARE #0,STS,R0      ; Error?
        BLT     ERROR           ; Yes if LT
        CLR     (R2)            ; Init the length
        $FETCH  (R2),RSL,R3     ; Get the string length
        TST     (R2)+           ; Advance
        $FETCH  (R2)+,RSA,R3    ; Get the string address
        MOV     #RSSSTR,R1      ; Select the format string
        TST     FILCNT          ; First file needs a blank
        BNE     NOTFST          ; line before it
        MOV     #RSSST1,R1      ; Insert CR/LF first
NOTFST:
        CALL    PRINT           ; Display the file
        MOV     #FAB,R0         ; Rest the FAB address
        INC     FILCNT          ; Count this file
        BR      GETFIL          ; And let's try another

EXIT:
        EXIT$S                  ; Task exit

ERROR:
        $COMPARE #ER$NMF,STS,R0 ; No more matches?
        BNE     ERROR0          ; No - some other error
        MOV     FILCNT,(R2)+    ; Set the cound of matches
        BEQ     ERROR2          ; No files...
        CLR     (R2)            ; Give the ESA
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #TTLSTR,R1      ; Set the format string
        BR      ERROR1          ; Go show it and exit
ERROR2:
        CLR     -(R2)           ; Setup for string length
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance to next word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #NOFILE,R1      ; Set the format string
        BR      ERROR1          ; Print the error
ERROR0:
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
ERROR1:
        CALL    PRINT           ; Go edit and print the message
        JMP     SEARCH          ; Let's try this again

PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,Q.IOPL(R5)
        MOV     R1,Q.IOPL+2(R5)
        DIR$    R5              ; Send to the terminal
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

SYDSKA: .Ascii  "SY:"
        SYDSKL = . - SYDSKA
QUES:   .Ascii  <15><12>"Enter a wildcard filespec: "
        QUESL = . - QUES
ERRSTR: .Asciz  "$SEARCH error -- STS=%P, STV=%P"
RSSST1: .Ascii  "%N"
RSSSTR: .Asciz  "  %VA"
```

```
NOFILE: .Asciz  "%NNo files matching %VA%N"
TTLSTR: .Asciz  "%NTotal of %D files matching %VA%N"
        .EVEN
FILCNT: .WORD   0
EDBLK:  .BLKW   6
BUFFER:
BUFF:   .Blkb   128.
EXPSTR: .BLKB   128.
RESSTR: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    SEARCH
```

Example 7-3: ERASE - $ERASE Test

```
        .TITLE  ERASE - $ERASE TEST
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $PARSE,$ERASE,$STORE,$FETCH,$COMPARE
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $ERASE function, with implicit $SEARCH.
;

;
; RMS-11 Data Structures
;

        GSA$    GSA

FAB::   FAB$B                   ; Argument FAB
         F$DNA  SYDSKA          ; If no device, SY:
         F$DNS  SYDSKL          ; should be used
         F$NAM  NAM             ; Link to NAM
         F$LCH  2.              ; Channel #2
        FAB$E

NAM::   NAM$B                   ; NAM definition
         N$ESA  EXPSTR          ; EXP STR address
         N$ESS  128.            ; EXP STR length
         N$RSA  RESSTR          ; RES STR address
         N$RSS  128.            ; RES STR length
        NAM$E


        .PSECT  $CODE$,RO,I

ERASE::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB,R0         ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #NAM,R3         ; Map the target NAM
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES,Q.IOPL(R5)
        MOV     #QUESL,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the DNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,FNS,R0    ; Set the string length
        $STORE  #BUFF,FNA,R0    ; Set the string address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
        CLR     FILCNT          ; Init count of matches
        $PARSE  R0              ; Parse the strings
        $COMPARE #0,STS,R0      ; An error?
        BLT     ERROR           ; Yes if MI; display it
GETFIL:
```

```
        MOV     #EDBLK,R2       ; Reset the edit block addr
        $ERASE  R0              ; Issue implicit $ERASE
        $COMPARE #0,STS,R0      ; Error?
        BLT     ERROR           ; Yes if LT
        CLR     (R2)            ; Init the length
        $FETCH  (R2),RSL,R3     ; Get the string length
        TST     (R2)+           ; Advance
        $FETCH  (R2)+,RSA,R3    ; Get the string address
        MOV     #RSSSTR,R1      ; Select the format string
        TST     FILCNT          ; First file needs a blank
        BNE     NOTFST          ; line before it
        MOV     #RSSST1,R1      ; Insert CR/LF first
NOTFST:
        CALL    PRINT           ; Display the file
        MOV     #FAB,R0         ; Rest the FAB address
        INC     FILCNT          ; Count this file
        BR      GETFIL          ; And let's try another

EXIT:
        EXIT$S                  ; Task exit

ERROR:
        $COMPARE #ER$NMF,STS,R0 ; No more matches?
        BNE     ERROR0          ; No - some other error
        MOV     FILCNT,(R2)+    ; Set the cound of matches
        BEQ     ERROR2          ; No files...
        CLR     (R2)            ; Give the ESA
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #TTLSTR,R1      ; Set the format string
        BR      ERROR1          ; Go show it and exit
ERROR2:
        CLR     -(R2)           ; Setup for string length
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance to next word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #NOFILE,R1      ; Set the format string
        BR      ERROR1          ; Print the error
ERROR0:
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
ERROR1:
        CALL    PRINT           ; Go edit and print the message
        JMP     ERASE           ; Let's try this again

PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,Q.IOPL(R5)
        MOV     R1,Q.IOPL+2(R5)
        DIR$    R5              ; Send to the terminal
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

SYDSKA: .Ascii  "SY:"
        SYDSKL = . - SYDSKA
QUES:   .Ascii  <15><12>"File(s) to erase: "
        QUESL = . - QUES
ERRSTR: .Asciz  "$ERASE error -- STS=%P, STV=%P"
RSSST1: .Ascii  "%N"
RSSSTR: .Asciz  "  File %VA deleted"
```

```
NOFILE: .Asciz  "%NNo files matching %VA%N"
TTLSTR: .Asciz  "%NTotal of %D files matching %VA deleted%N"
        .EVEN
FILCNT: .WORD   0
EDBLK:  .BLKW   6
BUFFER:
BUFF:   .Blkb   128.
EXPSTR: .BLKB   128.
RESSTR: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    ERASE
```

Example 7-4: RENAME - $RENAME Test

```
        .TITLE  RENAME - $RENAME TEST
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $PARSE,$SEARCH,$RENAME,$STORE,$FETCH,$COMPARE
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $RENAME function.
;

;
; RMS-11 Data Structures
;

        GSA$    GSA

FAB1::  FAB$B                   ; Old file name
         F$DNA  SYDSKA          ; Default to SY:
         F$DNS  SYDSKL
         F$NAM  NAM1            ; Link to NAM1
         F$LCH  2.              ; Channel #2
         F$FOP  FB$FID          ; Turn on NAM usage
        FAB$E

NAM1::  NAM$B                   ; NAM definition
         N$ESA  ESSTR1          ; EXP STR address
         N$ESS  128.            ; EXP STR length
         N$RSA  RSSTR1          ; RES STR address
         N$RSS  128.            ; RES STR length
        NAM$E


FAB2::  FAB$B                   ; New file name
         F$NAM  NAM2            ; Link to NAM2
         F$LCH  2.              ; Same channel
        FAB$E

NAM2::  NAM$B                   ; NAM definition
         N$ESA  ESSTR2          ; EXP STR address
         N$ESS  128.            ; EXP STR length
        NAM$E


        .PSECT  $CODE$,RO,I

RENAME::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB1,R0        ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES1,Q.IOPL(R5)
        MOV     #QUES1L,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the DNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF1,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
```

```
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,DNS,R0    ; Set the default length
        $STORE  #BUFF1,DNA,R0   ; Set the default address
        $PARSE  R0              ; Parse the input spec
        $COMPARE #0,STS,R0      ; An error?
        BLT     ERROR           ; Yes if LT
        MOV     #FAB2,R0        ; Map the 2d FAB
        MOV     #QUES2,Q.IOPL(R5)
        MOV     #QUES2L,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the new name
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF2,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,FNS,R0    ; Set the default length
        $STORE  #BUFF2,FNA,R0   ; Set the default address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
        CLR     FILCNT          ; Initialize file count
        BR      LOOP            ; Enter the RENAME loop

EXIT:
        EXIT$S                  ; Task exit

LOOP:
        MOV     #FAB1,R0        ; Get the input FAB
        MOV     #FAB2,R1        ; And the output FAB
        MOV     #NAM1,R2        ; Setup NAM references
        MOV     #NAM2,R3        ;
        $SEARCH R0              ; Attempt to find a file
        $COMPARE #0,STS,R0      ; Error?
        BLT     SEAERR          ; Yes if LT
        $FETCH  R4,RSA,R2       ; Get the resultant address
        $STORE  R4,DNA,R1       ; Set this as default
        $FETCH  R4,RSL,R2       ; Get the resultant length
        $STORE  R4,DNS,R1       ; Set the default length
        $RENAME R0,,,R1         ; Rename input as output
        $COMPARE #0,STS,R0      ; Error?
        BLT     ERROR           ; Yes if LT- investigate
        MOV     #EDBLK,R0       ; Setup to show the rename
        CLR     (R0)            ;
        $FETCH  (R0),RSL,R2     ; Set the length
        TST     (R0)+           ; Advance to next word
        $FETCH  (R0)+,RSA,R2    ; Set the address
        CLR     (R0)            ;
        $FETCH  (R0),ESL,R3     ; Set the length
        TST     (R0)+           ; Advance to next word
        $FETCH  (R0),ESA,R3     ; Set te address
        MOV     #RENMSG,R1      ; Format string
        CALL    PRINT           ; Display it
        INC     FILCNT          ; Count the file
        BR      LOOP            ; And try another file

ERROR:
        MOV     #EDBLK,R2       ; Map the edit block
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
        CALL    PRINT           ; Go edit and print the message
        JMP     RENAME          ; Let's try this again

SEAERR:
```

```
        $COMPARE #ER$NMF,STS,R0 ; End of wild card search?
        BNE     ERROR           ; No if NE- show why
        MOV     #EDBLK,R0       ; Map the edit block
        TST     FILCNT          ; Any files?
        BNE     TOTAL           ; Yes if NE, show total
        MOV     #NOFILE,R1      ; Show the total
SETES:
        CLR     (R0)            ;
        $FETCH  (R0),ESL,R2     ; Set the length
        TST     (R0)+           ; Advance
        $FETCH  (R0)+,ESA,R2    ; Get the ESA address
        CALL    PRINT           ;
        JMP     RENAME          ; Repeat
TOTAL:
        MOV     FILCNT,(R0)+    ; Set the rename count
        MOV     #TTLMSG,R1      ; Set the format string
        BR      SETES           ; Add ESA and print

PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,WRITE+Q.IOPL
        MOV     R1,WRITE+Q.IOPL+2
        DIR$    #WRITE          ; Send to the terminal
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

SYDSKA: .Ascii  "SY:"
        SYDSKL = . - SYDSKA
QUES1:  .Ascii  <15><12>"From:  "
        QUES1L = . - QUES1
QUES2:  .Ascii  <15><12>"To:    "
        QUES2L = . - QUES2
ERRSTR: .Asciz  "$RENAME error -- STS=%P, STV=%P"
RENMSG: .Asciz  "  File %VA renamed to %VA"
TTLMSG: .Asciz  "%NTotal of %D files matching %VA renamed%N"
NOFILE: .Asciz  "%NNo files matching %VA%N"

        .EVEN
FILCNT: .WORD   0
EDBLK:  .BLKW   6
BUFFER:
BUFF1:  .BLKB   64.
BUFF2:  .BLKB   64
ESSTR1: .BLKB   128.
ESSTR2: .BLKB   128.
RSSTR1: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    RENAME
```

Example 7-5: OPEN1 - $OPEN by Name/FID Test

```
        .TITLE  OPEN1 - $OPEN BY NAME/FID TEST
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $OPEN,$CLOSE,$STORE,$FETCH,$COMPARE,$SET,$OFF,ORG$
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $OPEN by name and FID functions.
;

;
; RMS-11 Data Structures
;

        ORG$    SEQ             ; $OPEN sequential
        ORG$    REL             ; $OPEN relative
        ORG$    IDX             ; $OPEN indexed

        GSA$    GSA

FAB::   FAB$B                   ; Argument FAB
         F$DNA  SYDSKA          ; If no device, SY:
         F$DNS  SYDSKL          ; should be used
         F$NAM  NAM             ; Link to NAM
         F$LCH  2.              ; Channel #2
        FAB$E

NAM::   NAM$B                   ; NAM definition
         N$ESA  EXPSTR          ; EXP STR address
         N$ESS  128.            ; EXP STR length
        NAM$E


        .PSECT  $CODE$,RO,I

OPEN1::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB,R0         ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #NAM,R3         ; Map the target NAM
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES,Q.IOPL(R5)
        MOV     #QUESL,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the FNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,FNS,R0    ; Set the string length
        $STORE  #BUFF,FNA,R0    ; Set the string address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
        MOV     #EDBLK,R2       ; Reset the edit block addr
        $OFF    #FB$BID,FOP,R0  ; Turn off FID bit
        $OPEN   R0              ; Open the file
```

```
        $COMPARE #0,STS,R0      ; Error?
        BLT     ERROR           ; Yes if LT
        CLR     (R2)            ; Init the length
        $FETCH  (R2),ESL,R3     ; Get the string length
        TST     (R2)+           ; Advance
        $FETCH  (R2)+,ESA,R3    ; Get the string address
        MOV     #ESSSTR,R1      ; Select the format string
        CALL    PRINT           ; Display the file's ES
        MOV     #2,(R2)+        ; Setup to show NAM info
        MOV     #NAM+O$DVI,(R2)+
        MOV     O$DVI+2(R3),(R2)+ ; Show the unit field
        MOV     O$FID(R3),(R2)+ ; Show the 3 FID words
        MOV     O$FID+2(R3),(R2)+
        MOV     O$FID+4(R3),(R2)+
        MOV     #NAMSTR,R1      ; Set the format string
        CALL    PRINT           ; Display it
        MOV     #FAB,R0         ; Reset for close
        $CLOSE  R0              ; And close off the file
        $SET    #FB$FID,FOP,R0  ; Turn on OPEN by FID
        $STORE  #0,FNA,R0       ; Remove the file name
        $STORE  #0,FNS,R0       ; Information
        $OPEN   R0              ; Attempt the open
        $COMPARE #0,STS,R0      ; Status OK?
        BLT     ERROR           ; No if LE; show error
        MOV     #FIDSTR,R1      ; Show that the open worked
        CALL    PRINT           ; Display the text
        MOV     #FAB,R0         ; Restore the FAB address
        $CLOSE  R0              ; (We assume no $CLOSE errors)
        JMP     OPEN1           ; Do this again

EXIT:
        EXIT$S                  ; Task exit

ERROR:
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
ERROR1:
        CALL    PRINT           ; Go edit and print the message
        JMP     OPEN1           ; Let's try this again

PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,Q.IOPL(R5)
        MOV     R1,Q.IOPL+2(R5)
        DIR$    R5              ; Send to the terminal
        MOV     #EDBLK,R2       ; Restore the edit block ptr
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

SYDSKA: .Ascii  "SY:"
        SYDSKL = . - SYDSKA
QUES:   .Ascii  <15><12>"File to $OPEN: "
        QUESL = . - QUES
ERRSTR: .Asciz  "%N$OPEN error -- STS=%P, STV=%P%N"
ESSSTR: .Asciz  "%N  File %VA successfully $OPENed"
NAMSTR: .Ascii  "  NAM data returned: DVI='%VA' DVI+2=%P "
        .Asciz  "FID=(%P,%P,%P)%N"
FIDSTR: .Asciz  "  File successfully $OPENed by FID%N"

        .EVEN
EDBLK:  .BLKW   10.
```

```
BUFFER:
BUFF:   .Blkb   128.
EXPSTR: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    OPEN1
```

Example 7-6: OPEN2 - $OPEN by FID with Wildcards Test

```
        .TITLE  OPEN2 - $OPEN BY FID WITH WILDCARDS TEST
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $PARSE,$SEARCH,$OPEN,$CLOSE
        .MCALL  $STORE,$FETCH,$COMPARE,$SET,$OFF,ORG$
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $PARSE, $SEARCH, $OPEN by FID functions.
;

;
; RMS-11 Data Structures
;

        ORG$    SEQ             ; $OPEN sequential
        ORG$    REL             ; $OPEN relative
        ORG$    IDX             ; $OPEN indexed

        GSA$    GSA

FAB::   FAB$B                   ; Argument FAB
         F$DNA  SYDSKA          ; If no device, SY:
         F$DNS  SYDSKL          ; should be used
         F$NAM  NAM             ; Link to NAM
         F$LCH  2.              ; Channel #2
        FAB$E

NAM::   NAM$B                   ; NAM definition
         N$ESA  EXPSTR          ; EXP STR address
         N$ESS  128.            ; EXP STR length
         N$RSA  RESSTR          ; RES STR address
         N$RSS  128.            ; RES STR length
        NAM$E


        .PSECT  $CODE$,RO,I

OPEN2::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB,R0         ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #NAM,R3         ; Map the target NAM
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES,Q.IOPL(R5)
        MOV     #QUESL,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the FNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,FNS,R0    ; Set the string length
        $STORE  #BUFF,FNA,R0    ; Set the string address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
```

```
        CLR     FILCNT          ; Init count of matches
        $SET    #FB$FID,FOP,R0  ; Turn on FID bit
        $PARSE  R0              ; Parse the strings
        $COMPARE #0,STS,R0      ; An error?
        BLT     ERROR           ; Yes if MI; display it
        CLR     FILCNT          ; Clear the match count
GETFIL:
        MOV     #EDBLK,R2       ; Reset the edit block addr
        MOV     #FAB,R0         ; Reset the FAB address
        $SEARCH R0              ; Get a matching file
        $COMPARE #0,STS,R0      ; Error?
        BLT     SEAERR          ; Yes if LT
        CLR     (R2)            ; Init the length
        $FETCH  (R2),RSL,R3     ; Get the string length
        TST     (R2)+           ; Advance
        $FETCH  (R2)+,RSA,R3    ; Get the string address
        MOV     #RSSSTR,R1      ; Select the format string
        CALL    PRINT           ; Display the file's RS
        MOV     #2,(R2)+        ; Setup to show NAM info
        MOV     #NAM+O$DVI,(R2)+
        MOV     O$DVI+2(R3),(R2)+ ; Show the unit field
        MOV     O$FID(R3),(R2)+ ; Show the 3 FID words
        MOV     O$FID+2(R3),(R2)+
        MOV     O$FID+4(R3),(R2)+
        MOV     #NAMSTR,R1      ; Set the format string
        CALL    PRINT           ; Display it
        MOV     #FAB,R0         ; Reset for $OPEN
        $OPEN   R0              ; Attempt the open
        $COMPARE #0,STS,R0      ; Status OK?
        BLT     ERROR           ; No if LE; show error
        MOV     #FIDSTR,R1      ; Show that the open worked
        CALL    PRINT           ; Display the text
        MOV     #FAB,R0         ; Restore the FAB address
        $CLOSE  R0              ; (We assume no $CLOSE errors)
        INC     FILCNT          ; Count this wildcard
        BR      GETFIL          ; Do this again


EXIT:
        EXIT$S                  ; Task exit

SEAERR:
        $COMPARE #ER$NMF,STS,R0 ; No more matches?
        BNE     ERROR           ; No - some other error
        MOV     FILCNT,(R2)+    ; Set the cound of matches
        BEQ     ERROR2          ; No files...
        CLR     (R2)            ; Give the ESA
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #TTLSTR,R1      ; Set the format string
        BR      ERROR1          ; Go show it and exit
ERROR2:
        CLR     -(R2)           ; Setup for string length
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance to next word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #NOFILE,R1      ; Set the format string
        BR      ERROR1          ; Print the error
ERROR:
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
ERROR1:
        CALL    PRINT           ; Go edit and print the message
        JMP     OPEN2           ; Let's try this again
```

```
PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,Q.IOPL(R5)
        MOV     R1,Q.IOPL+2(R5)
        DIR$    R5              ; Send to the terminal
        MOV     #EDBLK,R2       ; Restore edit block
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

SYDSKA: .Ascii  "SY:"
        SYDSKL = . - SYDSKA
QUES:   .Ascii  <15><12>"File(s) to $OPEN: "
        QUESL = . - QUES
ERRSTR: .Asciz  "%N$OPEN error -- STS=%P, STV=%P%N"
RSSSTR: .Asciz  "%N  File %VA found by $SEARCH"
NAMSTR: .Ascii  "    NAM data returned: DVI='%VA' DVI+2=%P "
        .Asciz  "FID=(%P,%P,%P)%N"
FIDSTR: .Asciz  "  File successfully $OPENed by FID"
NOFILE: .Asciz  "%NNo files matching %VA found%N"
TTLSTR: .Asciz  "%NTotal of %D files matching %VA $OPENed by FID%N"

        .EVEN
FILCNT: .WORD   0
EDBLK:  .BLKW   10.
BUFFER: .BLKB   128.
BUFF:   .Blkb   128.
EXPSTR: .BLKB   128.
RESSTR: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    OPEN2
```

Example 7-7: OPEN3 - $OPEN with Implicit Wildcards (Illegal)

```
        .TITLE  OPEN3 - $OPEN WITH IMPLICIT WILDCARDS (ILLEGAL)
        .IDENT  /X01.00/

        .ENABL  LC

        .MCALL  FAB$B,NAM$B,GSA$
        .MCALL  $PARSE,$OPEN,$CLOSE
        .MCALL  $STORE,$FETCH,$COMPARE,$SET,$OFF,ORG$
        .MCALL  ALUN$S,QIOW$,DIR$,EXIT$S

;
; This program tests/demonstrates the use of
; the RMS-11 $PARSE, $OPEN with implicit wildcards.
; NOTE: This test is intended to show failure, as
;       $OPEN by implicit wildcards is not supported.
;

;
; RMS-11 Data Structures
;

        ORG$    SEQ             ; $OPEN sequential
        ORG$    REL             ; $OPEN relative
        ORG$    IDX             ; $OPEN indexed

        GSA$    GSA

FAB::   FAB$B                   ; Argument FAB
         F$DNA  SYDSKA          ; If no device, SY:
         F$DNS  SYDSKL          ; should be used
         F$NAM  NAM             ; Link to NAM
         F$LCH  2.              ; Channel #2
        FAB$E

NAM::   NAM$B                   ; NAM definition
         N$ESA  EXPSTR          ; EXP STR address
         N$ESS  128.            ; EXP STR length
         N$RSA  RESSTR          ; RES STR address
         N$RSS  128.            ; RES STR length
        NAM$E


        .PSECT  $CODE$,RO,I

OPEN3::
        ALUN$S  #1,#"TI,#0      ; Assign the terminal
        MOV     #FAB,R0         ; Map the target FAB
        MOV     #EDBLK,R2       ; Map the exit block
        MOV     #NAM,R3         ; Map the target NAM
        MOV     #READ,R4        ; Map the input DPB
        MOV     #WRITE,R5       ; Map the output DPB
        CLR     Q.IOPL+4(R5)    ; Turn off carriage ctl
        MOV     #QUES,Q.IOPL(R5)
        MOV     #QUESL,Q.IOPL+2(R5)
        DIR$    R5              ; Prompt for the FNA
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        MOV     #BUFF,Q.IOPL(R4)
        MOV     #64.,Q.IOPL+2(R4)
        DIR$    R4              ; Get the response
        TSTB    IOSTAT          ; Check the IOSB
        BMI     EXIT            ; Exit if error
        $STORE  IOLEN,FNS,R0    ; Set the string length
```

```
        $STORE  #BUFF,FNA,R0    ; Set the string address
        MOV     #40,Q.IOPL+4(R5); Restore carriage control
        CLR     FILCNT          ; Init count of matches
;;      $SET    #FB$FID,FOP,R0  ; Turn on FID bit
        $PARSE  R0              ; Parse the strings
        $COMPARE #0,STS,R0      ; An error?
        BLT     ERROR           ; Yes if MI; display it
        CLR     FILCNT          ; Clear the match count
GETFIL:
        MOV     #EDBLK,R2       ; Reset the edit block addr
        MOV     #FAB,R0         ; Reset the FAB address
        $OPEN   R0              ; $OPEN a matching file
        $COMPARE #0,STS,R0      ; Error?
        BLT     SEAERR          ; Yes if LT
        CLR     (R2)            ; Init the length
        $FETCH  (R2),RSL,R3     ; Get the string length
        TST     (R2)+           ; Advance
        $FETCH  (R2)+,RSA,R3    ; Get the string address
        MOV     #RSSSTR,R1      ; Select the format string
        CALL    PRINT           ; Display the file's RS
        MOV     #2,(R2)+        ; Setup to show NAM info
        MOV     #NAM+O$DVI,(R2)+
        MOV     O$DVI+2(R3),(R2)+ ; Show the unit field
        MOV     O$FID(R3),(R2)+ ; Show the 3 FID words
        MOV     O$FID+2(R3),(R2)+
        MOV     O$FID+4(R3),(R2)+
        MOV     #NAMSTR,R1      ; Set the format string
        CALL    PRINT           ; Display it
        MOV     #FAB,R0         ; Reset for $OPEN
        $CLOSE  R0              ; (We assume no $CLOSE errors)
        INC     FILCNT          ; Count this wildcard
        BR      GETFIL          ; Do this again


EXIT:
        EXIT$S                  ; Task exit

SEAERR:
        $COMPARE #ER$NMF,STS,R0 ; No more matches?
        BNE     ERROR           ; No - some other error
        MOV     FILCNT,(R2)+    ; Set the cound of matches
        BEQ     ERROR2          ; No files...
        CLR     (R2)            ; Give the ESA
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #TTLSTR,R1      ; Set the format string
        BR      ERROR1          ; Go show it and exit
ERROR2:
        CLR     -(R2)           ; Setup for string length
        $FETCH  (R2),ESL,R3     ; Set the length
        TST     (R2)+           ; Advance to next word
        $FETCH  (R2),ESA,R3     ; Set the address
        MOV     #NOFILE,R1      ; Set the format string
        BR      ERROR1          ; Print the error
ERROR:
        $FETCH  (R2)+,STS,R0    ; Set the STS returned
        $FETCH  (R2),STV,R0     ; And the STV
        MOV     #ERRSTR,R1      ; Set the error format string
ERROR1:
        CALL    PRINT           ; Go edit and print the message
        JMP     OPEN3           ; Let's try this again

PRINT:
        MOV     #EDBLK,R2       ; Setup edit
        MOV     #BUFFER,R0      ; Output buffer
```

```
        CALL    $EDMSG          ; Exit the string
        MOV     #BUFFER,Q.IOPL(R5)
        MOV     R1,Q.IOPL+2(R5)
        DIR$    R5              ; Send to the terminal
        MOV     #EDBLK,R2       ; Restore edit block
        RETURN                  ; Return to caller


        .PSECT  $DATA$,RW,D

SYDSKA: .Ascii  "SY:"
        SYDSKL = . - SYDSKA
QUES:   .Ascii  <15><12>"File(s) to $OPEN: "
        QUESL = . - QUES
ERRSTR: .Asciz  "%N$OPEN error -- STS=%P, STV=%P%N"
RSSSTR: .Asciz  "%N  File %VA successfully $OPENed"
NAMSTR: .Ascii  "    NAM data returned: DVI='%VA' DVI+2=%P "
        .Asciz  "FID=(%P,%P,%P)%N"
NOFILE: .Asciz  "%NNo files matching %VA found%N"
TTLSTR: .Asciz  "%NTotal of %D files matching %VA $OPENed%N"

        .EVEN
FILCNT: .WORD   0
EDBLK:  .BLKW   10.
BUFFER: .BLKB   128.
BUFF:   .Blkb   128.
EXPSTR: .BLKB   128.
RESSTR: .BLKB   128.
IOSTAT: .WORD   0
IOLEN:  .WORD   0
READ:   QIOW$   IO.RLB,1,1,,IOSTAT
WRITE:  QIOW$   IO.WLB,1,1,,IOSTAT,,<,,40>

        .END    OPEN3
```

Example 7-8: GSA - Core Space Allocator

```
        .Title  GSA - Core space allocator
        .Ident  /V02.00/

        .Enabl  LC

;
; Copyright (C) 1982,   Digital Equipment Corporation
;                       Maynard, Massachusetts 01754
;
; **-GSA - Dynamic memory allocation for RMS-11 pool
;
;
;  Called by RMS-11 to manage pool space.
;  In the event of pool exhaustion, the task
;  image will be extended to obtain more space.
;
;  May be called by user written code providing
;  the interface standard is adhered to.
;
;  Interface:
;    Request space:
;      R0 ->  RMS/user Pool list head (maintained by RL/CQB)
;      R1 :=  Amount of space requested (bytes)
;      R2 :=  0 (differentiates between request and release)
;
;    Release space:
;      R0 ->  RMS Pool list head (maintained by RL/CQB)
;      R1 :=  Amount of space to be released (bytes)
;      R2 ->  Base address (for release)
;
;
;  Returns:
;    C-Bit "set"   if an error has occurred (failure)
;    C-Bit "clear" if no error has occurred (success)
;

        .Mcall  Extk$S

        .Page
        .Sbttl  Control block definitions

        .Psect  GSA$$D,RW,D

;
; GSA internal data:
;
;   GSABAS - Base address for the next memory allocation.
;            Initially set to zero, it will be assigned
;            the first address outside of the task's
;            current address limits.
;   GSAMIN - Decimal value reflecting the minimum size
;            (in bytes) to extend the task in order to
;            provide space to the pool.
;   GSAREQ - Requested pool block number.  If a request
;            for the 'GSAMIN' fails, then the original
;            allocation size will be attempted.  If that
;            fails, then there is no more memory left.
;

GSABAS::                        ; GSA base address
        .Word   000000          ; (for next allocation)
GSAMIN::                        ; Minimum allocation
```

```
        .Word   512./64.        ; (in 32-word blocks)
GSAREQ::                        ; Size of this request
        .Word   000000          ; (if 'GSAMIN' extends fail)


        .Page
        .Sbttl  GSA Initialization code

        .Psect  GSA$$I,RO,I

;
; GSA Initialization
;
;  This code is entered when GSA is entered with GSABAS
;  set to zero.  In order to be able to build valid pool
;  header tables, GSABAS must be properly initialized and
;  maintained.
;
;  Initialization consists of finding the size of the task
;  in 32-word units, and converting that value to a usable
;  16-bit address (which corresponds to the address of the
;  next task extension (Extk$S) call.  Once GSABAS has been
;  initialized, GSAINI will not be reused.
;

GSAINI:
        Mov     R0,-(SP)        ; R0-2 will be used to
        Mov     R1,-(SP)        ; communicate with $INIDM
        Mov     R2,-(SP)        ; NOTE: $INIDM uses EXTSK.

;
; The following code will use $INIDM to initialize the
; dynamic memory. Contrary to documentation, R1 will return
; the first address following the task image, and R2 will
; return the size of the "free" memory from that address.
;
; NOTE: $INIDM and EXTSK reside in LB:[1,1]VMLIB for RSX
;       systems, and in LB:[1,1]SYSLIB for RSTS/E systems.
;
; $INIDM interface:
;   Calls:
;     R0 ->  Pool list head
;
;   Returns:
;     R0 ->  First address in task
;     R1 ->  First address AFTER task
;     R2 :=  Size of free core after task (based at R1)
;

        Call    $INIDM          ; Initialize dynamic memory
        Mov     R1,GSABAS       ; Setup the "free" address
        Mov     (SP)+,R2        ; Restore the registers
        Mov     (SP)+,R1        ;
        Mov     (SP)+,R0        ;
        Return                  ; And return to GSA


        .Page
        .Sbttl  GSA Mainline code

        .Psect  GSA$$M,RO,I
;
; GSA Mainline
;
;   Entry point is "GSA", with registers 0-2 loaded as
;   described above.
```

```
;
GSA::

;
; First, determine if dynamic memory has been initialized.
; GSABAS (initially set to zero) will be non-zero if $INIDM
; has been called and the memory list initialized.  On RSX
; based systems it is possible to install tasks with an
; extension (/INCREMENT).  $INIDM will detect this and setup
; the first memory entry in the pool list.
;
; A point to note: If the RSX task has been installed with
; the non-checkpointable (/-CP) flag, then EXTKs will not
; return success.  If it is necessary to install the task
; non-checkpointable, then the task should be installed with
; and increment value.
;
        Tst     GSABAS          ; Dynamic memory initialized?
        Bne     10$             ; Yes if NE, proceed
        Call    GSAINI          ; Otherwise, initialize pool

10$:

;
; Determine if this call involves real memory.
; R1 should contain the size (in bytes) of the core
; block requested or to be released.  If zero then
; return to the caller without an error (TST leaves CC).
;
        Tst     R1              ; Real memory?
        Bne     20$             ; Yes if NE, then process it
        Return                  ; Otherwise return with success

20$:

;
; If this call is a request for space, pass control
; to the allocation routines.  Otherwise, pass control
; to the system deallocation module $RLCB.  There is
; no need to return, so control is passed via JMP.
;
        Tst     R2              ; Address specified? (release)
        Beq     30$             ; No if EQ, then it's a request
        Jmp     $RLCB           ; Otherwise it's a release; do it
30$:

;
; Save our current context:
;   R0 = Pool list head
;   R1 = Size of memory required
;   R2 = 0 (signifies request)
;
        Mov     R0,-(SP)
        Mov     R1,-(SP)
        Mov     R2,-(SP)

;
; Attempt an allocation from the current pool
; If this is successful, pass control to the
; common exit.
;
```

```
        Call    $RQCB           ; Try the allocation
        Bcc     70$             ; CC signifies success

;
; Now that the initial allocation failed, we must extend
; the task and give the new area (extended into) to the
; caller.  To do this, the following procedure is used:
;
;       1.  The task is extended
;       2.  The area extended is returned to the
;           pool specified as if a release was attempted
;       3.  We retry the allocation operation, but
;           this time it should succeed, since we have
;           increased the size of the pool area
;
; NOTE: $RQCB has a bad habit of nuking registers, so it
;       becomes necessary to save and restore them around
;       unsuccessful calls.
;

        Mov     2(SP),R1        ; Obtain the request size

;
; Determine what the requirement is in 32-word blocks.
; Retain this value to allow GSA to decide whether
; to issue further task extension directives in
; order to satisfy the requirements.
;

        Add     #63.,R1         ; Round the request
        Asr     R1              ; to a 32-word boundary
        Asr     R1              ; Then convert the value
        Asr     R1              ; to the number of
        Asr     R1              ; 32-word blocks.
        Asr     R1
        Asr     R1
        Mov     R1,GSAREQ       ; Save the real size

;
; We will allocate core to the pool in "reasonable"
; increments to cut down on system overhead, and pool
; fragmentation.  This is accomplished by using either
; the requested size, or "GSAMIN", whichever is LARGER.
; If the request is unsuccessful, and the amount is
; smaller than GSAMIN, then request that particular size.
;

        Cmp     R1,GSAMIN       ; Smaller than minimum?
        Bhi     40$             ; No if HI, use it as is
        Mov     GSAMIN,R1       ; Otherwise use GSAMIN
40$:

;
; Now we attempt to extend the task by that size.
; If the request fails, then use the size of the
; original request.  If that also fails, then we
; simply ran out of memory.
;

        Extk$S  R1              ; Extend the task
        Bcc     60$             ; CC if successful
        Cmp     R1,GSAREQ       ; Is this request?
        Blos    50$             ; Yes if LOS, the end
        Mov     GSAREQ,R1       ; Otherwise try to use
        Br      40$             ; the actual request
50$:
```

```
        Sec                     ; Mark failure
        Br      70$             ; And exit

;
; The task has been extended, now this memory must be
; released to the pool for future allocation.
; To do this, we setup the registers as if RMS were
; going to release the core, and call ourself to do
; the work.  When the area has been released to the
; pool, we will return inline and proceed to reenter
; our code again from start to reattempt the allocation.
;

60$:
        Mov     4(SP),R0        ; Setup the PLH
        Asl     R1              ; Convert the real
        Asl     R1              ; size to the actual
        Asl     R1              ; 16-bit size that
        Asl     R1              ; was allocated.
        Asl     R1              ; The virtual address
        Asl     R1              ; should be after the
        Mov     GSABAS,R2       ; task (which is now
        Add     R1,GSABAS       ; part of the task)
        Call    GSA             ; Call ourself to release

;
; At this point, the new memory has been added to the
; pool, and is available for use.  We now reattempt
; to allocate the memory required.
;

        Mov     (SP)+,R2        ; Restore our registers
        Mov     (SP)+,R1        ; to the initial state
        Mov     (SP)+,R0        ; upon entry, and reenter
        Br      GSA             ; as if it's a new request

;
; Common exit.  Leave the registers in their current state,
; and return control to the caller.
;

70$:
        Inc     (SP)+           ; These won't alter the
        Bit     (SP)+,(SP)+     ; C-bit, so status remains
        Return                  ; unchanged upon return

        .End
```

APPENDIX A

COMPLETION CODES AND FATAL ERROR CODES

Section A.1 describes RMS-11 completions that are returned in the STS and STV fields of FABs and RABs. Section A.2 describes RMS-11 fatal error completions.

A.1 COMPLETIONS RETURNED IN STS AND STV FIELDS

This section lists and explains RMS-11 completions that are returned in the STS and STV fields of FABs and RABs. For each completion, the symbol, message, octal and decimal values, and explanation are given.

SU$SUC **Operation succeeded** Octal: 000001 Decimal: 1

SU$DUP **Inserted record has duplicate key** Octal: 000002 Decimal: 2

The PUT or UPDATE operation inserted a record whose key duplicates a key already in the index.

SU$IDX **Error updating index** Octal: 000003 Decimal: 3

The PUT or UPDATE operation inserted the record properly, but RMS-11 did not optimize the index structure; subsequent retrievals of the record will require extra I/O operations.

ER$ACC **File access error** Octal: 177740 Decimal: -32

1. A relative or indexed file is in the initial stage of creation and cannot be accessed yet. 2. A write-accessed file was not properly closed. 3. The file processor could not access the file. The STV field of the FAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$ACT **Activity precludes operation** Octal: 177720 Decimal: -48

RMS-11 could not perform the requested operation because of an activity in progress (for example, RMS-11 cannot perform the CLOSE operation for a file that has an outstanding asynchronous operation).

ER$AID **Bad value in AID field** Octal: 177700 Decimal: -64

The file contains no area with the area number given in the AID field of an ALL block.

ER$ALN **Bad mask in ALN field** Octal: 177660 Decimal: -80

The ALN field of an ALL block contains an invalid value.

ER$ALQ **Bad value in ALQ field** Octal: 177640 Decimal: -96

The ALQ field of a FAB or an ALL block contains an invalid value; the value in the ALQ field is either too large, or is 0 for an EXTEND operation.

ER$AOP **Bad mask in AOP field** Octal: 177600 Decimal: -128

The AOP field of an ALL block contains an invalid mask value.

ER$ATR **Error reading attributes** Octal: 177540 Decimal: -160

The file processor could not read the attributes for the file. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$ATW **Error writing attributes** Octal: 177520 Decimal: -176

The file processor could not write the attributes for the file. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$BKS **Bad value in BKS field** Octal: 177500 Decimal: -192

The value in the BKS field of the FAB is too large.

ER$BKZ **Bad value in BKZ field** Octal: 177460 Decimal: -208

The value in the BKZ field of an ALL block is too large; or the bucket sizes of the lowest (LAN) and upper (IAN) areas of an index are not equal.

ER$BOF **Beginning-of-file found** Octal: 177430 Decimal: -232

The SPACE operation backspaced to the beginning-of-file.

ER$BPA **Bad address in BPA field** Octal: 177420 Decimal: -240

The value in the BPA field of the FAB is odd, and the BPS field contains a nonzero value.

ER$BPS Bad value in BPS field Octal: 177400 Decimal: -256

The value in the BPS field of the FAB is nonzero and not a multiple of 4, and the BPA field is nonzero.

ER$CCR RAB already in use Octal: 177340 Decimal: -288

The CONNECT operation could not connect a stream using the specified RAB because the file is sequential and does not allow multiple connected streams.

ER$CHG Illegal record key change Octal: 177320 Decimal: -304

The UPDATE operation did not allow a changed record key because the index does not allow key changes or does not allow duplicate key values.

ER$CHK Bad bucket header Octal: 177300 Decimal: -320

The bucket header data for an indexed file is corrupted.

To recover from the error, follow this procedure:

1. Move the disk to a different drive and try the process again. If the process succeeds, the error was a hardware error; report the faulty hardware and continue processing. If the process fails again, proceed to the next step.

2. Recreate the file by fetching records from the old file using sequential access on the primary index. If this fails, proceed to the next step.

3. Restore the file from a backup copy.

ER$CLS File processor error Octal: 177260 Decimal: -336

The file processor returned an error condition to the CLOSE operation. The STV field of the FAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$COD Bad code in COD field Octal: 177240 Decimal: -352

The value in the COD field of an XAB is not valid.

ER$CRE File processor error Octal: 177220 Decimal: -368

The file processor returned an error condition to the CREATE operation. The STV field of the FAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$CUR Undefined current-record context Octal: 177200 Decimal: -384

A DELETE, TRUNCATE, or UPDATE operation required a defined current-record context, but it was undefined.

ER$DAN Bad value in DAN field — Octal: 177140, Decimal: -416

The value in the DAN field of a KEY block specifies a nonexistent area.

ER$DEL Record having RFA deleted — Octal: 177120, Decimal: -432

The record specified by RFA has been deleted.

ER$DEV Bad device specification — Octal: 177100, Decimal: -448

The device specification given contains a syntax error, there is no such device, the device is inappropriate for the operation, or two different devices have been specified for a RENAME operation.

ER$DFW File processor error — Octal: 177070, Decimal: -456

The file processor returned an error while writing deferred-write data. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$DIR Bad directory specification — Octal: 177060, Decimal: -464

The directory specification contains a syntax error.

ER$DME Pool exhausted — Octal: 177040, Decimal: -480

One of the five pools that RMS-11 uses cannot provide needed space for the operation.

ER$DNA Bad address in DNA field — Octal: 177030, Decimal: -488

The DNA field of the FAB contains 0, but the DNS field is nonzero.

ER$DNF No such directory — Octal: 177020, Decimal: -496

The directory specification given specifies a nonexistent directory.

ER$DNR Device not ready — Octal: 177000, Decimal: -512

The device specified is not on line.

ER$DTP Bad code in DTP field — Octal: 176760, Decimal: -528

The value in the DTP field of a KEY block does not specify a valid key data type.

ER$DUP **Duplicate key not allowed** Octal: 176740 Decimal: -544

The record offered for insertion had a record key that would duplicate a record already in the index, but the index does not allow duplicate keys.

ER$ENT **File processor error** Octal: 176720 Decimal: -560

The file processor could not create the specified directory entry. The STV field of the FAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$ENV **Feature not in selected RMS-11 environment** Octal: 176700 Decimal: -576

The RMS-11 environment (selected with the ORG$ macro or by the compiler or by the manner in which RMS-11 code is linked with your program) does not include the attempted operation for the specified file organization.

ER$EOF **End-of-file reached** Octal: 176660 Decimal: -592

The operation specified a record or block that is past the last record or block.

ER$ESA **Bad address in ESA field** Octal: 176650 Decimal: -600

The ESA field of the NAM block contains 0.

ER$ESL **Bad value in ESL field** Octal: 176644 Decimal: -604

The ESL field of the NAM block contains 0.

ER$ESS **ESS field value too small** Octal: 176640 Decimal: -608

The value in the ESS field of the NAM block specifies an expanded string buffer that is too small to contain the expanded string.

ER$EXP **File expiration date not yet reached** Octal: 176630 Decimal: -616

ER$EXT **File processor error** Octal: 176620 Decimal: -624

The file processor could not make the requested extension to the file. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$FAC **FAC field forbids operation** Octal: 176560 Decimal: -656

The attempted record or block operation was not specified in the FAC field of the FAB when the file was created or opened.

ER$FEX **File already exists** Octal: 176540 Decimal: -672

The file specified for creation already exists, but supersession was not specified.

ER$FID **Bad value in FID field** Octal: 177530 Decimal: -680

The FID field of the NAM block contains a value that is not a file identifier.

ER$FLG **Bad mask in FLG field** Octal: 176520 Decimal: -688

The combination of masks specified in the FLG field of a KEY block is illegal.

ER$FLK **File locked by another task** Octal: 176500 Decimal: -704

The file sharing specified is not allowed by a task already accessing the file.

ER$FNA **Bad address in FNA field** Octal: 176470 Decimal: -712

The FNA field of the FAB contains 0, but the FNS field is nonzero.

ER$FND **File processor error** Octal: 176460 Decimal: -720

The file processor could not find the file specified. The STV field of the FAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$FNF **File not found** Octal: 176440 Decimal: -736

The file specified for a directory or file operation does not exist.

ER$FNM **Bad file name** Octal: 176420 Decimal: -752

The file name portion of a file specification string has a syntax error.

ER$FOP **Bad mask in FOP field** Octal: 176400 Decimal: -768

The FOP field of the FAB contains one or more illegal masks.

ER$FUL **Device or file full** Octal: 176360 Decimal: -784

The specified device or file has no room to allow file creation or extension.

ER$IAN **Bad value in IAN field** Octal: 176340 Decimal: -800

The value in the IAN field of a KEY block specifies a nonexistent file area.

ER$IDX Index not initialized Octal: 176320 Decimal: -816

This code is only returned in the STV field of the RAB in conjunction with the code ER$RNF in the STS field. It indicates that no entries have been made in the index specified for the GET or FIND operation.

ER$IFI Bad value in IFI field Octal: 176300 Decimal: -832

The value in the IFI field of the FAB is not the internal file identifier for a file.

ER$IMX Too many XABs of same type Octal: 176260 Decimal: -848

The number of XABs of the same type in the chain of XABs is too large (more than 254 ALL blocks or KEY blocks, more than 1 DAT block, PRO block, or SUM block).

ER$IOP Illegal operation for file Octal: 176220 Decimal: -880

The requested operation is illegal for the file organization or for the allowed access.

ER$IRC Illegal record found in sequential file Octal: 176200 Decimal: -896

The record length field of a record in a sequential file is invalid.

ER$ISI Bad value in ISI field Octal: 176160 Decimal: -912

The ISI field of the RAB contains a value that is not an internal stream identifier.

ER$KBF Bad address in KBF field Octal: 176140 Decimal: -928

The KBF field of the RAB contains 0.

ER$KEY Bad key Octal: 176120 Decimal: -944

The key specified for a key access operation is invalid (either a negative RRN or an erroneous packed-decimal key).

ER$KRF Bad value in KRF field Octal: 176100 Decimal: -960

The KRF field of the RAB contains (or contained) a value that does not specify a file index. For a key access FIND or GET operation, the RAB contains the invalid value in its KRF field; for a sequential access FIND or GET operation, the RAB contained the invalid value in its KRF field during an earlier CONNECT or REWIND operation.

ER$KSZ Bad value in KSZ field Octal: 176060 Decimal: -976

The KSZ field of the RAB contains an invalid value.

ER$LAN Bad value in LAN field Octal: 176040 Decimal: -992

The value in the LAN field of a KEY block specifies a nonexistent file area.

ER$LBY Logical channel busy Octal: 176000 Decimal: -1024

The LCH field of the FAB contains the number of a logical channel that is already in use by the task.

ER$LCH Bad value in LCH field Octal: 175760 Decimal: -1040

The LCH field of the FAB contains a value that is too large to be a logical channel number.

ER$LEX Extension not needed Octal: 175750 Decimal: -1048

The requested extension was not needed because the file area still contains an unused extent.

ER$LOC Bad value in LOC field Octal: 175740 Decimal: -1056

The LOC field of an ALL block contains a value that does not specify a valid location.

ER$MEM Memory address rollover Octal: 175710 Decimal: -1080

The area specified for the file string, default string, expanded string, or resultant string extends beyond the end of addressable memory.

ER$MKD File processor error Octal: 175700 Decimal: -1088

The file processor could not mark the specified file for deletion. The STV field of the FAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$MRN Bad value in MRN field or bad record number Octal: 175660 Decimal: -1104

The MRN field of the FAB contains a negative number (CREATE operation), or the record number specified for a key access record operation is larger than the file maximum record number (specified in the MRN field at file creation).

ER$MRS Bad value in MRS field Octal: 175640 Decimal: -1120

The MRS field of the FAB contains 0 even though the file to be created is requested either to be a relative file or to have fixed-length records.

ER$NAM Bad address in NAM field Octal: 175620 Decimal: -1136

The NAM field of the FAB contains 0 or an odd address.

ER$NEF Context not end-of-file Octal: 175600 Decimal: -1152

The PUT operation could not insert a record into a sequential file because the next-record context was not the end-of-file.

ER$NMF No more matching files Octal: 175554 Decimal: -1172

The SEARCH operation ended the wildcard SEARCH series because there are no more files matching the wildcard file specification.

ER$NOD Bad node name Octal: 175550 Decimal: -1176

The specified node name is invalid or, for the RENAME operation, the two node names are different.

ER$NPK No primary key for indexed file Octal: 175540 Decimal: -1184

The CREATE operation did not create the specified file because no primary index was specified even though the request specified indexed file organization.

ER$ORD Ordering of XABs illegal Octal: 175500 Decimal: -1216

The chain of XABs for a directory or file operation is improperly ordered.

ER$ORG Bad mask in ORG field Octal: 175460 Decimal: -1232

The ORG field of the FAB contains an invalid file organization code; the file was not created.

ER$PLG Error reading file prologue Octal: 175440 Decimal: -1248

The data read from the file prologue is incorrect.

To recover from the error, follow this procedure:

1. Move the disk to a different drive and try the process again. If the process succeeds, the error was a hardware error; report the faulty hardware and continue processing. If the process fails again, proceed to the next step.

2. Recreate the file by fetching records from the old file using sequential access on the primary index. If this fails, proceed to the next step.

3. Restore the file from a backup copy.

ER$PLV File prologue version level unsupported Octal: 175430 Decimal: -1256

The file prologue version number shows that the file was created by a version of RMS that is not supported on your system.

ER$POS **Bad value in POS field** Octal: 175420 Decimal: -1264

The POS field of a KEY block contains a value that is greater than the maximum record size for the file; the STV field of the FAB contains the address of the KEY block.

ER$PRM **Bad file date read** Octal: 175400 Decimal: -1280

The file dates read are illegal.

ER$PRV **Privilege violation** Octal: 175360 Decimal: -1296

The file processor denied the requested operation because the task has no privilege for the operation.

ER$RAC **Bad mask in RAC field** Octal: 175320 Decimal: -1328

The RAC field of the RAB contains an illegal value.

ER$RAT **Bad mask in RAT field** Octal: 175300 Decimal: -1344

The RAT field of the FAB contains illegal set bits.

ER$RBF **Bad address in RBF field** Octal: 175260 Decimal: -1360

The RBF field of the RAB contains an odd address; the address must be even for block access.

ER$RER **File processor error** Octal: 175240 Decimal: -1376

The file processor could not read the requested record or block. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$REX **Record already exists** Octal: 175220 Decimal: -1392

The target cell for a PUT operation to a relative file already contains a record.

ER$RFA **Bad value in RFA field** Octal: 175200 Decimal: -1408

The RFA field of the RAB contains an illegal RFA.

ER$RFM **Bad code in RFM field** Octal: 175160 Decimal: -1424

The RFM field of the FAB contains an illegal value.

ER$RLK **Record locked** Octal: 175140 Decimal: -1440

The bucket containing the specified record is locked by another task or by another stream in your task.

ER$RNF **No such record** Octal: 175100 Decimal: -1472

The record specified for key access does not exist.

ER$RNL **Record not locked** Octal: 175060 Decimal: -1488

The FREE operation found that no record was locked for the stream.

ER$ROP **Bad mask in ROP field** Octal: 175040 Decimal: -1504

The ROP field of the RAB contained illegal set bits.

ER$RPL **File processor error** Octal: 175020 Decimal: -1520

The data read from the file prologue is incorrect.

To recover from the error, follow this procedure:

1. Move the disk to a different drive and try the process again. If the process succeeds, the error was a hardware error; report the faulty hardware and continue processing. If the process fails again, proceed to the next step.

2. Recreate the file by fetching records from the old file using sequential access on the primary index. If this fails, proceed to the next step.

3. Restore the file from a backup copy.

ER$RRV **Bad internal pointer** Octal: 175000 Decimal: -1536

An internal pointer in the file is invalid.

ER$RSL **Bad value in RLS field** Octal: 174754 Decimal: -1556

The RSL field of the NAM block contains 0.

ER$RSS **Bad value in RSS field** Octal: 174750 Decimal: -1560

The RSS field of the NAM block contains 0.

ER$RST **Bad address in RSA field** Octal: 174744 Decimal: -1564

The RSA field of the NAM block contains 0.

ER$RSZ **Bad value in RSZ field** Octal: 174740 Decimal: -1568

The RSZ field of the RAB contains a value that is larger than the maximum allowed record size, or (for fixed-length records) is not equal to the maximum record size, or (for an UPDATE operation to a sequential file) is not equal to the length of the record to be updated.

ER$RTB Record too big for user buffer Octal: 174720 Decimal: -1584

The record read cannot fit into the user buffer; the STV field of the RAB contains the size of the record, and the portion that will fit is moved to the user buffer as for a successful GET.

ER$RVU Internal pointer corrupted Octal: 174710 Decimal: -1592

The record insertion succeeded and the primary index was updated successfully; however, RMS-11 could not update internal pointers.

To recover from the error, follow this procedure:

1. Recreate the file by fetching records from the old file using sequential access on the primary index. If this fails, proceed to the next step.

2. Restore the file from a backup copy.

ER$SEQ Sequential insertion records not in order Octal: 174700 Decimal: -1600

The sequential access PUT operation found records whose primary keys were not in ascending order.

ER$SHR Bad mask in SHR field Octal: 174660 Decimal: -1616

The SHR field of the FAB contains an illegal mask.

ER$SIZ Bad value in SIZ field Octal: 174640 Decimal: -1632

The SIZ field of a KEY block contains an illegal value.

ER$SYS System error Octal: 174600 Decimal: -1664

The interface between RMS-11 and the system is in error; the STV field of the FAB or RAB contains the status code for a system directive. Please submit a Software Performance Report.

ER$TRE Index error Octal: 174560 Decimal: -1680

The index contains invalid data. Build a new file using either an RMS-11 utility (RMSIFL or RMSCNV) or using sequential access and the primary index to fetch the old records.

ER$TYP Bad file extension Octal: 174540 Decimal: -1696

The file extension in a file specification contains invalid syntax.

ER$UBF **Bad address in UBF field** Octal: 174520 Decimal: -1712

The UBF field of the RAB contains 0 or, for block access, an odd address.

ER$USZ **Bad value in USZ field** Octal: 174500 Decimal: -1728

The USZ field of the RAB contains 0.

ER$VER **Bad file version number** Octal: 174460 Decimal: -1744

The file version portion of a file specification contains a syntax error.

ER$WCD **Illegal wildcard in merged string** Octal: 174430 Decimal: -1768

The merged string contains a wildcard character, but wildcarding is not in progress or is illegal for the operation.

ER$WER **File processor error** Octal: 174420 Decimal: -1776

The file processor could not write to the file. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$WLK **Device write-locked** Octal: 174410 Decimal: -1784

The device specified is write-locked.

ER$WPL **File processor error** Octal: 174400 Decimal: -1792

The file processor could not write the file prologue. The STV field of the FAB or RAB contains the file processor error code; see your operating system documentation for the meaning of the code.

ER$XAB **Bad address in XAB field** Octal: 174360 Decimal: -1808

The XAB field of the FAB contains an odd address.

ER$XTR **Extraneous data in file specification** Octal: 174340 Decimal: -1824

The file specification contains extraneous characters. The value in the STV field of the FAB is the address of the first character beyond the end of the valid file specification.

A.2 FATAL ERROR COMPLETIONS

This section lists and explains RMS-11 completions that are returned in general register R0. These errors are fatal either because RMS-11 detected an internal error condition and could not continue, or because the RAB or FAB is of questionable validity and RMS-11 therefore did not write the completion in its fields.

ER$ACT Illegal concurrent operation — Octal: 177720, Decimal: -48

1. The FAB you specified is already in use by another operation. 2. You have illegally interrupted RMS-11 processing.

ER$AST Illegal operation at AST level — Octal: 177560, Decimal: -144

Your program attempted to use WAIT operation at AST level.

ER$BUG Error in RMS-11 internal data — Octal: 177360, Decimal: -272

RMS-11 detected an error in its internal data structures. The error may have been caused by your task writing into the structures; if you think your task did not cause the error, please submit a Software Performance Report to DIGITAL, including the following information:

- Contents of general registers and stack
- Operation and file organization for which the error occurred
- Task builder map of the task

ER$CPB Bad parameter block — Octal: 177230, Decimal: -360

The parameter block (pointed to by R5) for an operation macro has an invalid argument count or is at a zero or odd address.

ER$FAB Bad FAB — Octal: 176600, Decimal: -640

The value in the BID or BLN field of the specified FAB is not the correct identifier or block length for a FAB, or the address of the FAB is 0 or odd.

ER$LIB Resident library not available — Octal: 175744, Decimal: -1052

The version of the RMS-11 resident library needed for your task is not available.

ER$MAP Error in internal buffer mapping data

Octal: 175720
Decimal: -1072

RMS-11 detected an error in its internal data structures. The error may have been caused by your task writing into the structures; if you think your task did not cause the error, please submit a Software Performance Report to DIGITAL, including the following information:

- Contents of general registers and stack
- Operation and file organization for which the error occurred
- Task builder map of the task

ER$RAB Bad RAB

Octal: 175340
Decimal: -1312

The value in the BID or BLN field of the specified RAB is not the correct identifier or block length for a RAB, or the address of the RAB is 0 or odd.

APPENDIX B

ASSEMBLY-TIME MESSAGES

RMS-11 macros detect some errors during assembly. For each such error, the macro issues a .PRINT or .ERROR assembler directive with a message. This appendix shows these messages and their meanings.

$COMPARE MACRO - FIELD TOO LARGE

You can specify only a 1-byte or 1-word field as the field parameter for the $COMPARE macro.

$COMPARE MACRO - FIELD PARAMETER INVALID

You must specify a valid field mnemonic as the field parameter for the $COMPARE macro.

$FETCH MACRO - PC DESTINATION INVALID

You cannot specify the PC as the destination for the $FETCH macro.

$FETCH OR $STORE MACRO - ADDRESS MODE INVALID

You have used an illegal address mode in the source for a $STORE macro or in the destination for a $FETCH macro. See Chapter 3 for a description of legal address modes for these macros.

$FETCH OR $STORE MACRO - FIELD PARAMETER INVALID

You can specify only a valid field mnemonic as the field parameter for a field access macro.

$FETCH OR $STORE MACRO - FIELD TOO LARGE FOR GIVEN REGISTERS

You cannot specify the given register as the source or destination address because the field is larger than the remaining registers.

$FETCH OR $STORE MACRO - FIELD TOO LARGE FOR IMMEDIATE MODE

You can specify an immediate mode value for a field access macro only if you specify a 1-byte or 1-word field.

$FETCH OR $STORE MACRO - FIELD TOO LARGE FOR REGISTERS

You cannot specify a register as the source or destination address because the given field is too large.

$FETCH OR $STORE MACRO - REGISTER PARAMETER INVALID

You can specify only R0, R1, R2, R3, R4, or R5 as the register parameter for a field access macro.

$FETCH OR $STORE MACRO - REGISTER USAGES OVERLAP

You cannot specify the given register as the source or destination address because the indicated registers overlap the register containing the control block address.

$OFF MACRO - FIELD TOO LARGE

You can specify only a 1-byte or 1-word field as the field parameter for the $OFF macro.

$OFF MACRO - FIELD PARAMETER INVALID

You must specify a valid field mnemonic as the field parameter for the $OFF macro.

$SET MACRO - FIELD TOO LARGE

You can specify only a 1-byte or 1-word field as the field parameter for the $SET macro.

$SET MACRO - FIELD PARAMETER INVALID

You must specify a valid field mnemonic as the field parameter for the $SET macro.

$SETGSA MACRO - REGISTER PARAMETER INVALID

You must specify R0, R1, R2, R3, R4, or R5 as the register parameter for the $SETGSA macro.

$TESTBITS MACRO - FIELD TOO LARGE

You can specify only a 1-byte or 1-word field as the field parameter for the $TESTBITS macro.

$TESTBITS MACRO - FIELD PARAMETER INVALID

You must specify a valid field mnemonic as the field parameter for the $TESTBITS macro.

F$BSZ MACRO - BSZ FIELD NOT USED IN RMS-11

RMS-11 has no BSZ field in the FAB; therefore the F$BSZ macro cannot initialize the field.

F$JFN MACRO - JFN FIELD NOT USED IN RMS-11

RMS-11 has no JFN field in the FAB; therefore the F$JFN macro cannot initialize the field.

FAB$B MACRO - ALREADY IN BLOCK OR POOL DECLARATION

You cannot use the FAB$B macro to begin FAB declaration until you have ended the current block or pool declaration (using the FABE, NAME, POOLE, RABE or XAB$E macro).

FAB$B MACRO - FAB NOT WORD-ALIGNED

Use the .EVEN assembler directive before the FAB$B macro; this assures word-alignment for the FAB.

FAB$E MACRO - NOT IN FAB DECLARATION

You must begin a FAB declaration with the FAB$B macro before ending it with a FAB$E macro.

NAM$B MACRO - ALREADY IN BLOCK OR POOL DECLARATION

You cannot use the NAM$B macro to begin NAM block declaration until you have ended the current block or pool declaration (using the FABE, NAME, POOLE, RABE or XAB$E macro).

NAM$B MACRO - NAM BLOCK NOT WORD-ALIGNED

Use the .EVEN assembler directive before the NAM$B macro; this assures word-alignment for the NAM.

NAM$E MACRO - NOT IN NAM BLOCK DECLARATION

You must begin a NAM block declaration with the NAM$B macro before ending it with a NAM$E macro.

OPERATION MACRO - FAB OR RAB ADDRESS PARAMETER MISSING

You must specify a control block address for the operation macro; for a file operation, specify a FAB address; for a stream, record, or block I/O operation, specify a RAB address.

ORG$ MACRO - OPERATION PARAMETER INVALID

You can specify only CRE, DEL, FIN, GET, PUT, and UPD as operation parameters for the ORG$ macro.

ORG$ MACRO - ORGANIZATION PARAMETER INVALID

You can specify only IDX, REL, or SEQ as the organization parameter for the ORG$ macro.

ORG$ MACRO - ORGANIZATION PARAMETER MISSING

You must specify IDX, REL, or SEQ as the organization parameter for the ORG$ macro.

POOL$B MACRO - ALREADY IN BLOCK OR POOL DECLARATION

You cannot use the POOL$B macro to begin pool declaration until you have ended the current block or pool declaration (using the FABE, NAME, POOLE, RABE or XAB$E macro).

POOL$E MACRO - NOT IN POOL DECLARATION

You must begin a POOL declaration with the POOL$B macro before ending it with a POOL$E macro.

R$LSN MACRO - LSN FIELD NOT USED IN RMS-11

RMS-11 has no LSN field in the RAB; therefore the R$LSN macro cannot initialize the field.

RAB$B MACRO - ALREADY IN BLOCK OR POOL DECLARATION

You cannot use the RAB$B macro to begin RAB declaration until you have ended the current block or pool declaration (using the FABE, NAME, POOLE, RABE or XAB$E macro).

RAB$B MACRO - RAB NOT WORD-ALIGNED

Use the .EVEN assembler directive before the RAB$B macro; this assures word-alignment for the RAB.

RAB$B MACRO - RAB TYPE PARAMETER INVALID

You can specify only SYN, ASYN, or a null as the parameter for the RAB$B macro.

RAB$E MACRO - NOT IN RAB DECLARATION

You must begin a RAB declaration with the RAB$B macro before ending it with a RAB$E macro.

X$SIZ MACRO - TOTAL KEY SIZE TOO LARGE

The sum of the segment sizes for a key is greater than 255. Specify smaller segments.

XAB$B MACRO - ALREADY IN BLOCK OR POOL DECLARATION

You cannot use the XAB$B macro to begin XAB declaration until you have ended the current block or pool declaration (using the FABE, NAME, POOLE, RABE or XAB$E macro).

XAB$B MACRO - XAB NOT WORD-ALIGNED

Use the .EVEN assembler directive before the XAB$B macro; this assures word-alignment for the XAB.

XAB$B MACRO - XAB TYPE PARAMETER INVALID

You can specify only XB$ALL, XB$DAT, XB$KEY, XB$PRO, or XB$SUM as the XAB type parameter for the XAB$B macro.

XAB$E MACRO - NOT IN XAB DECLARATION

You must begin a XAB declaration with the XAB$B macro before ending it with a XAB$E macro.

APPENDIX C

MACROS THAT DECLARE SYMBOLS AND OTHER MACROS

Table C-1 lists RMS-11 macros (and their arguments) that declare symbols and other macros. In the table, the expression **xxx** represents a 2- or 3-character string, so that the expression **O$xxx** represents all symbols that begin with O$; the expression **fld** represents a field mnemonic.

Note that you can declare symbols either globally or locally. For a FABBT, RABBT, XAB$BT, or $RMSTAT macro, give the argument DFIN$G (or omit the argument) to define symbols globally; give the argument DFIN$L to define symbols locally.

Note also that you can declare symbols for control block sizes without declaring field-offset symbols. For a FABOF$, NAMOF$, RABOF$, XABOF$, XBAOF$, XBDOF$, XBKOF$, XBPOF$, or XBSOF$ macro, give the argument DEF$SZ to define only symbols for block sizes, or give no argument to define both symbols for block sizes and field-offset symbols.

Table C-1: Macros That Declare Symbols and Other Macros

Macro	Argument	Declares
FAB$B		- FAB field-initialization macros: of the form F$fld - FAB end-block-declaration macro: FAB$E - FAB field-offset symbols: of the form O$fld - FAB code and mask symbols: of the form FB$xxx
FAB$BT	DFIN$G	- Global FAB code and mask symbols: of the form FB$xxx (except FAB length symbol FB$BLN)
FAB$BT	DFIN$L	- Local FAB code and mask symbols: of the form FB$xxx (except FAB length symbol FB$BLN)
FABOF$		- FAB field offset symbols: of the form O$fld - FAB length symbol: FB$BLN
FABOF$	DEF$SZ	- FAB length symbol: FB$BLN
$FBCAL		- Directory operation macros: $ENTER, $PARSE, $REMOVE, $RENAME, and $SEARCH - File operation macros: $CLOSE, $CREATE, $DISPLAY, $ERASE, $EXTEND, and $OPEN

(continued on next page)

Table C-1 (cont.): Macros That Declare Symbols and Other Macros

Macro	Argument	Declares
$GNCAL		- Get-space address macros: GSA$, $GETGSA, and $SETGSA - Facilities-declaration macro: ORG$ - RMS-11 initialization macros: $INIT and $INITIF (obsolete) - Field-access macros: $COMPARE, $FETCH, $OFF, $SET, $STORE, and $TESTBITS - Completion-handler return macro: $RETURN
NAM$B		- NAM block field-initialization macros: of the form N$fld - NAM block end-block-declaration macro: NAM$E - NAM block field-offset symbols: of the form O$fld - NAM block code and mask symbols: of the form NB$xxx
NAMOF$		- NAM block field offset symbols: of the form O$fld - NAM block length symbol: NB$BLN
NAMOF$	DEF$SZ	- NAM block length symbol: NB$BLN
POOL$B		- Pool declaration macros: PBDB, PBUF, PFAB, PIDX, P$RAB, and P$RABX - End-pool-declaration macro: POOL$E
RAB$B		- RAB field-initialization macros: of the form R$fld - RAB end-block-declaration macro: RAB$E - RAB field-offset symbols: of the form O$fld - RAB code and mask symbols: of the form RB$xxx
RAB$BT	DFIN$G	- Global RAB code and mask symbols: of the form RB$xxx (except RAB length symbol RB$BLN or RB$BLL)
RAB$BT	DFIN$L	- Local RAB code and mask symbols: of the form RB$xxx (except RAB length symbol RB$BLN or RB$BLL)
RABOF$		- RAB field offset symbols: of the form O$fld - RAB length symbol: RB$BLN (for synchronous RAB) or RB$BLL (for asynchronous RAB)
RABOF$	DEF$SZ	- RAB length symbol: RB$BLN (for synchronous RAB) or RB$BLL (for asynchronous RAB)

(continued on next page)

Table C-1 (cont.): Macros That Declare Symbols and Other Macros

Macro	Argument	Declares
$RBCAL		- Stream operation macros: $CONNECT, $DISCONNECT, $FLUSH, $FREE, $NXTVOL, $REWIND, and $WAIT - Record operation macros: $DELETE, $FIND, $GET, $PUT, $TRUNCATE, and $UPDATE - Block operation macros: $READ, $SPACE, and $WRITE
$RMSTAT	DFIN$G	- Global completion symbols: of the forms ER$xxx and SU$xxx
$RMSTAT	DFIN$L	- Local completion symbols: of the forms ER$xxx and SU$xxx
XAB$B	XB$ALL	- ALL block field-initialization macros: of the form X$fld - XAB end-block-declaration macro: XAB$E - ALL block field-offset symbols: of the form O$fld - XAB code and mask symbols: of the form XB$xxx
XAB$B	XB$DAT	- DAT block field-initialization macros: of the form X$fld - XAB end-block-declaration macro: XAB$E - DAT block field-offset symbols: of the form O$fld - XAB code and mask symbols: of the form XB$xxx
XAB$B	XB$KEY	- KEY block field-initialization macros: of the form X$fld - XAB end-block-declaration macro: XAB$E - KEY block field-offset symbols: of the form O$fld - XAB code and mask symbols: of the form XB$xxx
XAB$B	XB$PRO	- PRO block field-initialization macros: of the form X$fld - XAB end-block-declaration macro: XAB$E - PRO block field-offset symbols: of the form O$fld - XAB code and mask symbols: of the form XB$xxx
XAB$B	XB$SUM	- SUM block field-initialization macros: of the form X$fld - XAB end-block-declaration macro: XAB$E - SUM block field-offset symbols: of the form O$fld - XAB code and mask symbols: of the form XB$xxx

(continued on next page)

Table C-1 (cont.): Macros That Declare Symbols and Other Macros

Macro	Argument	Declares
XAB$BT	DFIN$G	- Global XAB code and mask symbols: of the form XB$xxx (except XAB length symbols: XB$LAL, XB$DTL, XB$KYL, XB$PRL, and XB$SML)
XAB$BT	DFIN$L	- Local XAB code and mask symbols: of the form XB$xxx (except XAB length symbols: XB$LAL, XB$DTL, XB$KYL, XB$PRL, and XB$SML)
XABOF$		- XAB field offset symbols: of the form O$fld - XAB length symbols: XB$LAL, XB$DTL, XB$KYL, XB$PRL, and XB$SML
XABOF$	DEF$SZ	- XAB length symbols: XB$LAL, XB$DTL, XB$KYL, XB$PRL, and XB$SML
XBAOF$		- ALL block field offset symbols: of the form O$fld - ALL block length symbol: XB$LAL
XBAOF$	DEF$SZ	- ALL block length symbol: XB$LAL
XBDOF$		- DAT block field offset symbols: of the form O$fld - DAT block length symbol: XB$DTL
XBDOF$	DEF$SZ	- DAT block length symbol: XB$DTL
XBKOF$		- KEY block field offset symbols: of the form O$fld - KEY block length symbol: XB$KYL
XBKOF$	DEF$SZ	- KEY block length symbol: XB$KYL
XBPOF$		- PRO block field offset symbols: of the form O$fld - PRO block length symbol: XB$PRL
XBPOF$	DEF$SZ	- PRO block length symbol: XB$PRL
XBSOF$		- SUM block field offset symbols: of the form O$fld - SUM block length symbol: XB$SML
XBSOF$	DEF$SZ	- SUM block length symbol: XB$SML

APPENDIX D

RMS-11 WITH DIFFERENT OPERATING SYSTEMS

This appendix contrasts the behaviors of RMS-11 on different operating systems:

- PRO/RMS-11 versus RSTS/E RMS-11
- PRO/RMS-11 versus RSX-11M/M-PLUS RMS-11
- RSTS/E RMS-11 versus RSX-11M/MPLUS RMS-11

D.1 PRO/RMS-11 VERSUS RSTS/E RMS-11

This section contrasts the behaviors of PRO/RMS-11 and RSTS/E RMS-11.

D.1.1 Different Behaviors

The following features behave differently for RSTS/E and P/OS users:

- **Macro library location**

  RMS-11 macro libraries for the systems are located in the files:

RSTS/E	LB:RMSMAC.MLB
P/OS	LB:[1,1]RMSMAC.MLB

- **RTV field in FAB**

  The RTV field in the FAB has different uses:

RSTS/E	Cluster size
P/OS	Retrieval pointer count

- **Maximum bucket size**

  The maximum bucket sizes (given by the BKS field in the FAB or the BKZ fields in ALL blocks) are different:

RSTS/E	15 blocks
P/OS	32 blocks

- **Area alignment**

  The meanings of area alignment codes (in the ALN field of an ALL block) are different:

RSTS/E	XB$LBN	Cluster alignment
P/OS	XB$CYL	Cylinder alignment
	XB$LBN	Logical block alignment
	XB$VBN	Virtual block alignment

- **Protection codes**

  The protection codes (and defaults) are system-specific.

D.1.2 Features Not Supported on RSTS/E

The following RMS-11 features are not supported on RSTS/E, but are supported on P/OS:

- ENTER operation ($ENTER macro)
- NXTVOL operation ($NXTVOL macro)
- REMOVE operation ($REMOVE macro)
- SPACE operation ($SPACE macro)
- WAIT operation ($WAIT macro)
- User-provided interlocking (FB$UPI mask in SHR field of FAB)
- File version numbers (NB$VER mask in FNB field of NAM)
- Asynchronous execution of operations (RB$ASY mask in ROP field of RAB; SYN and ASYN arguments to RAB$B macro; RB$BLL symbol for length of asynchronous RAB)
- Directories (DID field in NAM block)
- Area extension (ALL block fields for $EXTEND macro)
- Contiguous file extension (FB$CTG mask in FOP field of FAB for $EXTEND macro)
- Hard placement (XB$HRD mask in AOP field of ALL block)
- Return of date and protection information by DISPLAY operation (PRO block fields and DAT block fields for $DISPLAY macro)
- File expiration date (EDT field of NAM block) and file revision number (RVN field of NAM block)
- Magtape devices are not supported on P/OS.
- Remote operations are not supported on P/OS.

D.1.3 Features Not Supported on P/OS

On RSTS/E, for compatibility with older file system, RMS-11 treats certain sequential files with undefined records as sequential files with stream records. P/OS will allow only block access to such files.

D.2 PRO/RMS-11 VERSUS RSX-11M/M-PLUS RMS-11

The P/OS operating system does not support magtape devices or remote RMS-11 operations.

P/OS files have decimal version numbers (NB$VER mask in FNB field of NAM).

D.3 RSTS/E RMS-11 VERSUS RSX-11M/M-PLUS RMS-11

This section contrasts the behaviors of RSTS/E RMS-11 and RSX-11M/M-PLUS RMS-11.

D.3.1 Different Behaviors

The following features behave differently for RSTS/E and RSX-11M/M-PLUS users:

- **Macro library location**

  RMS-11 macro libraries for the systems are located in the files:

RSTS/E	LB:RMSMAC.MLB
RSX-11M/M-PLUS	LB:[1,1]RMSMAC.MLB

- **RTV field in FAB**

  The RTV field in the FAB has different uses:

RSTS/E	Cluster size
RSX-11M/M-PLUS	Retrieval pointer count

- **Maximum bucket size**

  The maximum bucket sizes (given by the BKS field in the FAB or the BKZ fields in ALL blocks) are different:

RSTS/E	15 blocks
RSX-11M/M-PLUS	32 blocks

- **Area alignment**

  The meanings of area alignment codes (in the ALN field of an ALL block) are different:

RSTS/E	XB$LBN	Cluster alignment
RSX-11M/M-PLUS	XB$CYL	Cylinder alignment
	XB$LBN	Logical block alignment
	XB$VBN	Virtual block alignment

D.3.2 Features Not Supported on RSTS/E

The following RMS-11 features are not supported on RSTS/E, but are supported on RSX-11M/M-PLUS:

- ENTER operation ($ENTER macro)
- NXTVOL operation ($NXTVOL macro)
- REMOVE operation ($REMOVE macro)
- REWIND operation for magtape device ($REWIND macro)
- SPACE operation ($SPACE macro)
- WAIT operation ($WAIT macro)
- User-provided interlocking (FB$UPI mask in SHR field of FAB)
- Octal file version numbers (NB$VER mask in FNB field of NAM)
- Asynchronous execution of operations (RB$ASY mask in ROP field of RAB; SYN and ASYN arguments to RAB$B macro; RB$BLL symbol for length of asynchronous RAB)
- Directories (DID field in NAM block)
- Area extension (ALL block fields for $EXTEND macro)
- Contiguous file extension (FB$CTG mask in FOP field of FAB for $EXTEND macro)
- Hard placement (XB$HRD mask in AOP field of ALL block)
- Return of date and protection information by DISPLAY operation (PRO block fields and DAT block fields for $DISPLAY macro)
- File expiration date (EDT field in NAM block) and file revision number (RVN field of NAM block)
- Initial end-of-file context for magtape file (FB$NEF mask in FOP field of FAB for $OPEN macro)
- Multivolume magtapes

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

— — Do Not Tear - Fold Here and Tape —

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — Do Not Tear - Fold Here —

Cut Along Dotted Line