pdp11

IAS/RSX-11M

RMS-11 MACRO Programmer's Reference Manual

Order No. AA-0002A-TC

digital

IAS/RSX-11M

RMS-11 MACRO Programmer's Reference Manual

Order No. AA-0002A-TC

digital equipment corporation · maynard. massachusetts

First Printing, May 1977

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.

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

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11

CONTENTS

CONTENTS (CONT.)

FIGURES

TABLES

TABLES

PREFACE

This manual contains a complete description of RMS-11 (Record Management Services for the PDP-11) as implemented on the IAS and RSX-11M operating systems. Because the file and record services described herein apply to MACRO-11 programs, you should be familiar with both the RSX-11 MACRO-11 REFERENCE MANUAL and an appropriate PDP-11 PROCESSOR HANDBOOK. Additionally, you should read the INTRODUCTION TO RMS-11 MANUAL before using this manual.

Throughout this manual, the following conventions are used in the description of RMS-11 macro syntax.

1. Upper case words and letters, and punctuation marks other than those described in this preface, are written as shown.

2. Lower case words indicate that a value is to be substituted. The accompanying text specifies the nature of the item to be substituted.

3. Square brackets ([]), unless used in a UIC specification, enclose optional items.

4. An ellipsis (...) indicates that the preceding item or bracketed group may be repeated any number of times.

Lastly, unless otherwise noted, decimal radix is used for numeric values in all examples and accompanying explanatory text.

CHAPTER 1

INTRODUCTION

1.1 RMS-11 OVERVIEW

Record Management Services for PDP-11 operating systems (RMS-11) is a set of routines that enables programs to process files and records within files. RMS-11's variety of file organizations and record access modes gives you the ability to choose processing methods best suited to your application. RMS-11 files can be organized sequentially, relatively, or with embedded indexes. Using these file organizations, you can access records in a number of ways:

1. Sequentially.
2. Randomly by relative record number, key value, or record's file address (RFA).
3. Dynamically through a mixture of sequential and random access modes.

Through control blocks allocated in your program at assembly time, you transmit file and record operation requests to RMS-11. Through these same control blocks, RMS-11 returns to you the data contents of files, attribute information about files, and status codes.

To utilize RMS-11 facilities, you must understand how to:

1. Declare the RMS-11 facilities that your program requires.
2. Access fields in control blocks at runtime.
3. Allocate and initialize control blocks.
4. Perform file and record operations.

1.1.1 Declaring RMS-11 Facilities

Before processing an RMS-11 file, you must declare the RMS-11 facilities that your program requires and allocate space in your program for I/O buffers and internal RMS-11 control structures. RMS-11 provides a set of macros that allows you to calculate buffer and control structure requirements and provide for the selective linking of only those portions of RMS-11 actually required by your program.

1.1.2 Accessing Fields in Control Blocks

You communicate with RMS-11 through control blocks. Control blocks are formatted areas in your program. Each control block consists of individual data fields. At runtime, you can access control block data fields through special RMS-11 macros.

1.1.3 Allocating and Initializing Control Blocks

You must allocate space in your program for control blocks at assembly-time. Additionally, you can establish initial values for the fields in these blocks through assembly-time initialization macros.

1.1.4 Performing File and Record Operations

In combination with control blocks, a set of RMS-11 file and record operation macros forms the complete runtime program interface with RMS-11. Each such macro represents a request for a particular file or record service provided by RMS-11. The fields of control blocks further describe the request. Using particular RMS-11 macros, you can:

- Create new files
- Process existing files
- Extend or delete files
- Read, write, update, or delete records within files

1.2 ORGANIZATION OF INFORMATION IN THIS MANUAL

The organization of this manual corresponds to the areas discussed in the previous overview. However, an additional chapter is provided. Chapter 2 provides an overview of the program interface with RMS-11, including runtime processing macros and user control blocks and the interaction between these elements to produce file and record operations.

Chapter 3 describes the declaration of RMS-11 facilities. It includes descriptions of macros that declare the use of other macros, cause the initialization of the RMS-11 system at runtime, declare buffer and control structure space requirements, and specify the processing environment.

Chapter 4, on accessing control block fields at runtime, describes a set of general-purpose macros. These macros result in the generation of code that, at runtime, manipulates the contents of control block fields.

Chapter 5 discusses the allocation of the user control block known as the File Access Block (FAB). This chapter further provides descriptions of the function of each field in the FAB and the macros provided by RMS-11 to initialize these fields at assembly-time.

Chapter 6 describes the user control block known as the Record Access Block (RAB).

Chapter 7 details control blocks known as Extended Attribute Blocks (XABs).

Chapter 8 discusses the Name Block (NAM).

Chapter 9 details file and record operations. In addition to describing the macros that cause these operations, this chapter also discusses establishing and terminating a record access stream, the current context of record operations, file sharing, synchronous and asynchronous operations, and record transfer modes.

Additionally, a number of appendixes provide detailed information of further interest. Appendix A lists the status codes returned by RMS-11. Appendix B describes block I/O, a facility that allows your program to bypass entirely the record processing capabilities of RMS-11. Appendix C summarizes the handling of ANSI-labeled magnetic tapes.

Appendix D contains formulas enabling you to calculate the data capacities of RMS-11 files. Sample code segments demonstrating the program interface to RMS-11 are shown in Appendix E.

Finally, Appendix F summarizes the steps you must take to assembly and task build programs that use RMS-11 facilities.

CHAPTER 2

THE PROGRAM INTERFACE WITH RMS-11

To obtain RMS-11 services at runtime, your program must contain processing macros and user control blocks. This chapter introduces these macros and control blocks and summarizes their role in the processing of RMS-11 files. Subsequent chapters expand this introductory material and provide the detailed information necessary to write programs that use RMS-11 facilities.

RMS-11 processing macros are expanded at assembly-time. The resulting code is executed at runtime to perform the desired operation. Each macro represents a program request for a file or record related service.

With every request for a service, information is exchanged between your program and RMS-11. User control blocks are the means by which this exchange occurs. Prior to issuing a request for an RMS-11 service, your program must place information detailing the request in a control block. For example, a request to open a file must be accompanied by the name of the file, information on how the file will be accessed, and details on how the file is to be shared. As another example, a program request to read a record from a file must specify an access mode and, if appropriate, a key value identifying the desired record.

After a request for service has been processed, RMS-11 uses the same control block to return information to your program. When a file has been successfully opened, RMS-11 provides attribute information such as the organization of the file and the format of the records in the file. After successfully obtaining a record from a file, RMS-11 provides your program with the location in memory and length of the record retrieved.

The amount of information exchanged between RMS-11 and your program varies with the nature of the request and the attributes of the file being processed. Detailed information on the input to and output from each type of runtime processing macro is provided in Chapter 9 of this manual. This current chapter emphasizes only the general interface used by a program when requesting RMS-11 services. This interface is described in three sections:

- Runtime processing macros
- User control blocks
- File and record operations

Table 2-3
User Control Blocks

Block Name	Function
File Access Block (FAB)	Describes a file and contains file-related information.
Record Access Block (RAB)	Describes a record and contains record-related information.
Extended Attribute Blocks (XABs)	Contain file attribute information beyond that in the FAB.
Name Block (NAM)	Describes a location containing the expanded file specification resulting from the application of default values to a primary name string.

The subsections that follow describe each control block listed in Table 2-3.

2.2.1 The File Access Block (FAB)

At runtime, a File Access Block (FAB) represents a particular file. The fields of the FAB are used to contain such file-related information as:

- The name of file
- The organization of the file
- The operations your program will perform on the file
- The format of records within the file
- Record size information
- Allocation information

2.2.2 The Record Access Block (RAB)

A Record Access Block (RAB) is needed whenever individual records in a file are to be accessed.

A RAB describes an individual record. It is used to communicate information about that record between your program and RMS-11. Once a file has been opened, you will use the fields of the RAB to describe a record to be accessed.

2.2.3 Extended Attribute Blocks (XABs)

There are several types of Extended Attribute Blocks (XABs). Each type contains fields that represent one attribute of a file. These attributes supplement those attributes in a File Access Block. Specifically, there are Extended Attribute Blocks that describe:

- File creation and revision dates
- Primary and alternate key definitions for indexed files
- File protection specification
- Allocation information

You may use Extended Attribute Blocks when you:

1. Create a new file.
2. Request that RMS-11 transmit the extended attributes of an existing file to your program.

In the first instance, your program uses XABs to pass file definition information to RMS-11. In the second instance, RMS-11 requires XABs in order to pass attribute information to your program.

2.2.4 The Name Block (NAM)

A Name Block (NAM) is an optional user control block. It is used to contain the full file specification resulting from the merger of explicit file name information with program- and system-provided defaults.

2.3 FILE AND RECORD OPERATIONS

To obtain any RMS-11 service, your program uses a combination of a runtime processing macro call and a user control block. The primary argument of every runtime processing macro, therefore, is the address of a user control block.

In the following sections, the division of RMS-11 services into file and record operations is continued to show the relationship of runtime processing macros with particular user control blocks. For each type of operation, the relationship among control blocks is described.

2.3.1 File Operations

The primary argument of a file processing macro call is the address of a FAB. The macro call itself represents the type of file service requested (e.g., $OPEN, $DISPLAY, $CLOSE, etc.) and the FAB identifies a specific file associated with the request. The combination of macro call and FAB results in a file operation.

Since each FAB represents a single file, your program must contain one FAB for each file that is open simultaneously.

You can optionally associate a Name Block (NAM) with a FAB. This association involves setting the address of the NAM in a data field of the FAB before the file is opened. While opening the file, RMS-11 places the results of the merger of explicit and default file specification information in an area described by the Name Block.

Your program can also associate Extended Attribute Blocks (XABs) with a File Access Block. The presence and purpose of associated XABs are related to whether a new file or an existing file will be processed. In both instances, the presence of associated XABs is indicated by the address of the first such block in a data field of the FAB. When multiple XABs for the same file are present, they are chained together through address fields in the XABs themselves.

2.3.1.1 **New Files and Extended Attribute Blocks** - You create a new RMS-11 file through a combination of the $CREATE macro call and a File Access Block. You will use the FAB to pass to RMS-11 a description of the primary attributes of the file, such as the file's organization and the format and size of the records the file will contain. However, there are no fields in a FAB that allow you to specify optional file attributes such as a protection specification, nor does the FAB allow you to define keys for an indexed file. Therefore, you will use XABs to pass to RMS-11 descriptions of file attributes beyond those contained in the FAB.

2.3.1.2 **Existing Files and Extended Attribute Blocks** - Extended attribute blocks are never used to define or alter the attributes of an existing file. The attributes of a file are fixed at the time you create the file. Thereafter, these attributes cannot be altered. However, there are two occasions when XABs are associated with a FAB that represents an existing file:

1. Your program contains a $DISPLAY macro.

2. You wish automatically to obtain extended file attributes as additional outputs of the $OPEN macro.

When your program issues a $DISPLAY macro call, RMS-11 retrieves the address of an XAB from a field in the FAB associated with the call. This XAB may be the first of a chained list of such blocks. Further, each block is self-identifying. That is, a field in the block specifies the type of information the block can contain, e.g., key definition, file protection specification, etc. Based on the types of XABs present, RMS-11 obtains the specified attribute information from the file and stores it in the appropriate XABs.

When your program issues an $OPEN macro call, RMS-11 examines the FAB associated with the call for the address of an XAB. If this block (possibly a chain of blocks) is present, RMS-11 automatically returns the appropriate attribute information into the fields of the block.

2.3.2 Record Operations

The primary argument of a record processing macro (e.g., $GET, $PUT, etc.) is the address of a Record Access Block. The macro call represents the type of record service requested and the RAB identifies a record associated with the request. The combination of macro call and RAB results in a record operation.

Once a file has been opened by a $OPEN or $CREATE file operation, your program must activate a record access stream before performing record operations. After a record access stream has been activated, your program can specify a record for access through the use of three RMS-11 access modes. The following subsections, therefore, describe:

- Record access streams
- Specifying a record for access

2.3.2.1 **Record Access Streams** - A record access stream is the association of a RAB with a FAB. This association occurs through the issuance of a $CONNECT macro call. Once this association has been established, your program can process records in the file represented by the FAB. When processing a relative or indexed file, you can associate more than one RAB with the same FAB. Each association represents an independent record access stream to the same file. For example, your program could access records within an indexed file by primary key while, through a second record access stream, accessing records within the same file through the index associated with an alternate key.

At any point in time, a particular RAB can be associated with only a single FAB. The number of RABs required by a program, therefore, depends on the maximum number of record access streams active simultaneously.

2.3.2.2 **Specifying a Record for Access** - The organization of a file establishes the techniques (called access modes) that can be used to specify records for access. The organization of a file is fixed at the time the file is created. The access mode used to process records in a file, however, can be different each time the file is opened. Further, your program can dynamically switch from one access mode to another during the runtime processing of a file.

RMS-11 supports three record access modes:

1. Sequential
2. Random
3. Record's File Address (RFA)

The following subsections describe each access mode.

2.3.2.2.1 **Sequential Access Mode** - When using sequential access mode, your program issues a series of requests for the next record. RMS-11 interprets these requests in the context of the organization of the file being processed. Thus, the order in which records are read or written is governed by the structure, or organization, of the file.

2.3.2.2.2 **Random Access Mode** - In random access mode, your program, rather than the organization of the file being accessed, determines the order in which records are processed. Each program request for a record specifies, through fields in the RAB, the identification of the record of interest. Thus, when using random access mode, your program does not read or write the next record. Rather, your program identifies a particular record. The identifier associated with the request allows RMS-11 to locate the record within the file. The random access mode cannot be used with sequentially organized files. Both the relative and indexed file organizations, however, permit random access to records.

2.3.2.2.3 **Record's File Address (RFA) Access Mode** - Record's file address (RFA) access mode is similar to random access mode in that it allows a specific record to be identified for retrieval. It can be used with any file organization so long as the file resides on a disk device. It cannot, however, be used for write operations.

The term record's file address is meant to convey the notion that every record within a file has a unique address. The actual format of this address depends on the organization of the file. In all instances, however, only RMS-11 can interpret this format.

The most important feature of record's file address access is that the RFA of any record remains constant while the record exists in the file. After every successful $GET, $PUT, or $FIND operation, the RFA of the desired record is returned by RMS-11 in a field of the Record Access Block associated with the operation. Your program can then save this RFA to be used again later to retrieve the same record. It is not required that an RFA be used for subsequent retrieval only during the current execution of the program. During a file's existence, RFA's can be used at any subsequent point in time.

CHAPTER 3

DECLARING RMS-11 FACILITIES

Every program that processes RMS-11 files must contain directives and special-purpose macros that declare the RMS-11 facilities required at assembly-time and at runtime. This chapter describes these directives and macros.

To declare RMS-11 facilities that are used by your program, you must do the following:

1. List RMS-11 macros in .MCALL directives.
2. Declare the processing environment.
3. Declare space pool requirements.
4. Issue an $INIT or an $INITIF macro.

The sections of this chapter describe each of these requirements.

NOTE

Unless otherwise noted, RMS-11 macros use decimal as the default radix for numeric values.

3.1 .MCALL DIRECTIVE - LISTING NAMES OF REQUIRED MACRO DEFINITIONS

All macro calls issued in your program must be listed as arguments in an .MCALL directive. Listing the macro calls in this way allows the corresponding macro definitions to be read in from macro libraries during assembly.

Each .MCALL directive takes the following form:

```
.MCALL    arg1,arg2,...,argn
```

where

arg1,etc. represents a list of symbolic names of the macro definitions required in the assembly of the program. Macro names may be listed in any order.

The number of .MCALL directives needed for RMS-11 macros can be minimized. Table 3-1 lists a set of macro names. The definitions in the system macro library for most of these macros contain embedded .MCALL directives. These embedded .MCALL directives list as arguments the names of additional RMS-11 macros. Thus, by providing .MCALL

directives with the names of the macros in Table 3-1, you effectively can provide .MCALL directives for any and all RMS-11 macros used in a program.

Table 3-1
Minimum Set of .MCALL Directives

User Supplied .MCALL Argument	Embedded .MCALL Arguments
ORG$	(none)
POOL$B	Space pool declaration macros (described in Section 3.3).
$INIT or $INITIF	(none)
$GNCAL	Runtime field manipulation macros (described in Chapter 4) and completion routine macros (described in Chapter 9).
FAB$B	File Access Block allocation and initialization macros (described in Chapter 5).
RAB$B	Record Access Block allocation and initialization macros (described in Chapter 6).
XAB$B	Extended Attribute Block allocation and initialization macros (described in Chapter 7).
NAM$B	Name Block allocation and initialization macros (described in Chapter 8).
$FBCAL	File processing macros (described in Chapter 9).
$RBCAL	Record processing macros (described in Chapter 9).

As shown in Table 3-1, you can ensure that all RMS-11 macros used in a program appear as arguments in .MCALL directives by coding the following sequence of .MCALL directives and macro calls.

```
.MCALL  ORG$,POOL$B,$INIT
.MCALL  $GNCAL,FAB$B,RAB$B,XAB$B,NAM$B
.MCALL  $FBCAL,$RBCAL
$GNCAL
$FBCAL
$RBCAL
```

In the preceding example, the $GNCAL, $FBCAL and $RBCAL macro calls are issued after being listed in .MCALL directives. By issuing these macro calls, you will cause the embedded .MCALL directives to take effect. Naturally, you can omit any macro names from .MCALL directives that do not apply to a particular program. If Name Blocks or Extended Attribute Blocks are not used, for example, there would be no need for listing the NAM$B or XAB$B macros. Further, you may choose not to use the $RBCAL or $FBCAL macros, for example, but rather

to list separately each record processing and file processing macro actually appearing in your program.

3.2 ORG$ - DECLARING THE PROCESSING ENVIRONMENT

You must include one or more ORG$ macros within the set of modules that you will link together through the Task Builder to produce an executable task.

NOTE

All ORG$ macros must be in modules that are part of the root of your task. An ORG$ macro for a particular file organization must be present even if no record operations are performed when such a file is opened.

The presence of ORG$ macros in your source modules allows the Task Builder to select for linking only those portions of RMS-11 actually required by your program. Each ORG$ macro declares a unique combination of file organization and record operations.

The ORG$ macro takes the following form:

```
ORG$ org[,<recop[,recop...]>]
```

where

org — is the type of file organization to be processed. One of the following symbolic values must be specified:

- IDX - indexed file organization
- REL - relative file organization
- SEQ - sequential file organization

recop — is a symbolic value identifying a type of operation that will be performed on a file of the specified organization. If a single value is specified, the angle brackets are not needed. If multiple values are specified, you must enclose them in angle brackets and use commas to separate each value from the preceding value. One or more of the following may be specified in any order:

- CRE - indicates a file of the specified organization may be created
- DEL - indicates $DELETE operations
- FIN - indicates $FIND operations
- GET - indicates $GET operations
- PUT - indicates $PUT operations
- UPD - indicates $UPDATE operations

The following is an example of the use of ORG$ macros in a source module:

```
ORG$  SEQ,<CRE,PUT>
ORG$  IDX,<GET,UPD,FIN>
ORG$  REL
```

In this example, the user declares that one or more sequential files will be created and $PUT operations performed. One or more indexed files will be opened and $GET, $UPDATE, and $FIND operations will be performed on such files. Finally, one or more relative files will be opened but no record operations will be performed (possibly such files will be opened only for the purpose of issuing an $EXTEND or $DISPLAY file operation).

3.3 DECLARING SPACE POOL REQUIREMENTS

RMS-11 requires a collection of I/O buffers and internal control structures to support file processing at runtime. The area in your program occupied by these buffers and control structures is known as the space pool. RMS-11 provides facilities that ensure that the space pool is large enough to accommodate only the requirements of the largest number of files that can be open simultaneously. By using these facilities at assembly-time, your program provides information that allows RMS-11 to calculate the minimum size requirements of the space pool.

The major portion of the space pool is composed of I/O buffers. To the user program, record processing under RMS-11 appears as the movement of records directly between a file and the program itself. Transparent to the user program, however, RMS-11 actually reads and writes either virtual blocks or buckets into I/O buffers. When the organization of a particular file is sequential, RMS-11 reads or writes virtual blocks. For relative and indexed files, RMS-11 reads or writes buckets.

The size of I/O buffers depends on the organizations of the files being processed, the number of files open simultaneously, and the number of simultaneously active record access streams. In providing the information needed to calculate the size requirements for the I/O buffers portion of the space pool, you have two choices:

1. A completely centralized space pool.

2. Private I/O buffers for one or more files.

In a completely centralized space pool, all I/O buffers as well as the internal control structures required for file processing are inaccessible to your program. RMS-11 totally manages the space within the pool and allocates portions, as needed, for buffer space and control structures for open files.

Unlike a completely centralized space pool, the use of private I/O buffers allows you some measure of control over I/O buffer space. You can allocate private I/O buffers on a per-file basis by specifying the address and total size of these buffers in fields of the File Access Block associated with a file. When the file is open, this buffer space is completely managed by RMS-11 and your program must not access it. However, when the file is closed, the private I/O buffer space is available for use by your program.

The major advantage of private I/O buffers is avoidance of fragmentation of a completely centralized space pool. Since particular files have varying buffer requirements based on their organization, a centralized space pool can reach the point where there is sufficient total space available for the opening of an additional file but the space is not contiguous. When such a situation arises, the desired file cannot be opened.

Whether you choose a completely centralized space pool or private I/O buffers, RMS-11 always requires certain internal control structures that must be allocated in the space pool to support file processing. Unlike the handling of I/O buffers, your program can never access these control structures or recover the space they occupy.

The number of internal control structures required by RMS-11 in the space pool is based on the organizations of the files being processed, the maximum number of files open simultaneously, and the maximum number of simultaneously connected record access streams. Once again, your program must provide, at assembly-time, the information needed to determine the size requirements of the internal control structures that must be allocated in the space pool.

The presence in your source modules of the macros listed in Table 3-2 allows RMS-11 to determine the size requirements for your program's space pool. The descriptions following the table identify those macros that are always required and those that are needed only in specific instances. If you want private I/O buffers for one or more files, you must also refer to the descriptions of the BPA (buffer pool address) and BPS (buffer pool size) fields in Chapter 5.

Table 3-2
Space Pool Declaration Macros

Macro	Description
POOL$B	Beginning of space pool declaration
P$BDB	Number of buffer descriptor blocks
P$FAB	Number of files open simultaneously
P$RAB	Non-indexed record access streams active simultaneously
P$RABX	Indexed record access streams active simultaneously
P$IDX	Number of defined keys
P$BUF	Input/output buffer requirements
POOL$E	End of space pool declaration

3.3.1 POOL$B/POOL$E - Space Pool Declaration

The POOL$B macro is required. It marks the beginning of a sequence of space pool definition macros. This macro takes the form:

```
POOL$B
```

The POOL$E macro is also required. It marks the end of a sequence of space pool declaration macros. It takes the form:

```
POOL$E
```

The remaining macros listed in Table 3-2 can be coded in any order following the POOL$B macro and before the POOL$E macro. Multiple instances of such space pool declarations can occur among the set of source modules that you will link together through the use of the Task Builder. The Task Builder will sum the size requirements indicated by all such space pool declarations.

3.3.2 P$BDB - Number of Buffer Descriptor Blocks

The P$BDB macro is required. It ensures that the space pool contains sufficient space for internal RMS-11 control structures known as buffer descriptor blocks.

The format of this macro is:

```
P$BDB bdbnum
```

where

bdbnum — is a numeric value or symbol representing the number of buffer descriptor blocks required to support the file processing performed by your program. To determine this value, you must apply the following formula:

```
bdbnum = maxbuf + maxrel + (2 * maxidx)
```

where

maxbuf — is the maximum number of I/O buffers ever in use for simultaneously open files. You calculate this value by totaling the multi-buffer counts in the MBF fields of RABs for all combinations of simultaneously connected record access streams. The maximum value among all such combinations is the desired maxbuf value.

maxrel — is the maximum number of record access streams ever connected simultaneously for write operations to relative files (whether or not an actual write operation is performed).

maxidx — is the maximum number of record access streams ever active simultaneously for write operations to indexed files (whether or not an actual write operation is performed).

3.3.3 P$FAB - Number of Files Open Simultaneously

The P$FAB macro is required. It ensures that there will be sufficient storage in the space pool for internal RMS-11 control structures related to File Access Blocks. The format of this macro is:

```
P$FAB number
```

where

number	is a numeric value representing the maximum number of files that can be open simultaneously at run time.

An example of the P$FAB macro follows:

```
POOL$B
  .
  .
  .
P$FAB 4
  .
  .
  .
POOL$E
```

In this example, the user specifies that a maximum of four files can be open simultaneously at run-time.

3.3.4 P$RAB - Non-indexed Record Access Streams

The P$RAB macro ensures that there will be sufficient storage in the space pool to accommodate internal control structures related to Record Access Blocks for relative or sequential files. This macro can be omitted, therefore, if no record operations are performed on relative or sequential files.

The format of the P$RAB macro is as follows:

```
P$RAB rabs
```

where

rabs	is a numeric value representing the maximum number of Record Access Blocks that will be connected simultaneously for relative and sequential files.

In the following example, the user declares that a maximum of three Record Access Blocks, representing access streams to non-indexed files, will be connected simultaneously.

```
POOL$B
  .
  .
  .
P$RAB 3
  .
  .
  .
POOL$E
```

3.3.5 P$RABX - Indexed Record Access Streams

The P$RABX macro ensures that there will be room in the space pool for internal RMS-11 control structures related to indexed files. This macro is required, therefore, if the program accesses records in any indexed file. The format of this macro is as follows:

```
P$RABX rabs,ksize,ckeys
```

where

rabs	is a numeric value representing the maximum number of Record Access Blocks that will be connected simultaneously for indexed files.
ksize	is a numeric value representing the size (in bytes) of the largest key field within all the files represented by the first argument.
ckeys	is a numeric value representing the number of keys that can change when a $UPDATE operation is performed on an indexed file. This value must be specified whenever an indexed file is accessed for $UPDATE operations (i.e., the FAC field in the FAB for the associated file contains FB$UPD), whether or not a $UPDATE is actually performed.

The following is an example of the P$RABX macro:

```
POOL$B
  .
  .
  .
P$RABX 1,32
  .
  .
  .
POOL$E
```

In this example, the user specifies that there will be at most a single Record Access Block connected for processing an indexed file at any point during program execution. The size of the largest key field in any such file is 32 bytes and no keys can change during a $UPDATE operation.

3.3.6 P$IDX - Number of Defined Keys

The P$IDX macro reserves storage in the space pool for control structures containing internal key summary information. This macro is required if any indexed file is accessed by your program. Its format is as follows:

```
P$IDX keys
```

where

keys	is a numeric value representing the total number of all keys defined for all indexed files opened simultaneously. This total must be specified even if certain keys within one or more files are never used for retrieval operations.

The following is an example of the P$IDX macro:

```
POOL$B
  .
  .
  .
P$IDX 3
  .
  .
  .
POOL$E
```

In this example, the user declares that a total of three keys are defined among all indexed files that are open simultaneously.

3.3.7 P$BUF - I/O Buffers

The P$BUF macro ensures that the space pool is large enough for I/O buffers required to support the file and record processing performed by your program. If you choose to allocate private I/O buffers for all files, this macro can be omitted. If, however, one or more files will be processed without associated private I/O buffers, this macro is required.

The amount of buffer space required depends on the following:

- The maximum number of simultaneously active record access streams.
- The bucket size or block size of the file associated with each stream.
- The multi-block count (MBC) specified in the Record Access Blocks representing streams connected to disk sequential files.
- The multi-buffer count (MBF) specified in each Record Access Block representing each stream.

To calculate buffer space requirements, you employ the following formula:

```
buffsize = stmsize1[+stmsize2...+stmsizen]
```

where

stmsize1, stmsize2, etc. are the I/O buffer space requirements (in bytes) for each simultaneously active record access stream associated with a file without private I/O buffers. The requirements of each such stream are determined as follows:

FOR DISK FILES

```
stmsize=BKS*(512*MBC)*MBF
```

where

BKS is the number of virtual blocks in the largest size bucket of the file associated with the stream. For sequential files, BKS equals 1 in this formula.

MBC is the value contained in the multi-block count (MBC) field of the Record Access Block associated with the stream. This value is used only with sequential files. For non-sequential files, MBC equals 1 in this formula.

MBF is the value contained in the multi-buffer count (MBF) field of the Record Access Block associated with the stream.

FOR MAGNETIC TAPE FILES

stream-size=BLS*MBF

where

BLS is the size (in bytes) of each physical block of the file associated with the stream. The default block size for magnetic tape files is 512 bytes.

MBF is the value contained in the multi-buffer count (MBF) field of the Record Access Block associated with the stream.

The format of the P$BUF macro is as follows:

```
P$BUF buffsize
```

where

buffsize is a numeric value representing the total size (in bytes) of the I/O buffers required to support files without private I/O buffers. The specified value must be a multiple of 4.

In the following example of the P$BUF macro, the user specifies that a total of 8192 bytes of I/O buffer space is required:

```
POOL$B
  .
  .
  .
P$BUF 8192
  .
  .
  .
POOL$E
```

3.4 $INIT OR $INITIF - INITIALIZING THE RMS-11 SYSTEM

You must include either an $INIT or an $INITIF macro in the initialization code of any program that uses RMS-11 facilities. When encountered at runtime, the $INIT macro call attempts the initialization of the RMS-11 system. The $INITIF macro performs initialization if RMS-11 is not presently initialized. RMS-11 uses the C-Bit of the Processor Status Word to indicate success or failure

of the initialization procedure. If the C-Bit is cleared, initialization was successful.

The formats of these macros are:

1. label:$INIT

2. label:$INITIF

where

label — is an optional user-specified symbol that allows control to be transferred to this location during program execution. Other instructions in the program may refer to this label, as in the case of a program that has been written so that it can be restarted.

NOTE

The $INIT macro will not cause initialization if any files are open.

CHAPTER 4

ACCESSING CONTROL BLOCK FIELDS AT RUN-TIME

This chapter describes RMS-11 runtime macros that retrieve, modify, and test the contents of data fields in user control blocks.

Runtime field access macros expand into code that affects the contents of data fields during execution of your program. Using one or more of these macros, you can perform any of the following actions during program execution:

- Store values into control block data fields before the control block is used for the first time. You will perform this action frequently since there are no runtime defaults for any fields in control blocks.
- Alter the contents of a control block data field to suit the logic of your program, e.g., dynamically changing the access mode used to process a file from random to sequential.
- Test or compare the contents of control block data fields returned by RMS-11 to your program, e.g., the status field (STS) of a Record Access Block or File Access Block.

The runtime macros that perform the preceding functions are listed in Table 4-1. RMS-11 limits all but two of these macros to use with 1 byte or 1 word fields. The following table, therefore, also includes the size of control block data fields that can be accessed by the specified macro.

Table 4-1
Runtime Field Access Macros

Macro Name	Field Size	Function
$COMPARE	1 byte or 1 word	Compares the contents of a field with a user-specified value.
$FETCH	Any size	Copies the contents of a field into a user-specified location.
$OFF	1 byte or 1 word	Resets one or more bits within a bit string field.
$SET	1 byte or 1 word	Sets one or more bits within a bit string field.
$STORE	Any size	Changes the contents of a field.
$TESTBITS	1 byte or 1 word	Tests one or more bits within a field.

The macros in Table 4-1 are provided so that you need not be aware of the specific placement (and, to a large extent, size) of the fields within RMS-11 control blocks. The placement of these fields may differ from release to release. There are instances, however, when knowledge of placement is desirable (e.g., during debugging). This information can be derived from the symbol table of an assembly listing file of any module containing the control block(s) of interest. Offset values are represented by symbols beginning with the two characters 'O$'. For one-word and one-byte fields, the offset symbol is simply the concatenation of 'O$' with the three-character field name. For example, O$STS represents the offset (in bytes) of the STS field from the beginning of a FAB or RAB. For multi-word fields (such as ALQ in the FAB or in an allocation XAB) and multi-byte fields (such as SIZ in the key definition XAB), each word (or byte) is represented by an offset symbol that is the concatenation of 'O$', plus the 3-character field name, plus a digit in the range 0 to n, where n is 1 less than the number of words (or bytes) in the field. For example, the ALQ field is represented by the offset symbols O$ALQ0 (the less significant word) and O$ALQ1 (the more significant word); the SIZ field is represented by the symbols O$SIZ0 (the size of the first, most significant, key segment), O$SIZ1, ..., O$SIZ7 (the size of the last, least significant, key segment).

The sections that follow describe each of the macros listed in Table 4-1. The 3-character names of fields in control blocks and the symbolic values that can be used to test and set these fields are specified in Chapters 5 through 8.

NOTES

1. Octal radix is assumed for all numeric values used as operands in the macros described in this chapter. You can indicate decimal radix through the use of an explicit decimal point following a numeric value.

2. In all instances in which a control block field is 2 words in length and contains a numeric value, the least significant bits appear in the first word of the field and the most significant bits in the second word.

4.1 $COMPARE - COMPARING THE CONTENTS OF A FIELD

The $COMPARE macro compares a 1-byte or 1-word control block data field with a user-specified value and sets PDP-11 condition codes.

The format of the $COMPARE macro is as follows:

```
label:$COMPARE   source,fnm,reg
```

where

label — is an optional user-specified symbol referring to the $COMPARE macro.

source — is a user-specified operand representing a value to be compared with the contents of a control block data field. You may express this operand using any valid addressing mode. If the operand is specified as #0, a TST (or TSTB) instruction is generated and condition codes set accordingly. The operand must be word aligned for comparison with 1-word data fields.

fnm — is the 3-character name of a 1-byte or 1-word data field within a user control block. The assembler will generate an error message if the specified field name represents a multi-word field. The user-specified source operand will be compared with the contents of this field.

reg — is a general register, R0 through R5, loaded with the address of the user control block containing the desired data field.

The following are examples of the $COMPARE macro:

```
$COMPARE  #SU$SUC,STS,R1

$COMPARE  2(R1),RSZ,R5
```

In the first example, the user compares the status field (STS) of a control block with the symbolic value SU$SUC. The address of either a File Access Block or Record Access Block (both contain a status field) is in general register 1. In the second example, the user specifies that the record size field (RSZ) is to be compared with the operand expressed using indexed addressing mode. Since the RSZ field exists only in Record Access Blocks, general register R5 must contain the address of a RAB.

4.2 $FETCH - COPYING THE CONTENTS OF A FIELD

The $FETCH macro copies the contents of a control block data field into a user-specified location. This macro can be used to access any data field, regardless of size.

The format of the $FETCH macro is as follows:

```
label:$FETCH   destination,fnm,reg
```

where

label — is an optional user-specified symbol referring to the $FETCH macro.

destination — is a location within the user program into which the contents of a control block field are to be copied. The following restrictions apply to this operand:

1. You cannot use immediate mode or any form of deferred addressing mode.

2. If the field name specified as the second argument is POS or SIZ (refer to description of key definition XAB$ in Chapter 7), you cannot use register mode addressing to express the destination operand.

3. For multi-word fields other than POS and SIZ, you must use care when expressing the destination operand with register mode addressing. The expanded code of the $FETCH macro will use successive registers as destination operands for successive words of the data field. Depending on the length of the data field and the register specified as the destination operand, this code could use the register containing the control block address as a destination operand.

4. If the data field to be copied is one or more words in length, the specified destination location must be word aligned.

fnm — is the 3-character name of any data field within a user control block. Regardless of length, the contents of this field are copied to the user-specified location.

The following conventions apply if the $FETCH macro is used to reference the key size (SIZ) or key position (POS) fields of a key definition XAB (refer to Chapter 7 for a description of key definition XABs):

- You specify the 3-character name, SIZ or POS, to access the entire 8-element array. The following example shows all eight words of the POS field copied into successive locations beginning with the user-specified destination:

```
$FETCH (R0)+,POS,R3
```

- To access a single element in the 8-element array, you specify the 3-character field name immediately followed by a numeric element number from 0 to 7. In the following example, the user fetches the first element of the POS field:

```
$FETCH R4,POS0,R3
```

reg — is a general register, R0 through R5, loaded with the address of the user control block containing the desired data field.

The following are examples of the $FETCH macro:

```
$FETCH   R2,RBF,R4

$FETCH   8.(R3),MRN,R1
```

In the first example, general register R4 contains the address of a Record Access Block. The user copies the contents of the record address field (RBF) into general register R2. In the second example, general register R1 contains the address of a File Access Block. The user copies both words of the maximum record number field (MRN) into successive words beginning with the specified location.

4.3 $OFF - RESETTING BITS WITHIN A FIELD

The $OFF macro resets one or more bits within 1-byte or 1-word bit string data fields.

The format of the $OFF macro is as follows:

```
label:$OFF   value,fnm,reg
```

where

label	is an optional user-specified symbol referring to the $OFF macro.
value	is an expression or location specifying the bits within the data field that are to be reset.
fnm	is the 3-character name of a 1-byte or 1-word data field within a user control block. The assembler will generate an error message if the specified field name represents a multi-word field.
reg	is a general register, R0 through R5, loaded with the address of the user control block containing the desired data field.

In the following example, general register R2 contains the address of a Record Access Block. The user resets the bit in the record options field (ROP) that specifies greater than or equal key searches.

```
$OFF   #RB$KGE,ROP,R2
```

4.4 $SET - SETTING BITS WITHIN A FIELD

The $SET macro sets one or more bits within 1-byte or 1-word bit string data fields.

The format of the $SET macro is as follows:

```
label:$SET   value,fnm,reg
```

where

label	is an optional user-specified symbol referring to the $SET macro.
value	is an expression or location specifying the bits within the data field that are to be set.
fnm	is the 3-character name of a 1-byte or 1-word data field within a user control block. The assembler will generate an error message if the specified field name represents a multi-word field.
reg	is a general register, R0 through R5, loaded with the address of the user control block containing the desired data field.

The following are examples of the $SET macro:

```
$SET   #FB$GET!FB$UPD,FAC,R4
$SET   #RB$EOF,ROP,R1
```

In the first example, general register R4 contains the address of a File Access Block. The user sets the bits within the file access field (FAC) that indicate $GET and $UPDATE operations will be performed on the associated file. In the second example, general register R1 contains the address of a Record Access Block. The user sets the bit within the record options field (ROP) that specifies positioning to end of file.

4.5 $STORE - CHANGING THE CONTENTS OF A FIELD

The $STORE macro changes the contents of an entire control block data field by storing values from a user-specified location. This macro can be used with any size data field. For multi-word fields, successive locations, beginning with the user-specified location, will be used as the source of values to be stored into the data field. For bit string data fields, the entire field will be altered. Therefore, if you want to change bit settings selectively in a bit string field, you should use the $OFF or $SET macro.

The format of the $STORE macro is as follows:

```
label:$STORE   source,fnm,reg
```

where

label — is an optional user-specified symbol referring to the $STORE macro.

source — is a location within the user program containing values to be stored into a control block data field. When this operand is expressed as #0, the expanded code of this macro will clear the entire data field to zero. The following restrictions apply to this operand:

1. You cannot use any form of deferred addressing mode.
2. Immediate mode addressing can be used only with 1-byte or 1-word fields.
3. If the field name specified as the second argument is POS or SIZ (refer to description of key definition XABs in Chapter 7), you cannot use register mode addressing to express the source operand.
4. For multi-word fields other than POS and SIZ, you must use care when expressing the source operand with register mode addressing. The expanded code of the $STORE macro will use successive registers as source operands for successive words of the data field. Depending on the length of the data field and the register specified as the source operand, this code could use the register containing the control block address as a source operand.
5. Word alignment of source operands is required when setting the contents of fields one or more words in length.

fnm — is the 3-character name of any data field in a user control block. The $STORE macro will change the contents of this field, regardless of its length.

The following conventions apply if the $STORE macro is used to change the contents of the key position (POS) or key size (SIZ) fields of a key definition XAB (refer to description of key definition XABs in Chapter 7):

- You specify the 3-character name, POS or SIZ, to change the contents of the entire 8-element array. The following example shows the changing of contents of all 8 words of the POS field with values taken from successive locations beginning with the user-specified source location:

```
$STORE   (R1)+,POS,R3
```

- To access a single element in the 8-element array, you specify the 3-character field name immediately followed by a numeric element number from 0 to 7. In the following example, the user changes the contents of the first element of the POS field:

```
$STORE   R2,POS0,R1
```

reg — is a general register, R0 through R5, loaded with the address of the user control block containing the desired data field.

The following are examples of the $STORE macro:

```
$STORE   #0,ALQ,R3

$STORE   #INPUT,FAB,R1
```

In the first example, general register R3 contains the address of a File Access Block. The user clears the 2-word allocation quantity field (ALQ) in the block to zero. In the second example, general register R1 contains the address of a Record Access Block. The user stores the address of a File Access Block in the FAB field of the Record Access Block.

4.6 $TESTBITS - TESTING BITS WITHIN A FIELD

The $TESTBITS macro compares one or more bits within a 1-byte or 1-word control block data field with a user-specified value and sets PDP-11 condition codes.

The format of the $TESTBITS macro is as follows:

```
label:$TESTBITS   source,fnm,reg
```

where

label — is an optional user-specified symbol referring to the $TESTBITS macro.

source — is an expression or location indicating which bits of the specified data field are to be tested.

fnm — is the 3-character name of a 1-byte or 1-word bit string data field containing the bits to be tested. The assembler will generate an error message if the specified field name represents a multi-word field.

reg — is a general register, R0 through R5, loaded with the address of the user control block containing the desired data field.

In the following example of the $TESTBITS macro, the address of a File Access Block is in general register R3. The user tests the file access (FAC) field of the block to determine if the current program can issue $UPDATE or $PUT operations:

```
$TESTBITS   #FB$UPD!FB$PUT,FAC,R3
```

If neither bit is set in the FAC field, condition code Z will be set, if either or both bits are set in SHR, code Z will be off.

CHAPTER 5

THE FILE ACCESS BLOCK

This chapter describes the File Access Block (FAB), the fields in a FAB, and the assembly-time macros that allocate FABs and initialize fields in FABs.

Certain conventions apply to the assembly-time macros presented in this chapter and in Chapter 6 (The Record Access Block), Chapter 7 (Extended Attribute Blocks), and Chapter 8 (The Name Block). For example, two macros are always required to allocate space for a user control block. These macros take the general form:

1. label:xyz$B

2. xyz$E

where

label — is the user-specified symbol that names this particular block.

xyz — is the three character name of the control block to be allocated (i.e., FAB, RAB, XAB, NAM).

An xyz$B macro causes the allocation of space for the specified block and delimits the beginning of an optional sequence of assembly-time initialization macros for the fields of the block. The xyz$E macro performs three functions:

1. Stores the values specified by intervening initialization macros in appropriate locations in the block.

2. Sets assembly-time default values in fields not explicitly initialized.

3. Terminates the definition of the block.

Each field within a user control block has a three character name. These three character names are always part of the name of the associated assembly-time initialization macros. The following is the format of initialization macros:

x$fnm arg

where

x — is the first character of the name of the block containing the field to be initialized, e.g., F for File Access Block, R for Record Access Block.

fnm	is the three character name of the field to be initialized.
arg	is the value to be loaded into the specified field.

The following are examples of assembly-time initialization macros:

```
F$FAC  arg
R$ROP  arg
```

The F$FAC macro initializes the file access field (FAC) of the File Access Block. The R$ROP macro initializes the record options field (ROP) of the Record Access Block.

Assembly-time initialization macros can appear only between the xyz$B and xyz$E macros that define and allocate the control block containing the field to be initialized.

The following are examples of the placement of field initialization macros:

```
INFILE:FAB$B
       F$FAC arg
       FAB$E

INP:   RAB$B
       R$ROP arg
       RAB$E
```

Depending on the nature of the field being initialized, three types of arguments may appear on initialization macros:

1. Symbolic values.
2. Labels.
3. Numeric values.

NOTES

1. You cannot use global symbols or labels as arguments to the assembly-time initialization macros described in this chapter. All symbols and labels used as arguments must be defined locally.
2. The default radix for numeric values in assembly-time initialization macros is decimal.

When symbolic values are provided by RMS-11, they take the following form:

```
xz$nam
```

where

xz	are the first and last characters of the associated three-character block name, e.g. FB for File Access Block, RB for Record Access Block.

nam — is a two- or three-character symbolic name representing the value to be loaded into the specified field.

The following are examples of symbolic values used as arguments in initialization macros:

```
INFILE:FAB$B
       F$FAC FB$PUT
       FAB$E

INP:   RAB$B
       R$ROP RB$EOF
       RAB$E
```

In the above examples, the file access field (FAC) of the File Access Block named INFILE is initialized with the symbolic value FB$PUT. This value indicates that the user requires write access to the file associated with the block. The record options field (ROP) of the Record Access Block INP is initialized with the value RB$EOF. This value indicates that the file is to be positioned to end of file.

When several symbolic values are used to initialize a field, each value must be separated by an exclamation point (!), as shown below:

```
INFILE:FAB$B
       F$FAC FB$GET!FB$PUT!FB$DEL
       FAB$E
```

The preceding example initializes the file access field (FAC) of the File Access Block INFILE with values indicating that the user requires read, write, and delete access to the contents of the associated file.

Certain fields in user control blocks can contain the address of another block or program work area. The argument used in initialization macros for these fields is the label specified by the user for the second block or program work area.

```
INFILE:FAB$B
       FAB$E
         .
         .
         .
INP:   RAB$B
       R$FAB INFILE
       RAB$E
```

The preceding example shows the File Access Block address field (FAB) of the Record Access Block INP initialized with the address of the File Access Block named INFILE.

Lastly, certain fields within structures may be initialized at assembly-time by macros that allow numeric values or user defined symbols as arguments:

```
1.  INFILE:FAB$B
           F$MRS 132
           FAB$E

2.  MAXSIZ=132.
             .
             .
             .
    INFILE:FAB$B
           F$MRS MAXSIZ
           FAB$E
```

In the preceding examples, the maximum record size field (MRS) of a File Access Block named OTFILE is initialized with a value of 132, representing the size, in bytes, of the largest record that will be written in the file.

5.1 ALLOCATING A FILE ACCESS BLOCK

The FAB$B macro allocates space for a File Access Block and delimits the beginning of an optional sequence of assembly-time initialization macros for the fields of the block.

The format of the FAB$B macro is as follows:

```
label:FAB$B
```

where

label	is a user-specified symbol that names this particular File Access Block. You must assure that the address assigned to this label is word-aligned. Therefore, a .EVEN directive should immediately precede the FAB$B macro.

The FAB$E macro delimits the end of an optional sequence of assembly-time initialization macros and stores any specified initial values in the appropriate fields of the block. At assembly-time, all fields not explicitly initialized are set to their default values. The FAB$E macro must always appear subsequent to an associated FAB$B macro, even when no intervening initialization macros are coded. This macro takes the form:

```
FAB$E
```

The following example shows the allocation of a File Access Block named MASTER:

```
        .EVEN
MASTER: FAB$B           ;ALLOCATE MASTER FAB
        FAB$E           ;END OF MASTER FAB
```

The following example shows the allocation of the same block with the assembly-time initialization of two fields -- the file access (FAC) and logical channel number (LCH) fields.

```
        .EVEN
MASTER: FAB$B           ;ALLOCATE MASTER FAB
        F$FAC  FB$PUT   ;ACCESS FOR $PUT OPERATIONS
        F$LCH  2        ;ACCESS FILE ON CHANNEL 2
        FAB$E           ;END OF MASTER FAB
```

5.2 FIELDS IN THE FILE ACCESS BLOCK

Table 5-1 summarizes the fields in the File Access Block:

Table 5-1
File Access Block Fields

Field Name	Field Size	Description
ALQ	2 words	Allocation quantity.
BID	1 byte	Block identifier.
BKS	1 byte	Bucket size.
BLN	1 byte	Block length.
BLS	1 word	Block size.
BPA	1 word	Private buffer pool address.
BPS	1 word	Private buffer pool size.
CTX	1 word	User context area.
DEQ	1 word	Default file extension quantity.
DEV	1 byte (bit string)	Device characteristics.
DNA	1 word	Default name string address.
DNS	1 byte	Default name string size.
FAC	1 byte (bit string)	File access.
FNA	1 word	File name string access.
FNS	1 byte	File name string size.
FOP	1 word (bit string)	File processing options.
FSZ	1 byte	Fixed control area size.
IFI	1 word	Internal file identifier.
LCH	1 byte	Logical channel number.
MRN	2 words	Maximum record number.
MRS	1 word	Maximum record size.
NAM	1 word	Name block address.
ORG	1 byte	File organization.
RAT	1 byte (bit string)	Record attributes.
RFM	1 byte	Record format.

(Continued on next page)

Table 5-1 (Cont.)
File Access Block Fields

Field Name	Field Size	Description
RTV	1 byte	Retrieval window size.
SHR	1 byte (bit string)	File sharing.
STS	1 word	Completion status code.
STV	1 word	Status value.
XAB	1 word	Extended attribute block pointer.

The subsections that follow describe the purpose of each of the fields listed in Table 5-1. RMS-11 provides macros that allow you to initialize most of these fields at assembly-time. These macros, when provided, are discussed at the conclusion of each field description.

5.2.1 ALQ - Allocation Quantity

The allocation quantity (ALQ) field is used with three file processing macros - $OPEN, $CREATE, and $EXTEND:

1. As output from the $OPEN macro call, RMS-11 sets the ALQ field to indicate the highest numbered virtual block currently allocated to the file.

2. Before creating a new file by issuing a $CREATE macro call, you can set the ALQ field of the FAB describing the file. The value you place in ALQ will be interpreted by RMS-11 as the number of virtual blocks to be contained in the initial extent of the file. If you set ALQ equal to zero, RMS-11 will determine the minimum number of virtual blocks that must be allocated for initial extent of the file. For sequential files, no blocks will be allocated. For relative and indexed files, the minimum number of virtual blocks is based on the attributes of the file.

3. Before extending a file by issuing a $EXTEND macro call, you must set ALQ equal to the number of virtual blocks to be added to the file.

NOTE

The function of the ALQ field during the $CREATE and $EXTEND macro calls is different from the preceding descriptions if allocation XABs are present during the operation. Refer to Section 7.6 in Chapter 7 for a description of allocation XABs and their effect on ALQ during $CREATE and $EXTEND operations.

The F$ALQ macro allows you to initialize the ALQ field at assembly-time. The format of this macro is as follows:

```
F$ALQ quantity
```

where

quantity — is a numeric value in the range of 0 to 16,777,215 representing a number of virtual blocks.

The following example of the F$ALQ macro shows an allocation quantity of 132 virtual blocks.

```
MASTER: FAB$B
          .
          .
          .
        F$ALQ 132
          .
          .
          .
        FAB$E
```

5.2.2 BID - Block Identifier

The block identifier (BID) field identifies the block as a FAB. This field, automatically set by the FAB$B macro, contains the value FB$BID. You must never alter this field.

5.2.3 BKS - Bucket Size

The bucket size (BKS) field is used only for relative and indexed files. When you open an existing relative or indexed file by issuing the $OPEN macro call, RMS-11 sets the BKS field to the defined size of buckets in the file. When, however, you wish to create a relative or indexed file, you can specify the size of buckets in the file by setting a value in this field before issuing the $CREATE macro call. For sequential files, RMS-11 ignores the BKS field during a $CREATE macro call and zeroes the field during a $OPEN macro call.

NOTE

The function of the BKS field during the $CREATE and $OPEN macro calls is different from the preceding description if allocation XABs are present during the operation. Refer to Section 7.6 in Chapter 7 for a description of allocation XABs and their effect on BKS during $CREATE and $OPEN operations.

In specifying a bucket size, you must be aware of the relationship between bucket size and record size. Since RMS-11 does not allow records to cross bucket boundaries, you should ensure that the number of virtual blocks per bucket conforms to one of the following formulas:

1. Indexed files with fixed length records:

```
Bnum=((Rlen+7)*Rnum)+15/512
```

where

Bnum — is the number of virtual blocks per bucket rounded up to the next higher integer. The result must be in the range of from 1 to 32.

Rlen — is the fixed record length.

Rnum — is the number of records that you want in each bucket.

2. Indexed files with variable length records:

```
Bnum=((Rmax+9)*Rnum)+15/512
```

where

Bnum — is the number of virtual blocks per bucket rounded up to the next higher integer. The result must be in the range of from 1 to 32.

Rmax — is the maximum size of any record in the file.

Rnum — is the number of records, of the maximum size, that you want in each bucket.

3. Relative files with fixed length records:

```
Bnum = ((Rlen+1)*Rnum)/512
```

where

Bnum — is the number of virtual blocks per bucket rounded up to the next higher integer. The result must be in the range of from 1 to 32.

Rlen — is the fixed record length.

Rnum — is the number of records that you want in each bucket.

4. Relative files with variable length records:

```
Bnum = ((Rmax+3)*Rnum)/512
```

where

Bnum — is the number of virtual blocks per bucket rounded up to the next higher integer. The result must be in the range of from 1 to 32.

Rmax — is the maximum size of any record in the file.

Rnum — is the number of records that you want in each bucket. Variable length records in a relative file bucket always occupy Rmax+3 bytes.

5. Relative files with VFC format records:

```
Bnum = ((Rmax+Fsiz+3)*Rnum)/512
```

where

Bnum	is the number of virtual blocks per bucket rounded up to the next higher integer.
Rmax	is the maximum size of the data portion of any VFC record in the file.
Fsiz	is the size of the fixed control area portion of the VFC records.
Rnum	is the number of records that you want in each bucket. VFC records in a relative file bucket always occupy Rmax+Fsiz+3 bytes.

The F$BKS macro allows you to initialize the BKS field at assembly-time. The format of this macro is as follows:

```
F$BKS bucket-size
```

where

bucket-size	is a numeric value, in the range of 0 to 32, representing the number of virtual blocks contained in each bucket of the file. The assembly-time default is 0. RMS-11 interprets a value of 0 as identical to a value of 1.

The following is an example of the F$BKS macro showing the specification of four virtual blocks per bucket:

```
MASTER: FAB$B
          .
          .
          .
        F$BKS 4
          .
          .
          .
        FAB$E
```

5.2.4 BLN - Block Length

The block length (BLN) field specifies the length of the FAB. This field, automatically set by the FAB$B macro, contains the value FB$BLN. You must never alter this field.

5.2.5 BLS - Block Size

The block size (BLS) field is used for files on magnetic tape. When you open an existing file on magnetic tape by issuing a $OPEN macro call, RMS-11 sets the BLS field to the size of the physical blocks in the file. When, however, you are creating a file on magnetic tape, you can specify the physical block size by setting a value in this field before issuing the $CREATE macro call.

The F$BLS macro allows you to initialize the BLS field at assembly-time. Its format is:

```
F$BLS block-size
```

where

block-size — is a numeric value representing the size (in bytes) of the physical blocks on a tape. This value must be either 0 or in the range of 18 to 8192 bytes. At runtime, a value of 0 is interpreted as the operating system default of 512 bytes. The assembly-time default is 0.

An example of the F$BLS macro follows:

```
MASTER: FAB$B
          .
          .
          .
        F$BLS 4096
          .
          .
          .
        FAB$E
```

In this example, the user specifies that each physical block in a magnetic tape file will be 4096 blocks.

NOTE

To allow data interchange with other DEC systems, you should specify a block size less than or equal to 512 bytes. To allow data interchange with non-DEC systems, you should specify a block size that is less than or equal to 2048 bytes.

5.2.6 BPA - Buffer Pool Address

The buffer pool address (BPA) field can contain the address of a private I/O buffer pool for all record access streams for the file represented by the FAB. If you wish to use a private I/O buffer pool, you must set the address of the pool in this field before issuing a $CREATE or $OPEN macro call for the file. If you do not want to use a private I/O buffer pool for the file, you must assure that this field is zero before you issue the $CREATE or $OPEN macro. If this field is zero at the time a $CREATE or $OPEN macro call is issued, RMS-11 allocates I/O buffers for the file from the centralized space pool (refer to Section 3.3.7 in Chapter 3).

The F$BPA macro allows you to initialize the BPA field at assembly-time. The format of this macro is as follows:

```
F$BPA address
```

where

address	is the symbolic address of the starting location of a private I/O buffer pool. This address must be on a double-word boundary. The assembly-time default for the BPA field is zero (i.e., no private I/O buffer pool).

In the following example, the user sets, at assembly-time, the address of a private I/O buffer pool in the File Access Block called MASTER.

```
        .EVEN
INBUF:  .BLKB 4096.
          .
          .
          .
MASTER: FAB$B
          .
          .
          .
        F$BPA INBUF
          .
          .
          .
        FAB$E
```

5.2.7 BPS - Buffer Pool Size

You use the buffer pool size (BPS) field to specify the size of a private I/O buffer pool. You specify this size only if you placed a valid address in the BPA field of the same FAB before issuing a $OPEN or $CREATE macro call.

The F$BPS macro allows you to initialize the BPS field at assembly-time. Its format is:

```
F$BPS poolsz
```

where

poolsz	is a numeric value representing the total size (in bytes) of the private I/O buffer pool. The specified value must be a multiple of 4.

To calculate the size of the private pool, you use the following formula:

```
poolsz = stmsize1[+stmsize2...stmsizen]
```

where

stmsize1, stmsize2, etc.	are the I/O buffer space requirements (in bytes) for each simultaneously active record access stream associated with the file. The requirements of each such stream are determined as follows:

FOR DISK FILES

stmsize=BKS*(512*MBC)*MBF

where

BKS is the number of virtual blocks in the largest size bucket of the file associated with the stream. For sequential files, BKS equals 1 in this formula.

MBC is the value contained in the multi-block count (MBC) field of the Record Access Block associated with the stream. This value is used only with sequential files. For non-sequential files, MBC equals 1 in this formula.

MBF is the value contained in the multi-buffer count (MBF) field of the Record Access Block associated with the stream.

FOR MAGNETIC TAPE FILES

stmsize=BLS*MBF

where

BLS is the size (in bytes) of each physical block of the file associated with the stream. The default block size for magnetic tape files is 512 bytes.

MBF is the value contained in the multi-buffer count (MBF) field of the Record Access Block associated with the stream.

In the following example, the user defines the address and size of a private I/O buffer pool:

```
        .EVEN
INBUF:  .BLKB 4096.
          .
          .
          .
MASTER: FAB$B
          .
          .
          .
        F$BPA INBUF
        F$BPS 4096
          .
          .
          .
        FAB$E
```

5.2.8 CTX - User Context Area

The user context area (CTX) is never used in any way by RMS-11. It is intended exclusively for the user. Therefore, you can set any value you choose in this 1-word field. You might, for example, use this field as a means of communicating with a common completion routine in your program.

The F$CTX macro allows you to initialize the CTX field at assembly-time. The format of this macro is as follows:

F$CTX argument

where

argument represents any user-selected value.

The following are two examples of the F$CTX macro, showing a numeric value placed in the user context area in the first case, and a symbolic value in the second case.

```
1.  MASTER: FAB$B
              .
              .
              .
            F$CTX 3
              .
              .
              .
            FAB$E

2.  MASTER: FAB$B
              .
              .
              .
            F$CTX INPUT
              .
              .
              .
            FAB$E
```

5.2.9 DEQ - Default File Extension Quantity

The default file extension quantity (DEQ) field contains the number of virtual blocks to be used when RMS-11 must automatically extend the file. This automatic extension occurs whenever your program attempts a $PUT or $UPDATE operation that cannot be accommodated within the space currently allocated to the file.

When you create a new file, you can specify a default extension quantity by setting the desired value in the DEQ field before issuing the $CREATE macro call. RMS-11 saves the specified value as a permanent attribute of the file. A value of zero indicates that the default extension quantity for the file is the volume default.

When you are processing an existing file, you can temporarily override the default extension quantity specified when the file was created. To do this, you set the desired value in the DEQ field before issuing the $OPEN macro call. When RMS-11 finds a non-zero value in this field during $OPEN processing, it uses the specified value for any automatic extension operations needed while the file is open. Once you close the file, the default extension quantity reverts to that specified at create time.

The F$DEQ allows you to initialize the DEQ field at assembly-time. Its format is:

F$DEQ quantity

where

quantity — is a numeric value representing a number of virtual blocks. This number must be in the range of from 0 to 65,535. For relative and indexed files, the quantity you specify should be a multiple of bucket size. The assembly-time default is 0.

The following example of the F$DEQ macros shows the user specifying a default extension of 80 virtual blocks.

```
MASTER: FAB$B
          .
          .
          .
        F$DEQ 80
          .
          .
          .
        FAB$E
```

5.2.10 DEV - Device Characteristics

When you open a file by issuing a $OPEN or $CREATE macro call, RMS-11 sets the device characteristics (DEV) field. This field allows RMS-11 to communicate to your program the generic characteristics of the device on which the file resides. After the file is open, you can use appropriate runtime field accessing macros (refer to Chapter 4) to test for the following characteristics:

FB$CCL	Carriage control device, e.g., printers and terminals.
FB$MDI	Multiple directory structured device, e.g., disk volumes.
FB$REC	Record oriented device, e.g., terminal, line printer, etc. All record oriented devices are considered sequential in nature.
FB$SDI	Single directory device, i.e., a master file directory is used but no user file directories are present.
FB$SQD	Sequential and block oriented device (magnetic tape).
FB$TRM	Terminal with both keyboard and printer.

5.2.11 DNA - Default Name String Address

The default name string address (DNA) field allows you to provide program defaults for any missing components of the file name string addressed by the FNA field. You can specify defaults for one or more of the following file specification components:

1. Device
2. Directory
3. Filename
4. File type
5. File version number

You can use the F$DNA macro to initialize the DNA field at assembly-time. The format of this macro is:

```
F$DNA address
```

where

address — is the symbolic address of an ASCII string representing one or more components of a file specification. The components (e.g., device, file type) must be specified in the order in which they would occur in a complete file specification string. If this field is zero, RMS-11 assumes that there is no default name string.

As an example of the use of a default name string and the F$DNA macro, assume that a user's program contains the following directive:

```
DFNAM:  .ASCII /SY:.DAT/
```

The following use of the F$DNA macro would associate the default name string with a particular File Access Block:

```
MASTER: FAB$B
          .
          .
          .
        F$DNA DFNAM
          .
          .
          .
        FAB$E
```

During a $OPEN or $CREATE macro call, RMS-11 uses the ASCII string whose address is DFNAM to provide the device (SY:) and file type (DAT) components of the file specification of the file to be accessed -- if these components are missing from the string whose address is stored in the FNA field.

5.2.12 DNS - Default Name String Size

The default name string size (DNS) field is used to specify the size of the default name string whose address is contained in the DNA field of the same File Access Block.

The F$DNS macro allows you to initialize the DNS field at assembly-time:

```
F$DNS size
```

where

size — is a numeric value representing the size of the default name string expressed in bytes. The specified value must be in the range of zero to 255. If this field is zero, RMS-11 assumes that there is no default name string.

Assume that a user's program contains the following directive:

```
DFNAM:  .ASCII /SY:.DAT/
```

The following example shows the use of the F$DNS macro and its relationship to the F$DNA macro:

```
MASTER: FAB$B
          .
          .
          .
        F$DNA  DFNAM
        F$DNS  7
          .
          .
          .
        FAB$E
```

5.2.13 FAC - File Access

You must ensure that the file access (FAC) field contains indications of operations you intend to perform on the file. After you open a file, RMS-11 rejects any of the seven operations listed below if that operation was not specified in the FAC Field during execution of the $OPEN or $CREATE macro call for the file.

```
$DELETE
$GET
$PUT
$TRUNCATE
$UPDATE
$READ (block I/O)
$WRITE (block I/O)
```

If, for example, a $DELETE record operation macro appears in your program for the file associated with this File Access Block, the symbolic value FB$DEL must be present in the file access field before you open the file.

The F$FAC macro allows you to initialize the FAC field at assembly-time. The format of this macro is as follows:

```
F$FAC operation[!operation...]
```

where

operation — is a symbolic value representing a type of file or record operation that may be performed on the file. One or more values may be listed in any order. Allowable values are:

FB$DEL	$DELETE operations.
FB$GET	$GET and/or $FIND operations (assembly-time default).
FB$PUT	$PUT operations. You must specify this value if you are creating a file.
FB$REA	$READ block I/O operations (refer to Appendix B for a description of block I/O).

FB$TRN $TRUNCATE operations. You should not specify this value unless the associated file is sequential.

FB$UPD $UPDATE operations.

FB$WRT $WRITE block I/O operations (refer to Appendix B for a description of block I/O).

The following are two examples of the F$FAC macro, indicating $GET operations for the associated file in the first case, and $DELETE, $GET, and $PUT operations in the second case.

1.
```
MASTER: FAB$B
          .
          .
          .
        F$FAC FB$GET
          .
          .
          .
        FAB$E
```

2.
```
MASTER: FAB$B
          .
          .
          .
        F$FAC FB$DEL!FB$GET!FB$PUT
          .
          .
          .
        FAB$E
```

5.2.14 FNA - File Name String Address

In combination with the file name string size field (FNS), the FNA field describes an ASCII string that represents the file specificaiton of the file associated with the File Access Block. If this string does not contain all the components of a full file specification, RMS-11 will use the defaults supplied in the default name string described by the DNA and DNS fields. If no default name string is present or if the file specification is still incomplete, RMS-11 will apply system defaults, if any, for the missing components.

The F$FNA macro allows you to initialize the FNA field at assembly-time. Its format is:

```
F$FNA address
```

where

address is the symbolic address of an ASCII string representing the file specification of the file associated with the File Access Block. If this field is zero, RMS11 assumes that there is no file name string.

The following is an example of the F$FNA macro:

```
 FLNAM: .ASCII /PAYROLL/
          .
          .
          .
MASTER: FAB$B
          .
          .
          .
        F$FNA FLNAM
          .
          .
          .
        FAB$E
```

5.2.15 FNS - File Name String Size

The file name string size (FNS) field contains the size of the ASCII string file specification whose address is in the FNA field of the block.

The F$FNS macro allows you to initialize the FNS field at assembly-time. The format of this macro is:

```
F$FNS size
```

where

size	is a numeric value representing the size (in bytes) of the file name string. The specified value must be in the range of zero to 255. If this field is zero, RMS-11 assumes that there is no file name string.

The following example shows the use of the F$FNS macro and its relationship to the F$FNA macro:

```
 FLNAM: .ASCII /PAYROLL/
          .
          .
          .
MASTER: FAB$B
          .
          .
          .
        F$FNA FLNAM
        F$FNS 7
          .
          .
          .
        FAB$E
```

5.2.16 FOP - File Processing Options

The file processing options (FOP) field contains indicators representing your requests for optional file handling operations. This field can contain one or more of the following values:

FB$CTG — indicates that, during a $EXTEND or $CREATE operation, RMS-11 is to allocate contiguously the amount of space specified in the allocation quantity field (ALQ). For an existing file, RMS-11 returns this value in the FOP field following a $OPEN if the corresponding file is contiguous.

FB$DLK — indicates that the file is not to be locked from further access if it is not closed in the normal manner.

FB$NEF — inhibits positioning to end of file when an ANSI tape file is opened and the file access field (FAC) contains the symbolic value FB$PUT.

FB$POS — indicates that the magnetic tape volume set should be positioned to immmediately after the most recently closed file when the file represented by this File Access Block is created. The FB$RWO value takes precedence over FB$POS. The default operation is to position to the end of the volume set.

FB$RWC — specifies that the magnetic tape volume set is to be rewound when the file is closed.

FB$RWO — indicates that the magnetic tape volume set is to be rewound before the file associated with this File Access Block is opened or created.

FB$SUP — causes any corresponding existing file to be superseded during a $CREATE macro call if an explicit version number is present in the full file specification of the file associated with this File Access Block.

FB$TMD — indicates that the file associated with the File Access Block is to be created as a temporary file and deleted when the file is closed. This value takes precedence over the value FB$TMP, if both are present.

FB$TMP — indicates that a temporary file is to be created and retained after closing. The file name will not be entered in the directory.

The F$FOP macro allows you to initialize the FOP field at assembly-time. Its format is:

```
F$FOP  option[!option...]
```

where

option — is a symbolic value representing a processing option to be applied to the file represented by the FAB. One or more of the options listed above can be specified.

In the following example of the F$FOP macro, the user requests that the magnetic tape volume is to be rewound when the file is closed.

```
MASTER: FAB$B
          .
          .
          .
        F$FOP FB$RWC
          .
          .
          .
        FAB$E
```

5.2.17 FSZ - Fixed Control Area Size

The fixed control area size (FSZ) field is used only when the records of the file represented by the FAB are in VFC format. When you create such a file, you can set the desired value in the FSZ field before issuing the $CREATE macro call. When you open an existing file that contains VFC format records, RMS-11 will set this field equal to value specified when the file was created.

You can initialize this field at assembly-time by using the F$FSZ macro. Its format is as follows:

```
F$FSZ size
```

where

size — is a numeric value, expressed in bytes, representing the size of the fixed control area of each record of the file. The specified value must be in the range of from 1 to 255. The default size is 2 bytes. If this field is zero, RMS-11 assumes that the default size is desired.

An example of the F$FSZ macro follows:

```
MASTER: FAB$B
          .
          .
          .
        F$FSZ 8
          .
          .
          .
        FAB$E
```

In this example, the user specifies that each record of the file (with VFC format records) contains an 8-byte fixed control area.

5.2.18 IFI - Internal File Identifier

The internal file identifier (IFI) field is used by RMS-11 to associate the File Access Block with a corresponding internal control structure in the space pool. RMS-11 sets this field during a $OPEN or $CREATE operation. It is cleared to zero during a $CLOSE macro call. Your program must never alter this field.

5.2.19 LCH - Logical Channel Number

The LCH field is used to indicate a logical channel number. Every File Access Block representing an open file must contain a unique logical channel number. All I/O operations performed on the file represented by a FAB will use this number.

You can initialize the LCH field at assembly-time by using the F$LCH macro. The format of this macro is as follows:

```
F$LCH channel
```

where

channel — is a numeric value from 1 to 255.

In the following example, the user assigns a logical channel number of 4 to the associated File Access Block:

```
MASTER: FAB$B
          .
          .
          .
        F$LCH 4
          .
          .
          .
        FAB$E
```

5.2.20 MRN - Maximum Record Number

The maximum record number (MRN) field is meaningful only for relative files. When creating a relative file, you can use this field to specify the highest numbered record that can ever be written into the file. Thereafter, if any program attempts to write a record whose relative number exceeds the specified limit, RMS-11 will return an illegal maximum record number error (ER$MRN). Conversely, if you do not want to limit the number of records that can be written into a relative file, you set the MRN field to zero before issuing the $CREATE macro for the file. If the MRN field contained a value of zero at the time the file was created, RMS-11 will not perform limit checks when records are written.

When you open an existing relative file by issuing a $OPEN macro call, RMS-11 returns to your program the MRN value specified when the file was created. If no MRN was specified, RMS-11 returns the default value.

You can initialize the MRN field at assembly-time by using the F$MRN macro:

```
F$MRN max-rec-num
```

where

max-rec-num — is a numeric value representing the highest numbered record that can be written into the file associated with the File Access Block. The assembly-time default is zero. RMS-11 converts a value of zero to the highest positive integer for the length of the field. It is this latter value that will be returned during a $OPEN operation if MRN was defaulted during a $CREATE operation.

The following are two examples of the F$MRN macro. In the first example, the user indicates that the highest numbered record that can be written into the file is record 10,000. In the second example, the user explicitly specifies that no limit checks are to be performed.

```
1.  MASTER: FAB$B
              .
              .
              .
            F$MRN 10000
              .
              .
              .
            FAB$E

2.  MASTER: FAB$B
              .
              .
              .
            F$MRN 0
              .
              .
              .
            FAB$E
```

5.2.21 MRS - Maximum Record Size

When you create a file you should set the MRS field before issuing the $CREATE macro. The value you set in this field is based on the format of records in the file you are creating.

Fixed format records - the specified value represents the actual size of each record in the file. When creating a new file with fixed format records, you must place a non-zero value in the MRS field.

Variable and stream format records - the specified value represents the size of the largest record that can be written into the file. A value of zero means no limit on record size. The assembly-time default value is zero. However, if the organization of a file is relative, you must create the file with a non-zero maximum record size, which establishes the size of the record cells in the file.

VFC format records - the specified value must not include the size of the fixed control area. Rather, the specified value represents only the remaining data portion of VFC records. Non-zero and zero values are interpreted by RMS-11 in the same manner as in variable format records.

When you open an existing file by issuing a $OPEN macro call, RMS-11 returns, in the MRS field, the maximum record size specified when the file was created.

The F$MRS macro allows you to initialize the MRS field at assembly-time. This macro takes the form:

```
F$MRS size
```

where

size	is a numeric value expressed in bytes, representing record size. This value must be in the range of 0 to 16,383.

In the following example, the user specifies a maximum record size of 1024 bytes:

```
MASTER: FAB$B
          .
          .
          .
        F$MRS 1024
          .
          .
          .
        FAB$E
```

5.2.22 NAM - Name Block Address

The Name Block address (NAM) field is always an optional field. You will place an address in this field only when you have allocated a NAM block in your program and want the facilities provided by such a block when you open a file (refer to Chapter 8 for a description of the purpose of a NAM block).

You can initialize the NAM field at assembly-time with the F$NAM macro. This macro takes the form:

```
F$NAM address
```

where

address	is the symbolic address of an optional name block associated with this File Access Block. A value of zero indicates no NAM block. The assembly-time default is zero.

An example of the F$NAM macro follows:

```
 NMBLK: NAM$B
          .
          .
          .
        NAM$E
          .
          .
          .
MASTER: FAB$B
          .
          .
          .
        F$NAM NMBLK
          .
          .
          .
        FAB$E
```

5.2.23 ORG - File Organization

The organization (ORG) field identifies the organization of the file represented by the File Access Block. When you open an existing file, RMS-11 will set this field. When you create a new file, your program must set this field before issuing the $CREATE macro call.

The F$ORG macro allows you to initialize the ORG field at assembly-time. Its format is:

```
F$ORG organization
```

where

organization — is a symbolic value representing the organization of the file. One of the following values must be specified:

- FB$IDX - indexed
- FB$REL - relative
- FB$SEQ - sequential (assembly-time default)

The following is an example of the F$ORG macro:

```
MASTER: FAB$B
          .
          .
          .
        F$ORG FB$IDX
          .
          .
          .
        FAB$E
```

In this example, the user specifies that the new file associated with the File Access Block is to have an indexed organization.

5.2.24 RAT - Record Attributes

The record attributes (RAT) field specifies characteristics of the records in the file. When you open an existing file, RMS-11 sets this field with the characteristics specified at create time. When you create a new file, your program sets desired values in this field before issuing the $CREATE macro call.

The F$RAT macro allows you to initialize the RAT field at assembly-time. It has the following format:

```
F$RAT attribute[!attribute]
```

where

attribute — is a symbolic value used to define an attribute of the records of the file. One or more values may be listed in any order. Allowable values are:

- FB$BLK — indicates that records will not cross block boundaries. This value can be specified for sequential files only.

FB$CR indicates that each record, when written to a line printer or terminal, is to be preceded by a line feed character and followed by a carriage return character.

FB$FTN specifies that the first byte of each record contains a FORTRAN carriage control character.

In the following example, the user specifies that records do not cross block boundaries and each record is to be preceded by line feed and followed by carriage return when written to a carriage control device:

```
MASTER: FAB$B
          .
          .
          .
        F$RAT FB$BLK!FB$CR
          .
          .
          .
        FAB$E
```

5.2.25 RFM - Record Format

The record format (RFM) field indicates the format of the records in the file represented by the File Access Block. When you create a file, your program should ensure that the desired value is present in the RFM field before issuing the $CREATE macro. When you open an existing file, RMS-11 sets this field with the value specified when the file was created.

You can initialize the RFM field at assembly-time with the F$RFM macro. This macro takes the form:

```
F$RFM record-format
```

where

record-format is a symbolic value defining the format of all records contained within the file. One of the following values can be specified:

FB$FIX indicates fixed format records.

FB$STM indicates ASCII stream record format. This value can be specified only for sequential files on disk devices.

FB$UDF indicates no record format is defined for the file. When creating a new file or accessing an existing file with undefined records, your program must use block I/O (refer to Appendix B). When creating such a file, the ORG field must specify sequential file organization.

FB$VAR specifies variable format records (assembly-time default).

FB$VFC	indicates VFC format records. This value can be specified for sequential (on disk devices only) or relative files.

The following is an example of the F$RFM macro:

```
MASTER: FAB$B
          .
          .
          .
        F$RFM FB$FIX
          .
          .
          .
        FAB$E
```

5.2.26 RTV - Retrieval Window Size

The retrieval window size (RTV) field identifies the number of retrieval pointers you want RMS-11 to maintain in memory for the file represented by the File Access Block.

The F$RTV macro allows you to initialize the RTV field at assembly-time. This macro takes the form:

```
F$RTV number
```

where

number	is either a positive integer in the range of 0 to 127 or a -1. A value of 0 (assembly-time default) indicates that RMS-11 is to determine a minimum window size. A non-zero positive value represents the actual number of retrieval pointers to be maintained in memory. A -1 indicates that the entire file is to be mapped through a window maintained in memory.

In the following example, the user requests a retrieval window of 10 retrieval pointers.

```
MASTER: FAB$B
          .
          .
          .
        F$RTV 10
          .
          .
          .
        FAB$E
```

5.2.27 SHR - File Sharing

The file sharing (SHR) field allows you to indicate whether you are willing to let other programs write to the file while your program has the file open at runtime (for either reading or writing). Note that sequentially organized files can be shared for reading only. However, RMS-11 permits relative and indexed files to be shared for both reading and writing.

You can initialize the SHR field at assembly-time by using the F$SHR macro. Its format is:

```
F$SHR value
```

where

value — is a symbolic value or 0. A value of 0 (assembly-time default) indicates that you do not want to share the file at runtime with any program that will write to the file. If, in contrast, you are willing to share the file with one or more programs that will write to the file, you can set this field with the following symbolic value:

FB$WRI - allow shared writers.

In the following example, the user indicates willingness to share the file with programs that write to the file:

```
MASTER: FAB$B
          .
          .
          .
        F$SHR FB$WRI
          .
          .
          .
        FAB$E
```

5.2.28 STS - Completion Status Code

Before returning control to your program, RMS-11 always sets the completion status code (STS) field to indicate success or failure of the file operation. Appendix A contains a complete list of symbolic completion codes that your program can use to test the contents of this field.

5.2.29 STV - Status Value

Based on the type of operation performed and the contents of the STS field, RMS-11 may use the status value (STV) field to communicate additional completion information to your program (refer to Appendix A for a complete listing of the instances in which RMS-11 uses the STV field).

5.2.30 XAB - Extended Attribute Block Pointer

When a particular operation requires the association of Extended Attribute Blocks with a File Access Block, you set the address of the first associated block (of a potential chained list of such blocks) in the XAB field of the FAB. For example, when you create an indexed file, you must always provide at least one XAB -- a key definition XAB for the primary key.

The F$XAB macro allows you to initialize the XAB field at assembly-time:

```
F$XAB xab-address
```

where

xab-address — is the symbolic address of the first Extended Attribute Block associated with this File Access Block. A value of zero (assembly-time default) indicates no XABs are present for a particular operation involving this FAB.

The following is an example of the F$XAB macro:

```
KEYDEF: XAB$B XB$KEY
          .
          .
          .
        XAB$E
          .
          .
          .
MASTER: FAB$B
          .
          .
          .
        F$XAB KEYDEF
          .
          .
          .
        FAB$E
```

In this example, KEYDEF is the label (address) of an Extended Attribute Block.

CHAPTER 6

THE RECORD ACCESS BLOCK

This chapter describes the Record Access Block (RAB), the fields in a RAB, and the assembly-time macros that allocate RABs and initialize fields in RABs.

Record Access Blocks are the second type of control structure that you allocate at assembly-time and use at runtime to communicate with RMS-11. During program execution, you associate (by issuing $CONNECT macro call) a Record Access Block with a File Access Block. The association of a RAB and a FAB is known as a record access stream. Once you have established a record access stream, you use the fields of the Record Access Block to define to RMS-11 the next logical record you want to access in the file.

6.1 ALLOCATING A RECORD ACCESS BLOCK

The RAB$B macro allocates space for a Record Access Block and delimits the beginning of an optional sequence of assembly-time initialization macros for the fields of the block.

The format of the RAB$B macro is as follows:

```
label:RAB$B [type]
```

where

label — is a user-specified symbol that names this particular Record Access Block. You must ensure that the address assigned to this label is word-aligned. Therefore, a .EVEN directive should immediately precede the RAB$B macro.

type — indicates whether or not the Record Access Block can support asynchronous I/O operations (refer to Section 9.3.4 of Chapter 9 for a description of asyncronous record operations). One of the following values can be specified:

- SYN — indicates synchronous record operations only (default).
- ASYN — indicates that the RAB can support both synchronous and asynchronous record operations.

The RAB$E macro delimits the end of an optional sequence of assembly-time initialization macros and stores any specified initial values in the appropriate fields of the block. At assembly-time, all

fields not explicitly initialized are set to their default values. The RAB$E macro must always appear subsequent to an associated RAB$B macro, even when no intervening initialization macros are coded. This macro takes the form:

```
RAB$E
```

The following example shows the allocation for synchronous operations of a Record Access Block named INPUT:

```
        .EVEN
INPUT:  RAB$B           ;ALLOCATE INPUT RAB
        RAB$E           ;END OF INPUT RAB
```

The following example shows the allocation of the same block with the assembly-time initialization of two fields--the File Access Block address and Record Access (RAC) fields of the block.

```
        .EVEN
INPUT:  RAB$B           ;ALLOCATE INPUT RAB
        R$FAB MASTER    ;ADDRESS OF ASSOCIATED FAB
        R$RAC RB$SEQ    ;SEQUENTIAL ACCESS
        RAB$E           ;END OF INPUT RAB
```

6.2 FIELDS IN THE RECORD ACCESS BLOCK

Table 6-1 summarizes the fields in the Record Access Block:

Table 6-1
Record Access Block Fields

Field Name	Field Size	Description
BID	1 byte	Block identifier.
BKT	2 words	Bucket code.
BLN	1 byte	Block length.
CTX	1 word	User context area.
FAB	1 word	File Access Block address.
ISI	1 word	Internal stream identifier.
KBF	1 word	Key buffer address.
KRF	1 byte	Key of reference.
KSZ	1 byte	Key size.
MBC	1 byte	Multi-block count.
MBF	1 byte	Multi-buffer count.
RAC	1 byte	Record access mode.

(Continued on next page)

Table 6-1 (Cont.)
Record Access Block Fields

Field Name	Field Size	Description
RBF	1 word	Record address.
RFA	3 words	Record's file address.
RHB	1 word	Record header buffer.
ROP	1 word (bit string)	Record processing options.
RSZ	1 word	Record size.
STS	1 word	Completion status code.
STV	1 word	Status value.
UBF	1 word	User record area address.
USZ	1 word	User record area size.

The subsections that follow describe the purpose of each of the fields listed in Table 6-1. RMS-11 provides macros that allow you to initialize most of these fields at assembly-time. These macros, when provided, are discussed with the description of the associated field.

6.2.1 BID - Block Identifier

The block identifier (BID) field identifies the block as a RAB. This field, automatically set by the RAB$B macro, contains the value RB$BID. You must never alter this field.

6.2.2 BKT - Bucket Code

The bucket code (BKT) field is used in two instances:

1. When the record access stream represented by the RAB is performing I/O on a relative file.

2. When the record access stream represented by the RAB is performing block I/O.

When the record access stream is performing I/O on a relative file, RMS-11 sets the BKT field to the relative record number of the record accessed by an operation (e.g., $GET, $PUT). This feature is valid only when your program is using sequential access mode.

When performing block I/O, your program uses the BKT field to specify the virtual block number of a block to be read or written (refer to Appendix B for a description of block I/O).

THE R$BKT macro initializes the BKT field at assembly-time:

```
R$BKT vbn
```

where

vbn — is a numeric value representing a virtual block number.

In the following example, the user specifies that the tenth virtual block of the file will be accessed when the program performs its first block I/O operation at runtime.

```
INPUT:  RAB$B
          .
          .
          .
        R$BKT   10
          .
          .
          .
        RAB$E
```

6.2.3 BLN - Block Length

The block length (BLN) field specifies the length of the RAB. This field, automatically set by the RAB$B macro, contains the value RB$BLN for a synchronous RAB and the value RB$BLL for an asynchronous RAB. You must never alter this field.

6.2.4 CTX - User Context Area

The user context area (CTX) is never used in any way by RMS-11. It is intended exclusively for the user. Therefore, you can set any value you choose in this 1-word field. You might, for example, use this field as a means of communicating with a common completion routine in your program.

The R$CTX macro allows you to initialize the CTX field at assembly-time. The format of this macro is as follows:

```
R$CTX argument
```

where

argument — represents any user-selected value.

The following are two examples of the R$CTX macro, showing a numeric value placed in the user context area in the first case, and a symbolic value in the second case.

```
1.  INPUT:  RAB$B
              .
              .
              .
            R$CTX   1
              .
              .
              .
            RAB$E
```

```
2.  INPUT:  RAB$B
              .
              .
              .
            R$CTX   MASTER
              .
              .
              .
            RAB$E
```

6.2.5 FAB - File Access Block Address

The FAB field must contain the address of a File Access Block associated with an open file at the time a $CONNECT macro call (refer to Section 9.3.1) is issued for the current Record Access Block.

The R$FAB macro allows you to initialize the FAB field at assembly-time. Its format is as follows:

```
R$FAB fab-address
```

where

fab-address is the symbolic address of a File Access Block.

The following is an example of a R$FAB macro:

```
MASTER:  FAB$B
           .
           .
           .
         FAB$E
           .
           .
           .
 INPUT:  RAB$B
           .
           .
           .
         R$FAB   MASTER
           .
           .
           .
         RAB$E
```

6.2.6 ISI - Internal Stream Identifier

The internal stream identifier (ISI) field is used by RMS-11 to associate the Record Access Block with a corresponding internal control structure in the space pool. RMS-11 sets this field during a $CONNECT operation. Your program must never alter this field.

6.2.7 KBF - Key Buffer Address

The key buffer address (KBF) field contains the address of a location in your program. You will use this location during certain operations in random access mode to indexed and relative files.

Before issuing a $GET or $FIND operation in random mode to an indexed file, you place in KBF the address of a location containing a character string key value. The size of this character string must be specified in the KSZ field. During execution of the $GET or $FIND operation, RMS-11 uses the character string described by the KBF and KSZ fields to search an index (which you specify through the contents of the KRF field of the RAB) and locate the desired record in the file. The type of match (i.e., exact, generic, approximate, or approximate and generic) that RMS-11 attempts between the character string you specify and key values in records of the file is determined by the KSZ field and the ROP field.

Before issuing a $GET, $FIND, or $PUT operation in random mode to a relative file, you must place in KBF the address of a location containing a relative record number (note that relative record numbers for a relative file begin with 1). The size of this location must be in the KSZ field and must always equal 4.

The R$KBF macro allows you to initialize the KBF field at assembly-time. The format of this macro is:

```
R$KBF address
```

where

address — is the symbolic address of a location containing the key value or relative record number of a record. RMS-11 does not require that this address be word-aligned. The size of this field is specified by the KSZ field. When a relative file is being accessed, the address in KBF points to the least significant bits of the desired relative record number.

The following is an example of the R$KBF macro:

```
INKEY:  BLKB    32.
          .
          .
          .
INPUT:  RAB$B
          .
          .
          .
        R$KBF   INKEY
          .
          .
          .
        RAB$E
```

6.2.8 KRF - Key of Reference

The key of reference (KRF) field is meaningful only when an indexed file is being processed. It is used when your program:

1. Issues $GET or $FIND operations in random access mode to indexed files.

2. Issues $CONNECT or $REWIND operations for indexed files.

During random $GET or $FIND operations, the contents of the KRF field specify which key field is described by the KBF and KSZ fields, e.g., primary key, first alternate key, etc. Thus, KRF tells RMS-11 which index in the file to search seeking a match on the character string key value you described through the contents of the KBF and KSZ fields.

During $CONNECT or $REWIND operations, RMS-11 uses the contents of the KRF field to determine the current context of the record access stream (refer to Section 9.3.3 in Chapter 9 for a discussion of current context). In this case, KRF identifies an index of the file which, in turn, identifies the next record for the stream.

The R$KRF macro can be used to initialize the KRF field at assembly-time. The following is the format of this macro:

```
R$KRF key-number
```

where

key-number — is a numeric value representing a key within the records of the file. A value of zero indicates the primary key. Values of 1 through 254 indicate the desired alternate keys. The assembly-time default value is zero (primary key).

In the following example, the user specifies that the contents of the location called INKEY represent a first alternate key value.

```
INKEY:  BLKB    32.
          .
          .
          .
INPUT:  RAB$B
          .
          .
          .
        R$KBF   INKEY
        R$KRF   1
          .
          .
          .
        RAB$E
```

6.2.9 KSZ - Key Size

The key size (KSZ) field contains the size of the location whose address is in the KBF field of the block.

When you access an indexed file in random mode, you use the contents of the KSZ and ROP fields to specify the type of match that RMS-11 is to perform on the character string key value addressed by the KBF field. If the size in KSZ equals the length of the key field as defined when the file was created, RMS-11 attempts an exact match (if neither RB$KGE or RB$KGT is present in ROP) or an approximate match (if RB$KGE or RB$KGT is present in ROP). If, however, the size in KSZ is less than the defined key length, RMS-11 attempts a generic match (RB$KGE or RB$KGT not in ROP) or a generic-approximate match (either RB$KGE or RB$KGT present in ROP).

When you access a relative file in random mode, you must ensure that KSZ contains the value 4.

The R$KSZ macro allows you to initialize this field at assembly-time. The format of this macro is as follows:

```
R$KSZ size
```

where

size — is a numeric value representing the size (in bytes) of the record key. The specified value must be in the range of 1 to 255.

The following is an example of the R$KSZ macro:

```
INKEY:  BLKB    32.
          .
          .
          .
INPUT:  RAB$B
          .
          .
          .
        R$KBF   INKEY
        R$KSZ   32
        R$KRF   1
          .
          .
          .
        RAB$E
```

In this example, the user specifies that the first alternate key value in location INKEY is 32 bytes in length.

6.2.10 MBC - Multi-block Count

Use of the multi-block count (MBC) field is optional. It applies only when the stream represented by the RAB is accessing a sequentially organized file on disk.

The contents of the MBC field, examined by RMS-11 during execution of the $CONNECT macro, represent the number of virtual blocks you want transferred as a single entity during I/O operations between the file and each of the stream's buffers. Effectively, therefore, the MBC field defines the size of each buffer supporting the stream's access to the file while the MBF field (refer to Section 6.2.11) specifies the number of such buffers RMS-11 is to allocate for the stream.

The primary use of the MBC field is for throughput optimization. It in no way effects the structure of the file being accessed. Rather, it minimizes the number of disk accesses required to support your program's record operation requests. In particular, if your program issues all read requests ($GETs) or all write requests ($PUTs), you can increase execution speed by specifying at $CONNECT time an MBC value that is greater than 1. In combination with this value, you must ensure that the central space pool (refer to Section 3.3.7 in Chapter 3) or your private I/O buffer pool (refer to Section 5.2.7 in Chapter 5) contains sufficient space for the larger buffers.

The R$MBC macro initializes the MBC field at assembly-time. Its format is:

```
R$MBC  blocks
```

where

blocks	is a numeric value from 0 to 255 representing the number of blocks to be transferred during I/O operations and, therefore, the number of virtual blocks contained in each I/O buffer supporting the stream. The assembly-time default is 0. Both 0 and 1 indicate 1 block per buffer.

In the following example, the user specifies that 8 virtual blocks are to be transferred during each I/O operation between the stream's buffers and the file.

```
INPUT:  RAB$B
          .
          .
          .
        R$MBC   8
          .
          .
          .
        RAB$E
```

6.2.11 MBF - Multi-buffer Count

The contents of the multi-buffer count (MBF) field represent the number of I/O buffers you want RMS-11 to allocate when your program issues a $CONNECT operation for this Record Access Block. RMS-11 allocates these buffers either from the centralized space pool or, if the BPA and BPS fields are non-zero in the FAB associated with this RAB, from the private I/O buffer pool associated with the file.

The minimum number of buffers that RMS-11 requires for a record access stream is based on the organization of the file. Table 6-2 lists the minimum number of buffers for each file organization and the maximum number of buffers that you can specify.

Table 6-2
Minimum and Maximum Number of Buffers

File Organization	Minimum MBF	Maximum MBF
Sequential	1	2
Relative	1	No maximum
Indexed	2	No maximum

The R$MBF macro allows you to initialize the MBF field at assembly-time. The format of this macro is:

```
R$MBF buffers
```

where

buffers — is a numeric value representing the number of buffers to be allocated to the record access stream upon execution of a $CONNECT macro call. The specified value must be in the range of 0 to 255. A value of 0 (assembly-time default) indicates the minumum number of buffers based on the file's organization. If the user specifies less than the minimum required by the organization of the file, RMS-11 ignores the user specification and allocates the minimum required. If the user, for a sequential file, specifies more than the maximum, RMS-11 allocates the maximum number (2). If the user specifies more buffers than are available, RMS-11 allocates as many as possible.

In the following example, the user specifies the allocation of four buffers:

```
INPUT:  RAB$B
          .
          .
          .
        R$MBF   4
          .
          .
          .
        RAB$E
```

6.2.12 RAC - Record Access Mode

The RAC field specifies the access mode to be used to retrieve or store a record.

The R$RAC macro allows you to initialize the RAC field at assembly-time. Its format is:

```
R$RAC access-mode
```

where

access-mode — is a symbolic value representing the type of access desired. One of the following may be specified:

- RB$KEY — indicates random access. You can specify this value only with relative or indexed files.
- RB$RFA — indicates access by record's file address (for files on disk devices only). When you specify this value, RMS-11 uses the record's file address (RFA) field of this RAB to access the record.
- RB$SEQ — indicates sequential access. You can specify this value with any file organization.

The following is an example of the R$RAC macro in which the user indicates random access:

```
INPUT:  RAB$B
          .
          .
          .
        R$RAC   RB$KEY
          .
          .
          .
        RAB$E
```

6.2.13 RBF - Record Address

The record address (RBF) field is used to pass the address of a record between RMS-11 and your program. As output from a successful $GET operation, RMS-11 always places the memory address of the retrieved record in RBF. Conversely, your program, prior to a $PUT or $UPDATE operation, must place in RBF the memory address of the record to be written out to the file. In both instances, the RSZ field describes the size of the record. Refer to Section 9.3.5.2.1 in Chapter 9 for further details on the RBF and RSZ fields.

You can initialize the RBF field at assembly-time with the R$RBF macro. The format of this macro is as follows:

```
R$RBF address
```

where

address — is the symbolic address of an area in your program.

In the following example, the user initializes the RBF field with the address of a program location that, at runtime, will contain a record to be written to the file.

```
RECBUF:  .BLKB   150.
           .
           .
           .
 INPUT:  RAB$B
           .
           .
           .
         R$RBF   RECBUF
           .
           .
           .
         RAB$E
```

6.2.14 RFA - Record's File Address

The record's file address (RFA) field is used to pass the RFAs of records between RMS-11 and your program. After successful $GET, $PUT, and $FIND operations, RMS-11 returns to your program the RFA of the record successfully operated upon. Your program can then save the contents of this field for subsequent retrieval of the record using RFA access mode. Before retrieving a record using RFA access, your program must load the RFA of the desired record in the RFA field.

6.2.15 RHB - Record Header Buffer

RMS-11 uses the record header buffer (RHB) field only when the record format field (RFM) of the associated File Access Block indicates VFC format records. When RHB is non-zero, RMS-11 interprets its contents as the address of a buffer for the fixed control area portion of VFC records. On $GET operations, RMS-11 strips the fixed control area portion of VFC records and places it in the buffer whose address is in the RHB field. On $PUT and $UPDATE operations, RMS-11 prefixes the data portion of the record with the fixed control area portion described by the RHB field before the entire record is written to the file. For all such operations, the size of the fixed control area portion is a defined attribute of the file. During a $OPEN macro call, RMS-11 returns this size in the FSZ field of the associated File Access Block. When your program processes the fixed control area portion of VFC records, you must ensure that the buffer described by the RHB field is at least as large as the value in FSZ.

When you do not want to process the fixed control area portion of VFC records, you zero the RHB field. When RHB is zero, RMS-11 does the following:

1. Skips the fixed control area portion of the record on $GET operations.
2. Writes out a zero-filled fixed control area on $PUT operations.
3. Leaves the original fixed control area unaltered on $UPDATE operations.

You can initialize the RHB field at assembly-time with the R$RHB macro:

```
R$RHB header-address
```

where

header-address is the symbolic address of an area within your program. An address of zero in the RHB field indicates the nonexistence of the buffer.

6.2.16 ROP - Record Processing Options

The record processing options (ROP) field allows you to request optional functions during the execution of a record operation.

The R$ROP macro allows you to initialize the ROP field at assembly-time. The format of this macro is as follows:

```
R$ROP option[!option...]
```

where

option is a symbolic value representing record processing options. One or more of the following options may be specified in any order.

RB$ASY indicates asynchronous operation. When this value is set, RMS-11 can return control to your program before the specified record operation is completed (assuming that the RAB has been defined as asynchronous in the RAB$B macro).

RB$EOF indicates that RMS-11 is to position to end of file when the $CONNECT macro call is issued for the record access stream. This option applies only to sequential files on disk devices.

RB$FDL indicates fast delete (indexed files only). When this value is present, $DELETE operations will mark records in the file as deleted but RMS-11 will not remove pointers to the record from the indexes.

RB$KGE indicates that RMS-11 is to search for the first record containing a value in the key specified by the KRF field that is greater than or equal to the value described by the KBF and KSZ fields. This option applies only to indexed files and is one form of the approximate match facility (see also RB$KGT).

RB$KGT specifies the second form of the approximate match facility (see also RB$KGE). It indicates that RMS-11 is to search for the first record containing a value in the key specified by the KRF field that is greater than the value described by the KBF and KSZ fields. This option applies only to indexed files.

RB$LOA indicates that RMS-11 is to follow bucket fill sizes for record insertions to indexed files. The fill sizes for a file are specified by the user at file creation time in the DFL and IFL fields of key definition XABs. The assembly-time default is to ignore bucket fill sizes (i.e., buckets will be completely filled).

RB$LOC indicates locate mode. Locate mode on $GET operations is supported for all file organizations. Locate mode on $PUT operations is supported only for sequential files. Locate mode is not permitted if the file was opened for $UPDATE operations (i.e., the value FB$UPD was present in the FAC field of the FAB during the $OPEN or $CREATE operation).

RB$UIF indicates that if a $PUT operation to a relative file encounters an existing record in the target record cell, the existing record is to be updated. If RB$UIF is not set, RMS-11 returns an ER$REX (record already exists) error if a $PUT operation to a relative file encounters an existing record.

The following are two examples of the R$ROP macro. In the first example, the user specifies positioning to end of file and locate mode operations. In the second example, the user specifies an approximate key search, i.e., RMS-11 is to search for a record containing a key value equal to or greater than the key described by the KBF, KSZ, and KRF fields.

```
1.  INPUT:  RAB$B
              .
              .
              .
            R$ROP   RB$EOF!RB$LOC
              .
              .
              .
            RAB$E

2.  INPUT:  RAB$B
              .
              .
              .
            R$ROP   RB$KGE
              .
              .
              .
            RAB$E
```

6.2.17 RSZ - Record Size

The record size (RSZ) field contains the size (in bytes) of the record whose address is in the RBF field. Following a successful $GET operation, RMS-11 places the size of the retrieved record in the RSZ field. Conversely, your program, before issuing a $PUT or $UPDATE operation, must ensure that the RSZ field contains the size of the record to be written.

The R$RSZ macro can be used to initialize the RSZ field at assembly-time. The format of this macro is:

```
R$RSZ record-size
```

where

record-size is a numeric value representing the size (in bytes) of the record whose address is in the RBF field. The value specified must range from 1 to 16383.

The following is an example of the R$RSZ macro:

```
RECBUF:  .BLKB   150.
           .
           .
           .
 INPUT:  RAB$B
           .
           .
           .
         R$RBF   RECBUF
         R$RSZ   150
           .
           .
           .
         RAB$E
```

In this example, the user specifies a record size of 150 bytes.

6.2.18 STS - Completion Status Code

RMS-11 always sets the completion status code (STS) field to indicate success or failure of the record operation. Appendix A contains a complete list of symbolic completion codes that your program can use to test the contents of this field.

6.2.19 STV - Status Value

Based on the type of operation performed and the contents of the STS field, RMS-11 may use the status value (STV) field to communicate additional completion information to your program (refer to Appendix A for a complete list of the instances in which RMS-11 uses the STV field).

6.2.20 UBF - User Record Area Address

The user record area address (UBF) field must contain a valid address regardless of the data transfer mode (i.e., move or locate mode) associated with the record access stream. Refer to Section 9.3.5.2.2 in Chapter 9 for details on the function of the UBF field.

The R$UBF macro allows you to initialize the UBF field at assembly-time. The format of this macro is:

```
R$UBF address
```

where

address — is the symbolic address of a work area within your program. You must specify the size of this work area in USZ.

The following is an example of the R$UBF macro.

```
WAREA:  .BLKW   1024.
          .
          .
          .
INPUT:  RAB$B
          .
          .
          .
        R$UBF   WAREA
          .
          .
          .
        RAB$E
```

6.2.21 USZ - User Record Area Size

The USZ field contains the size of the user record area whose address is in the UBF field.

You can initialize the USZ field at assembly-time with the R$USZ macro:

```
R$USZ area-size
```

where

area-size — is a numeric value representing the size (in bytes) of the user record area whose address is in the UBF field. The value specified must range from 1 to 16383.

In the following example, the user specifies a user record area 2048 bytes in size.

```
WAREA:  .BLKW   1024.
          .
          .
          .
INPUT:  RAB$B
          .
          .
          .
        R$UBF   WAREA
        R$USZ   2048
          .
          .
          .
        RAB$E
```

CHAPTER 7

EXTENDED ATTRIBUTE BLOCKS

This chapter describes Extended Attribute Blocks, the fields in Extended Attribute Blocks, and the assembly-time macros that allocate XABs and initialize fields in XABs.

Extended Attribute Blocks contain file and record attribute information beyond that specified in the associated File Access Block. XABs are generally required only when a file is being created (particularly an indexed file) or when the $DISPLAY macro call is used to retrieve file attributes.

When more than one Extended Attribute Block is required, they are linked together. The XAB field of the File Access Block points to the first block in the linked list.

There are five types of Extended Attribute Blocks:

1. Date and time XABs
2. Key definition XABs
3. File protection specification XABs
4. Allocation XABs
5. Summary XAB

After descriptions of XAB allocation and linking, the subsections that follow describe each type of XAB.

7.1 ALLOCATING AN EXTENDED ATTRIBUTE BLOCK

The XAB$B macro allocates space for an Extended Attribute Block and delimits the beginning of an optional sequence of assembly-time initialization macros for the fields of the block.

The format of the XAB$B macro is as follows:

```
label:XAB$B  type
```

where

label — is a user-specified symbol that names this particular Extended Attribute Block. You must ensure that the address assigned to this label is word-aligned. Therefore, a .EVEN directive should immediately precede the XAB$B macro.

type — is a symbolic value representing the type of attribute information contained in the block. At assembly-time, the specified value is stored in the COD field of the block. At runtime, the value in the COD field determines how RMS-11 interprets the remainder of the Extended Attribute Block.

One of the following values must be specified at assembly-time.

XB$DAT — indicates a date and time XAB.

XB$KEY — indicates a key definition XAB.

XB$PRO — indicates a file protection specification XAB.

XB$ALL — indicates an allocation XAB.

XB$SUM — indicates a summary XAB

The XAB$E macro delimits the end of an optional sequence of assembly-time initialization macros and stores any specified initial values in the appropriate fields of the block. At assembly-time all fields not explicitly initialized are set to their default values. The XAB$E macro must always appear subsequent to an associated XAB$B macro, even when no intervening initialization macros are coded. This macro takes the form:

```
XAB$E
```

The following example shows the allocation of a key definition Extended Attribute Block named KEYDEF:

```
        .EVEN
KEYDEF: XAB$B  XB$KEY
        XAB$E
```

7.2 LINKING AND ORDERING EXTENDED ATTRIBUTE BLOCKS

Regardless of the type of information contained (e.q., key definition, protection specification, etc.), every XAB has an NXT field. The NXT field links one XAB with another. When multiple XABs are needed for a particular operation (e.g., $CREATE, $DISPLAY), the address of the first XAB is placed in the XAB field of the File Access Block. The address of the second XAB is placed in the NXT field of the first XAB, and so forth. The NXT field of the last XAB in the linked chain is set to zero to indicate the end of the chain.

Within a chain of XABs, there is no mandatory ordering of blocks based on XAB type (i.e., contents of the COD field). Assume, for example, that you want to obtain the attributes of a single-key indexed file. At assembly-time, you could allocate a date and time XAB, a key definition XAB, a file protection XAB, and a summary XAB. Then, either at assembly-time or at runtime, you could link these XABs in any order by appropriately setting the contents of the NXT field in each block. Finally, you would issue a $DISPLAY macro call and RMS-11 would fill attribute information into each block -- the type of information going into each block being determined by the COD field of the block.

While different types of XABs can appear in any order in a chain, multiple instances of the same type must appear in a particular order. This rule applies to key definition and allocation XABs, since these are the only ones that RMS-11 permits multiple instances of in a chain. Key definition XABs must be linked together in ascending order based on the contents of the key of reference (REF) field (refer to Section 7.4.10). Allocation XABs must be linked together in ascending order based on the contents of the area identification number (AID) field (refer to Section 7.6.1). In both instances, there cannot be any intervening XABs of another type in the sub-chain of XABs of the same type. Further, the operation for which these XABs are present determines whether or not the ascending order must be dense. For $CREATE operations, key definition and allocation XABs, if present, must appear in densely ascending order by, respectively, key of reference (REF) and area identification number (AID). For $OPEN, $EXTEND, and $DISPLAY operations, key definition and allocation XABs, if present, must be in ascending order but need not be dense.

You use the X$NXT macro to initialize the NXT field of an XAB at assembly-time. The format of the X$NXT macro is as follows:

```
X$NXT xab-address
```

where

xab-address — is the symbolic address of the next XAB within the current chain of XABs. A value of zero indicates the last (or only) XAB in a chain.

7.3 DATE AND TIME EXTENDED ATTRIBUTE BLOCKS

Date and time Extended Attribute Blocks contain fields specifying the date and time the associated file was created and the date and time the file was last updated. There are no initialization macros for the fields in this XAB since you can only examine, but not alter, the date and time attributes of a file. The allocation of this XAB at assembly-time causes the fields of the block to be initialized to zero.

Table 7-1 describes the fields of a date and time XAB.

Table 7-1
Date and Time XAB Fields

Field Name	Field Size	Description
CDT	4 words	The date and time the file was created, expressed as an 8 byte binary number.
COD	1 byte	Code field. Contains the value XB$DAT.
NXT	1 word	Next XAB.
RDT	4 words	The date and time the file was last updated, expressed as an 8 byte binary number.
RVN	1 word	Revision number. Contains the number of times the file was opened (via a $OPEN macro call) for write operations.

The 8-byte binary numbers (in which the lowest-addressed byte is the least significant) in the CDT and RDT fields represent the number of hundreds of nanoseconds elapsed since the beginning of November 17, 1858.

7.4 KEY DEFINITION EXTENDED ATTRIBUTE BLOCKS

Each key definition Extended Attribute Block describes one key of an indexed file. When you create an indexed file, you must set the contents of the fields of this XAB before you issue the $CREATE macro call. You must, further, provide one key definition XAB for each key you want the file to have. Since every indexed file has at least one key (the primary key) you will always require at least one key definition XAB.

When you open an existing indexed file or issue a $DISPLAY operation for such a file, you use key definition XABs only if you want RMS-11, as output from the macro call, to provide your program with one or more of the key definitions specified when the file was created.

Table 7-2 summarizes the fields of a key definition XAB.

Table 7-2
Key Definition XAB Fields

Field Name	Field Size	Description
COD	1 byte	Code field. Contains the value XB$KEY.
DAN	1 byte	Area number for data buckets.
DFL	1 word	Data bucket fill size.
FLG	1 byte (bit string)	Key options.
IAN	1 byte	Area number for index buckets.
IFL	1 word	Index bucket fill size.
KNM	1 word	Key name address.
LAN	1 byte	Area number for lowest level of index.
NUL	1 byte	Null key value.
NXT	1 word	Next XAB.
POS	8 words	Key position.
REF	1 byte	Key of reference.
RVB	2 words	Root virtual block number.
SIZ	8 bytes	Key size.

The following subsections describe the fields listed in Table 7-2 that are unique to the key definition XAB.

7.4.1 DAN - Area Number for Data Buckets

You set a value in the DAN field only if both of the following are true:

1. You are creating a new indexed file.
2. You are using allocation XABs (described in Section 7.6) to control the placement and structure of the file.

If both of the above are true, then you use the DAN field to assign the buckets of the data level of the index defined by this XAB to one of the areas defined by an allocation XAB.

When the key definition XAB describes the primary key, the data level of the index consists of those buckets containing the actual data records of the file. When, however, the XAB describes an alternate key, the data level of the index consists of buckets in which RMS-11 maintains pointer arrays describing the data records.

The X$DAN macro allows you to initialize the DAN field at assembly-time. Its format is:

```
X$DAN    aid
```

where

aid — is a numeric value from 0 to 254 representing an area identification number contained in the AID field of an allocation XAB present in the same chain. The assembly-time default is 0 (i.e., area 0).

In the following example, the user assigns the data level of the key defined by the XAB called KEYDEF to area 1.

```
AREA1:  XAB$B  XB$ALL
          .
          .
          .
        X$AID  1
          .
          .
          .
        XAB$E

          .
          .
          .
KEYDEF: XAB$B  XB$KEY
          .
          .
          .
        X$DAN  1
          .
          .
          .
        XAB$E
```

7.4.2 DFL - Data Bucket Fill Size

At $CREATE time, you can use the DFL field to specify how many bytes you want used in each data level bucket of the associated index. By specifying less than the total bucket size, you indicate that the data buckets are not to be completely filled but are to contain some amount of free space. At runtime, RMS-11 adheres to the fill size specified at $CREATE time only if the RB$LOA value is present in the record processing options (ROP) field of the RAB.

When the key definition XAB describes the primary key, the DFL field describes space in buckets containing the actual user data records. When the XAB describes an alternate key, the DFL field describes space in buckets of the alternate index containing pointers to the user data records.

You may want to use the facility provided by the DFL field (and the similar facility provided by the IFL field, described in Section 7.4.5) in the following situation: If you expect to perform numerous $PUT and $UPDATE operations on the file after it has been initially populated, you can minimize the resultant movement of records (known as bucket splitting) by specifying a less than maximum bucket fill size at $CREATE time. To utilize the free space thereby reserved in the buckets, programs that perform $PUT or $UPDATE operations on the file should not place, at $CONNECT time, the value RB$LOA in the ROP field of the RAB.

The X$DFL macro allows you to initialize the DFL field at assembly-time. The format of this macro is:

```
X$DFL space
```

where

space — is a numeric value representing the maximum number of bytes to be used in the data buckets. The assembly-time default value is 0, which is interpreted at runtime by RMS-11 as meaning the bucket size in bytes (i.e., no unused space).

The following is an example of the X$DFL macro:

```
KEYDEF:  XAB$B  XB$KEY
           .
           .
           .
         X$DFL  256
           .
           .
           .
         XAB$E
```

In this example, the user specifies that each bucket in the data level is to be filled to a maximum of 256 bytes.

7.4.3 FLG - Key Options

The key options (FLG) field defines optional characteristics of the key represented by the XAB. These characteristics are:

1. Duplicate key values are allowed.
2. Key values can change.
3. Null key value.

You use the X$FLG macro to initialize the FLG field at assembly-time. Its format is:

```
X$FLG option[!option...]
```

where

option — is a symbolic value specifying a characteristic of the key field defined by the Extended Attribute Block. The following values may be specified:

XB$CHG — indicates that the associated key within any record can be changed by a program during a $UPDATE operation. This option can be specified only for alternate keys and only in conjunction with XB$DUP.

XB$DUP — indicates that records within the file may have the same values within the key field associated with this XAB.

The allowed combinations of XB$CHG and XB$DUP depend on the type of key (i.e., primary or alternate) represented by the extended attribute block. Table 7-3 summarizes these combinations.

XB$NUL — indicates that the NUL field of this XAB contains a null key value. You can specify this characteristic only for alternate keys.

Table 7-3
Key Option Combinations

Key Type	Combination			
	CHANGE + DUPLICATES	CHANGE + NO DUPLICATES	NO CHANGE + DUPLICATES	NO CHANGE + NO DUPLICATES
Primary	Error	Error	Allowed	Default
Alternate	Default	Error	Allowed	Allowed

The assembly-time default values for the FLG field depend on the type of key (i.e., primary or alternate) defined by the XAB. The defaults for a primary key are:

1. Duplicate key values are not allowed.

2. Key values cannot change.

3. No null key value.

The defaults for alternate keys are:

1. Duplicate key values allowed.

2. Key values can change.

3. No null key value.

The symbolic values you specify using the X$FLG assembly-time macro or the $STORE runtime macro are mapped into bit settings in the FLG field. When a bit is set, it means that the corresponding characteristic was specified (e.g., key values can change or duplicate key values allowed or null key value defined). Conversely, when a bit is zero, it means that the corresponding characteristic is not defined for the key (e.g., key values cannot change, or duplicate keys not allowed or null key not defined). Both the X$FLG and the $STORE macro affect the entire FLG field, setting those bits specified and clearing to zero those not specified. Therefore, when defining the characteristics of a key, you should specify exactly what you want the field to contain, expressing bit settings as symbolic values. Any symbolic values you omit will result in a zero in the corresponding bit position. As examples, consider the following:

1.
```
KEYDEF:  XAB$B  XB$KEY
           .
           .
           .
         X$FLG  XB$DUP!XB$NUL
           .
           .
           .
         XAB$E
```

2.
```
KEYDEF:  XAB$B  XB$KEY
           .
           .
           .
         X$FLG  XB$NUL
           .
           .
           .
         XAB$E
```

3.
```
STORE  #0,FLG,R3
```

In the first example, the user specifies that duplicate keys are allowed, a null key value is defined, and (through the absence of XB$CHG) that key values cannot change. In the second example, the user specifies that a null key value is defined but duplicate key values are not permitted and key values cannot change. In the third example, the user sets the FLG field to indicate that duplicate key values are not permitted, key values cannot change, and no null key is defined.

7.4.4 IAN - Area Number for Index Buckets

You set a value in the area number for index buckets (IAN) field only if both of the following are true:

1. You are creating a new indexed file.

2. You are using allocation XABs (refer to Section 7.6) to control the placement and structure of the file.

If both of the above are true, then you use the IAN field to assign the buckets of the index level of the index defined by this XAB to one of the areas defined by an allocation XAB.

When the key definition XAB describes the primary key, the index level of the index consists of all levels of the tree-structured primary index down to and including the level containing pointers to the user data records themselves. When, however, the XAB describes an alternate key, the index level consists of all levels of the tree-structured alternate index down to, but not including, the level containing buckets in which RMS-11 maintains pointer arrays describing the user data records (refer to the LAN field description for an additional facility in index bucket assignment).

The X$IAN macro allows you to initialize the IAN field at assembly-time. Its format is:

```
X$IAN    aid
```

where

aid — is a numeric value from 0 to 254 representing an area identification number contained in the AID field of an allocation XAB present in the same chain. The assembly-time default is 0 (i.e., area 0).

In the following example, the user assigns the index level of the key defined by the XAB called KEYDEF to area 2.

```
AREA2:  XAB$B  XB$ALL
          .
          .
          .
        X$AID  2
          .
          .
          .
        XAB$E

          .
          .
          .

KEYDEF: XAB$B  XB$KEY
          .
          .
          .
        X$IAN  2
          .
          .
          .
        XAB$E
```

7.4.5 IFL - Index Bucket Fill Size

At $CREATE time, you can use the IFL field to specify how many bytes you want used in each index level bucket of the associated index. By specifying less than the total bucket size, you indicate that the index buckets are not to be completely filled but are to contain some amount of free space. At runtime, RMS-11 adheres to the fill size specified at $CREATE time only if the RB$LOA value is present in the record processing options (ROP) field of the RAB.

When the key definition XAB describes the primary key, the IFL field describes space in buckets in all levels of the tree-structured primary index down to and including the level containing pointers to the user data records themselves. When the XAB describes an alternate key, IFL applies to all levels of the tree-structured alternate index down to, but not including, the level containing buckets in which RMS-11 maintains pointer arrays describing the user data records.

You may want to use the facility provided by the IFL field (and the similar facility provided by the DFL field, described in Section 7.4.2) in the following situation: If you expect to perform numerous $PUT and $UPDATE operations on the file after it has been initially populated, you can minimize the resultant movement of index entries (known as bucket splitting) by specifying a less than maximum bucket fill size at $CREATE time. To use the free space thereby reserved in the index buckets, programs that perform $PUT or $UPDATE operations on the file should not place, at $CONNECT time, the value RB$LOA in the ROP field of the RAB.

You can initialize the IFL field at assembly-time with the X$IFL macro. The format of the X$IFL macro is as follows:

```
X$IFL space
```

where

space — is a numeric value representing the maximum number of bytes to be used in the index buckets. The assembly-time default is 0, which is interpreted at runtime by RMS-11 as meaning the bucket size in bytes (i.e., no unused space).

The following is an example of the X$IFL macro:

```
KEYDEF:  XAB$B  XB$KEY
           .
           .
           .
         X$IFL  300
           .
           .
           .
         XAB$E
```

In this example, the user specifies that 300 bytes of each index bucket are to be used.

7.4.6 KNM - Key Name Address

The Key Name Address (KNM) field can contain the address of a 32-character ASCII string. When you are defining a key, you can associate any 32-character string you choose with the key field represented by the XAB. RMS-11 never examines this character string but retains it in the file as part of the key definition information.

You can initialize the KNM field at assembly-time with the X$KNM macro. Its format is:

```
X$KNM address
```

where

address — is the symbolic address of a buffer. This buffer should always be at least 32 bytes in length. A value of 0 in this field indicates that no key name is defined ($CREATE) or is to be displayed ($OPEN or $DISPLAY).

7.4.7 LAN - Lowest Index Level Area Number

You set a value in the LAN field only when both of the following are true:

1. You are creating a new indexed file.
2. You are using allocation XABs (refer to Section 7.6) to control the placement and structure of the file.

If both of the above are true, then the LAN field allows you to separate the lowest level of the index from all higher levels by specifying a different area number than that specified by the IAN field. However, the bucket size of the area specified by the LAN field must be the same as that of the area identified by the IAN field.

You can initialize the LAN field at assembly-time with the X$LAN macro:

```
X$LAN    aid
```

where

aid — is a numeric value from 0 to 254. A value of 0 indicates that the lowest level of the index is to occupy the same area as the remainder of the index. Values from 1 to 254 represent an area identification number contained in the AID field of an allocation XAB. The assembly-time default is 0.

7.4.8 NUL - Null Key Value

The NUL field can contain any user-selected character value. When you create an indexed file, however, RMS-11 examines and saves the contents of this field only if the value XB$NUL is present in the FLG field of the block. Further, the null key facility is available only for Extended Attribute Blocks that define an alternate key.

When writing a record into an indexed file, RMS-11 normally updates all indexes of the file to reflect the values found in the corresponding key fields of the record. However, if a null key value is defined for a particular alternate key, RMS-11 examines the contents of the key field in the record. If this field consists solely of the null key characters specified in the NUL field of the XAB that defined the key at $CREATE time, RMS-11 will not make an entry in the associated alternate index for the record.

The X$NUL macro can be used to initialize the NUL field at assembly-time. The format of this macro is as follows:

```
X$NUL value
```

where

value is any user-selected character value.

The following are two examples of the X$NUL macro:

```
1.  KEYDEF:  XAB$B   XB$KEY
               .
               .
               .
             X$FLG   XB$NUL
             X$NUL  <'Z>
               .
               .
               .
             XAB$E

2.  KEYDEF:  XAB$B   XB$KEY
               .
               .
               .
             X$FLG   XB$NUL
             X$NUL  <^O177>
               .
               .
               .
             XAB$E
```

In the first example, the user specifies that any record containing all Z's in the key field defined by the current XAB is not to have an entry made for it in the associated alternate index. In the second example, the user specifies the same action but the null key value is the ASCII character DEL.

7.4.9 POS - Key Position

In combination with the SIZ field, the POS field defines the location of the key within each record of the file. The POS field is eight words in length because two types of keys can be defined -- simple keys and segmented keys.

A simple key is a single string of contiguous bytes in the record. The first word of the POS field specifies the starting position of the string and the remaining words contain zeros.

A segmented key consists of two to eight strings of bytes in the record. Each individual string (segment) is a set of contiguous bytes, but the strings need not be contiguous with each other, nor

need they be in any particular order. Each successive word of the POS field specifies the starting position of one of the segments. When processing records that contain segmented keys, RMS-11 treats the individual segments of the key as a single, logically contiguous string beginning with the first segment and ending with the last.

The X$POS macro can be used to initialize the POS field at assembly-time. The format of this macro for simple keys is as follows:

```
X$POS position
```

where

position — is a numeric value representing the starting position of the key within each record. The first byte of a record is represented by a value of 0, the second by a value of 1, etc.

For segmented keys, the format of the X$POS macro is as follows:

```
X$POS  <pos0,pos1[,pos2...,pos7]>
```

where

pos0,pos1,etc. — are numeric values representing starting positions of each segment of the segmented key. Up to eight values can be specified. It is not necessary that the list of values represent ascending byte positions in the record. To define a segmented key, the POS field and the SIZ field must contain an equal number of successive values.

Three examples of the X$POS macro follow:

```
1.  KEYDEF:  XAB$B  XB$KEY
               .
               .
               .
             X$POS  0
               .
               .
               .
             XAB$E

2.  KEYDEF:  XAB$B  XB$KEY
               .
               .
               .
             X$POS  8
               .
               .
               .
             XAB$E

3.  KEYDEF:  XAB$B  XB$KEY
               .
               .
               .
             X$POS  <19,0,13,28>
               .
               .
               .
             XAB$E
```

In the first example, the user defines a simple key that begins in the first byte of each record. The second example shows the specification of a simple key beginning in the ninth byte of each record. To complete the definition of these simple keys, the user must establish a corresponding value in the key size (SIZ) field of the block. In the third example, the user specifies a segmented key. The first segment of the key begins in the twentieth byte, the second segment begins in the first byte, the third segment in the fourteenth byte, and the fourth segment in the twenty-ninth byte. To complete the definition of this segmented key, the user must establish four values for the size field (SIZ) corresponding to these four segment positions.

7.4.10 REF - Key of Reference

The key of reference (REF) field identifies which key (i.e., primary, first alternate, second alternate, etc.) is defined by the XAB.

The X$REF macro allows you to initialize the REF field at assembly-time. The format of this macro is as follows:

```
X$REF value
```

where

value — is a numeric value, in the range of 0 to 254, indicating which key is represented by the current block. A value of 0 indicates the primary key, 1 indicates the first alternate key, etc. The assembly-time default value for the REF field is zero (i.e., primary key).

The following are two examples of the X$REF macro, showing the specification of a primary key in the first case, and the specification of a third alternate key in the second case.

```
1.  KEYDEF:  XAB$B  XB$KEY
               .
               .
               .
             X$REF  0
               .
               .
               .
             XAB$E

2.  KEYDEF:  XAB$B  XB$KEY
               .
               .
               .
             X$REF  3
               .
               .
               .
             XAB$E
```

7.4.11 RVB - Root Virtual Block Number

When a key definition XAB is present during an $OPEN or $DISPLAY operation, RMS-11 sets RVB to the number of the virtual block of the file that contains the root of the index for the key described by the XAB.

7.4.12 SIZ - Key Size

The key size (SIZ) field specifies the length of the key whose starting position is contained in the POS field of the same block.

The X$SIZ macro can be used to initialize the SIZ field at assembly-time.

For simple keys, the format of the X$SIZ macro is as follows:

```
X$SIZ size
```

where

size — is a numeric value representing the length (in bytes) of the key defined by the block. Allowed values for the SIZ field are 1 to 255.

For segmented keys, the format of the X$SIZ macro is as follows:

```
X$SIZ <size0,sizel [,size2...size7]>
```

where

size0,sizel,etc. — are numeric values representing the length of each segment of the segmented key. Up to eight values can be specified. The total size specified must be less than 256. To define a segmented key, you must provide an equal number of values for the SIZ and POS fields of the block. For example, the first value specified on an X$POS macro will be combined with the first value on an X$SIZ macro to define the location and length of the first segment of the segmented key, etc.

Two examples of the X$SIZ macro follow:

1.
```
KEYDEF:  XAB$B   XB$KEY
           .
           .
           .
         X$SIZ  8
           .
           .
           .
         XAB$E
```

2.
```
KEYDEF:  XAB$B   XB$KEY
           .
           .
           .
         X$SIZ  <8,2,5,32>
           .
           .
           .
         XAB$E
```

In the first example, the user specifies a key length of 8 bytes. In the second example, the user specifies the lengths of the four segments of a segmented key. Assume that the following initialization macro is also present for the same block:

```
X$POS <19,0,13,28>
```

The following is the full definition of the segmented key:

Start Position		Length
20th byte		8 bytes
1st byte		2 bytes
14th byte		5 bytes
29th byte		32 bytes
Total Key Length	=	47 bytes

7.5 FILE PROTECTION SPECIFICATION EXTENDED ATTRIBUTE BLOCK

A file protection specification Extended Attribute Block contains protection information for the associated file. When you create a file, you can use this type of XAB to explicitly assign protection codes. When you open an existing file, you can use this type of XAB to obtain the protection codes specified when the file was created. Table 7-4 describes the fields of this type of XAB.

Table 7-4
File Protection Specification XAB Fields

Field Name	Field Size	Description
COD	1 byte	Code field contains the value XB$PRO.
NXT	1 word	Next XAB.
PRG	1 word	Programmer number portion of owner's user identification code (UIC).
PRJ	1 word	Project number portion of owner's user identification code (UIC).
PRO	1 word	System dependent file protection value.

The following subsections describe the fields listed in Table 7-4 that are unique to the file protection specification XAB.

7.5.1 PRG - Programmer Number

The PRG field contains the programmer number portion of the file owner's user identification code.

The X$PRG macro can be used to initialize the PRG field at assembly-time. The format of the X$PRG macro is as follows:

```
X$PRG number
```

where

number	is a numeric value representing a programmer number. The specified value must range from 1 to 255.

The following is an example of the X$PRG macro:

```
PRODEF:  XAB$B  XB$PRO
           .
           .
           .
         X$PRG  11
           .
           .
           .
         XAB$E
```

7.5.2 PRJ - Project Number

The PRJ field contains the project number portion of the file owner's user identification code.

The X$PRJ macro allows you to initialize the PRJ field at assembly-time. The X$PRJ macro takes the following form:

```
X$PRJ number
```

where

number	is a numeric value representing a project number. The specified value must range from 1 to 255.

The following is an example of the X$PRJ macro:

```
PRODEF:  XAB$B  XB$PRO
           .
           .
           .
         X$PRG  211
           .
           .
           .
         XAB$E
```

7.5.3 PRO - System File Protection Value

The system file protection value (PRO) field identifies the file access privileges of four classes of users:

1. Group - the class of users with the same project number as contained in the PRJ field of the current block.
2. Owner - the owner of the file.
3. System - users executing under a privileged user identification code.
4. World - all users not within the group, owner, and system categories.

The format of the one-word PRO field is as follows:

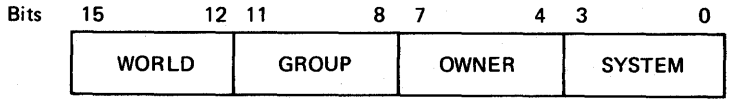

Figure 7-1 Format of PRO Field

Each of the four categories above has four bits; each bit has the following meaning with respect to file access:

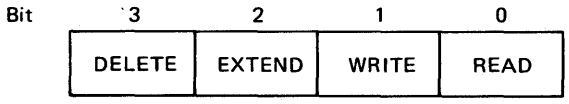

Figure 7-2 File Access Bits

A bit value of zero (0) indicates that the respective type of access to the file is to be allowed; a bit value of one (1) indicates that the respective type of access to the file is to be denied.

The X$PRO macro initializes the PRO field at assembly-time. Its format is:

```
X$PRO          protect
```

where

protect — is a numeric value representing the desired encoding for the 1-word field. The default radix for this value is decimal. Use of the MACRO-11 octal radix unary operator will facilitate the setting of this field.

In the following example, the user grants full access rights to the system and owner and restricts the group and world classes to read only privileges.

```
X$PRO <^O167000>
```

7.6 ALLOCATION EXTENDED ATTRIBUTE BLOCKS

Each allocation XAB describes one area of a file. An area is a portion of a file that is treated as a single entity for the purposes of:

- initial allocation
- extension
- placement control
- bucket size

Only indexed files can contain more than one area. Multiple areas, if present, in an indexed file are consecutively numbered beginning with 0. Sequential and relative files are always composed of a single area (area 0).

In allowing you to define file areas, allocation XABs serve the following two purposes:

1. They allow you to control the physical placement of a file (of any organization) on a volume.

2. They allow you to control the internal structure of an indexed file.

To control the physical placement of a file on a volume, you provide one (for any type of file) or more (for indexed files only) allocation XABs linked to the FAB you use to create the file. Within each allocation XAB are two fields -- alignment (ALN) and location (LOC). If you set the symbolic value XB$LBN in the ALN field, and a numeric value in the LOC field, RMS-11 considers the contents of the LOC field as a logical block number on the volume. RMS-11 then attempts to allocate the initial virtual block of the area represented by the XAB at the specified logical block. The same allocation XAB allows you to specify the initial allocation size (ALQ) and, for indexed files only, a default extension quantity (DEQ) for the area.

Either with or without the placement control facility, you can use allocation XABs to control the internal structure of an indexed file. You can define multiple areas within such a file, assign the index and data levels associated with keys to particular areas, and vary the size of buckets on a per-area basis. Indeed, varying the size of buckets among different areas of an indexed file can be the primary motivation for your use of allocation XABs. For example, if your indexed file will contain large data records, you may want particularly large buckets for the data level of the primary key. However, you may not want to use the same size buckets for the index levels of the key. Conversely, for another indexed file, you might want index level buckets that are larger than data level buckets in order to minimize the number of index levels that RMS-11 must traverse to access any record.

Both facilities provided by allocation XABs -- placement control and indexed file structuring -- are completely optional. If you do not require either of these facilities, you do not need allocation XABs. In the absence of allocation XABs, RMS-11 provides the following defaults:

1. When you create a new file (of any organization), RMS-11 allocates a single area (implicitly, area 0) to the file. The initial size of this area is specified in the ALQ field of the FAB and the file-level default extension quantity in the DEQ field of the FAB. The physical placement of the file on the volume is completely under the control of the host operating system.

2. For an indexed file, the absence of allocation XABs when creating a file results in the allocation of a single area (again, area 0) for the entire file and the use of a single bucket size for the entire file.

When the preceding defaults are unsuitable for a particular file, you will need allocation XABs in your program.

Once you create (via $CREATE) a particular file using allocation XABs, you can also use such XABs with the $EXTEND, $OPEN, and $DISPLAY macro calls. Table 7-5 lists the fields in an allocation XAB. The subsections that follow describe the function of each field with each of the macro calls.

Table 7-5
Allocation XAB Fields

Field Name	Field Size	Description
AID	1 byte	Area identification number.
ALN	1 byte	Alignment boundary type.
ALQ	2 words	Allocation quantity.
AOP	1 byte (bit string)	Allocation option.
BKZ	1 byte	Bucket size.
COD	1 byte	Code field. Contains the value XB$ALL.
DEQ	1 word	Default area extension quantity.
LOC	2 words	Allocation starting point.
NXT	1 word	Next XAB.
VOL	1 word	Relative volume number.

7.6.1 AID - Area Identification Number

The area identification (AID) field identifies the area of the file described by the allocation XAB. You are always responsible for the contents of this field. It is never set by RMS-11. Rather, RMS-11 uses this field to:

1. Check the sequencing of allocation XABs in an XAB chain. For all operations (i.e., $CREATE, $EXTEND, $OPEN, $DISPLAY), allocation XABs in an XAB chain must appear in ascending order, based on the contents of the AID field in each.

2. Identify the target area for a specific operation (e.g., $CREATE, $EXTEND, etc.).

You can initialize the AID field at assembly-time with the X$AID macro. Its format is:

```
X$AID    area
```

where

area — is a numeric value indicating which area of the file is described by the current block. Area identification numbers range from 0 to 254. If the organization of the file is sequential or relative, only a single allocation XAB can be used for any operation and its AID field must contain 0. The assembly-time default for this field is 0.

In the following example, the user establishes an allocation XAB for area 1 of an indexed file:

```
AR1:  XAB$B  XB$ALL
        .
        .
        .
      X$AID  1
        .
        .
        .
      XAB$E
```

7.6.2 ALN - Alignment Boundary Type

The alignment boundary type (ALN) field specifies the type of alignment requested for the area described by the block. If you require placement control over the initial allocation ($CREATE) or explicit extension ($EXTEND) of the area, you use this field to specify whether the LOC field contains a logical block number or a virtual block number. If you do not wish to exercise placement control during a $EXTEND or $CREATE operation, you set this field equal to zero.

When allocation XABs are present during a $OPEN or $DISPLAY operation, RMS-11 sets the ALN field equal to the value specified for the corresponding area when the file was created.

You can use the X$ALN macro to initialize the ALN field at assembly-time. Its format is:

```
X$ALN    type
```

where

type is a symbolic value or 0. One of the following symbolic values can be specified:

XB$LBN - align area (or extension to area) on or near logical block described by LOC field.

XB$VBN - perform allocation as near as possible to the virtual block described by LOC field. If LOC contains 0, RMS-11 will allocate space as near as possible to the current high virtual block of the file. If the AID field of the block specifies area 0, the value XB$VBN is invalid during a $CREATE operation.

RMS-11 interprets a value of 0 in the ALN field as meaning no placement control is being exercised by the user. The assembly-time default is 0.

In the following example, the user indicates that no placement control is described by the current allocation XAB.

```
AR1:  XAB$B  XB$ALL
        .
        .
        .
      X$ALN  0
        .
        .
        .
      XAB$E
```

7.6.3 ALQ - Allocation Quantity

The allocation quantity (ALQ) field specifies the number of virtual blocks in the initial allocation of the area (when the allocation XAB is used with the $CREATE operation) or the number of virtual blocks by which the area is to be extended (when the block is part of a $EXTEND operation). For both operations, the value you specify in this field overrides the contents of the ALQ field in the FAB. RMS-11 neither examines nor sets this field during $OPEN and $DISPLAY operations.

The X$ALQ macro allows you initialize the ALQ field in this block at assembly-time. The format of this macro is as follows:

```
X$ALQ    quantity
```

where

quantity is a numeric value in the range of 0 to 16,777,215 representing a number of virtual blocks. If, during a $CREATE operation, the user-specified quantity in the ALQ field for allocation area 0 is less than the number required to store necessary file attributes, RMS-11 ignores the user-specified value and allocates a sufficient number of virtual blocks.

In the following example of the X$ALQ macro, the user specifies that 400 virtual blocks are to be allocated to the area described by the block:

```
AR1:  XAB$B  XB$ALL
        .
        .
        .
      X$ALQ  400
        .
        .
        .
      XAB$E
```

7.6.4 AOP - Allocation Options

The allocation options (AOP) field allows you to qualify the action RMS-11 is to perform when the block is present during a $CREATE or $EXTEND operation. For $OPEN and $DISPLAY operations, RMS-11 returns into this field the options specified when the area was created (via $CREATE).

The X$AOP macro can be used to initialize the AOP field at assembly-time. Its format is:

```
X$AOP    option[!option]
```

where

option — is a symbolic value representing an option to be applied during a $CREATE or $EXTEND operation. You can specify either or both of the following:

- XB$HRD - Hard request. This option should be specified only if the ALN field contains the value XB$LBN. RMS-11 returns an error code if the initial allocation ($CREATE) or extension ($EXTEND) of the area cannot be aligned on the logical block specified by the LOC field. The default is to perform the allocation as close as possible to the requested alignment.

- XB$CTG - Contiguous allocation requested. The assembly-time default is non-contiguous.

In the following example of the X$AOP macro, the user specifies a hard request for a contiguous allocation:

```
AR1:  XAB$B  XB$ALL
        .
        .
        .
      X$AOP  XB$CTG!XB$HRD
        .
        .
        .
      XAB$E
```

7.6.5 BKZ - Bucket Size

The BKZ field specifies the size of the buckets in the area described by the allocation XAB. This field is used for indexed files only. During a $CREATE operation, the value you specify in this field supersedes the contents of the BKS field in the FAB. During $OPEN and $DISPLAY operations, RMS-11 returns into this field the value specified when the area was created. The primary purpose of this field is to allow you to vary the size of buckets among the multiple areas of an indexed file.

The X$BKZ macro allows you to initialize the BKZ field at assembly-time. The format of this macro is as follows:

```
X$BKZ    bucket-size
```

where

bucket-size — is a numeric value, in the range of 0 to 32, representing the number of virtual blocks contained in each bucket in the area of the file. The assembly-time default is 0. RMS-11 interprets a value of 0 as identical to 1.

The following is an example of the X$BKZ macro showing the specification of four virtual blocks per bucket:

```
AR1:  XAB$B
        .
        .
        .
      X$BKZ  4
        .
        .
        .
      XAB$E
```

7.6.6 DEQ - Default Area Extension Quantity

The default area extension quantity (DEQ) field applies to indexed files only. It specifies the number of virtual blocks to be used when RMS-11 must automatically extend the area described by the XAB. This automatic extension occurs whenever your program attempts a $PUT or $UPDATE operation that cannot be accommodated within the space currently allocated to the area.

During a $CREATE operation, RMS-11 examines the value you specified in this field. If this value is nonzero, RMS-11 saves it as the default extension quantity for the area defined by the XAB. If, however, the DEQ field in this block is zero, RMS-11 saves the FAB's DEQ as the default extension quantity for the area defined by the XAB.

The DEQ field in the allocation XAB is neither examined nor set during a $EXTEND operation. During a $OPEN or $DISPLAY operation, however, RMS-11 returns into this field the value specified when the area was created.

You can use the X$DEQ macro to initialize the DEQ field of an allocation XAB at assembly-time. The format of this macro is:

```
X$DEQ    quantity
```

where

quantity is a numeric value representing a number of virtual blocks. This number must be in the range of from 0 to 65,535. The quantity you specify should be a multiple of the area's bucket size. The assembly-time default is 0.

In the following example, the user specifies that, when automatic extension is necessary, RMS-11 is to extend the area by 80 virtual blocks.

```
AR1:  XAB$B
        .
        .
        .
      X$DEQ  80
        .
        .
        .
      XAB$E
```

7.6.7 LOC - Allocation Starting Point

The allocation starting point (LOC) field is used by RMS-11 only during $CREATE and $EXTEND operations. Further, the contents of this field are ignored during such operations if the ALN field of the same block is zero. When, however, the ALN field contains XB$LBN, RMS-11 interprets the contents of LOC as the starting logical block number on the volume for the initial allocation ($CREATE) or extension ($EXTEND) of the area. The ALQ field of the same block contains the number of blocks to be allocated and the AOP field can optionally specify (XB$HRD) that the allocation must begin with the specified logical block.

When the ALN field contains XB$VBN during a $CREATE or $EXTEND operation, RMS-11 interprets the contents of LOC as a virtual block number within the file. For $CREATE operations, LOC must contain zeroes since no virtual blocks as yet exist in the file. However, during $EXTEND operations, LOC can contain the number of a virtual block in the file near which the extension to the area is to be allocated. Again, you use the ALQ field to specify the size of the extension. Note, however, that the allocation options (AOP) field cannot contain XB$HRD.

During $OPEN and $DISPLAY operations, RMS-11 neither sets nor examines the LOC field.

The X$LOC macro allows you to initialize the LOC field at assembly-time. Its format is:

```
X$LOC    number
```

where

number is a numeric value interpreted as follows:

1. If ALN contains XB$LBN, the specified number is the starting logical block for the allocation. The specified number may vary from 0 to the maximum number of blocks on the volume.

2. If ALN contains XB$VBN, the specified number is a virtual block within the file. The specified number may vary from 0 to the number of virtual blocks in the file. A value of 0 (assembly-time default) implies that allocation should occur as near as possible to the current end of file.

The following is an example of the X$LOC macro:

```
AR1:  XAB$B
        .
        .
        .
      X$AOP  XB$HRD
      X$ALN  XB$LBN
      X$LOC  1024
        .
        .
        .
      XAB$E
```

In this example, the user specifies that allocation of an extent for the area represented by the XAB must begin on logical block number 1024.

7.6.8 VOL - Relative Volume Number

The relative volume number (VOL) field must contain 0. The assembly-time default for this field is 0.

7.7 SUMMARY EXTENDED ATTRIBUTE BLOCK

The summary XAB allows you to determine the number of keys defined for an existing indexed file and/or the number of allocation areas defined for an existing file.

You never use this type of XAB with a $CREATE macro call. However, a summary XAB can be associated with a FAB at the time a $OPEN or $DISPLAY macro call is issued. The presence of this XAB during these macro calls allows RMS-11 to return to your program the total number of keys and allocation areas defined when the file was created.

Table 7-6 describes the fields of a summary XAB.

Table 7-6
Summary XAB Fields

Field Name	Field Size	Description
COD	1 byte	Code field. Contains the value XB$SUM.
NOA	1 byte	Number of allocation areas defined for the file.
NOK	1 byte	Number of keys defined for the file.
NXT	1 word	Next XAB.

CHAPTER 8

THE NAME BLOCK

The Name Block is an optional structure. After a successful $CREATE, $OPEN, or $ERASE macro call, a location described by this block contains the full file specification resulting from RMS-11's merger of the default file name string (described by the DNA and DNS fields of the associated FAB) with the primary name string (described by the FNA and FNS fields of the associated FAB) and system defaults. You indicate that this service is desired by assuring that the NAM field of the FAB contains the address of a Name Block at the time a $CREATE, $OPEN, or $ERASE macro is issued.

The following subsections describe the allocation of a Name Block and assembly-time initialization macros for fields in the block.

8.1 ALLOCATING A NAME BLOCK

The NAM$B macro allocates space for a Name Block and delimits the beginning of an optional sequence of assembly-time initialization macros for the fields of the block.

The format of the NAM$B macro is as follows:

```
label:NAM$B
```

where

label — is a user-specified symbol that names this particular Name Block. You must ensure that the address assigned to this label is word-aligned. Therefore, a .EVEN directive should immediately precede the NAM$B macro.

The NAM$E macro delimits the end of the optional sequence of assembly-time initialization macros initiated by NAM$B and stores any specified values in the appropriate fields of the block. This macro must always appear subsequent to an associated NAM$B macro, even when no intervening initialization macros are coded.

The NAM$E macro takes the following form:

```
NAM$E
```

The following example shows the allocation of a Name Block called FNAME:

```
        .EVEN
FNAME:  NAM$B
        NAM$E
```

8.2 FIELDS IN THE NAME BLOCK

Table 8-1 summarizes the fields in a Name Block.

Table 8-1
Name Block Fields

Field Name	Field Size	Description
ESA	1 word	Expanded string address
ESL	1 byte	Expanded string length
ESS	1 byte	Expanded string size

The subsections that follow describe the fields listed in Table 8-1.

8.2.1 ESA - Expanded String Address

The expanded string address (ESA) field contains the address of a user-allocated buffer. Following a $OPEN, $CREATE, or $ERASE macro call, RMS-11 places in this buffer the file specification string resulting from the application of default information (provided by the default name string of the FAB and system defaults) to the original file string (file name string of the FAB). The ESA buffer must be present if the block itself is input to an operation (i.e., the NAM field of the FAB is nonzero).

The N$ESA macro allows you to initialize the ESA field at assembly-time. The following is the format of the N$ESA macro:

```
N$ESA address
```

where

address — is the symbolic address of a buffer in the user program.

The following is an example of the N$ESA macro:

```
NAMBUF: .BLKB   32.
          .
          .
          .
FNAME:  NAM$B
          .
          .
          .
        N$ESA   NAMBUF
          .
          .
          .
        NAM$E
```

8.2.2 ESL - Expanded String Length

The expanded string length (ESL) field is set by RMS-11 after a $OPEN, $CREATE, or $ERASE macro call. This field contains the actual length of the expanded file specification whose address is in the ESA field.

8.2.3 ESS - Expanded String Size

The expanded string size (ESS) field contains the size of the user-allocated buffer whose address is in the ESA field of the block.

You can initialize the ESS field at assembly-time with the N$ESS macro. The format of the N$ESS macro is as follows:

```
N$ESS size
```

where

size	is a numeric value representing the size (in bytes) of the buffer whose address is in the ESA field. The specified value must range from 1 to 255.

In the following example, the user specifies an expanded string area of 32 bytes:

```
FNAME:  NAM$B

        N$ESA   NAMBUF

        N$ESS   32

        NAM$E
```

CHAPTER 9

PERFORMING FILE AND RECORD OPERATIONS

This chapter describes the RMS-11 macros used to access and manipulate files and records within files.

As described in Chapter 2, file and record processing macros in combination with user control blocks form the runtime program interface with RMS-11. The primary argument of a file processing macro call is the address of a File Access Block. The primary argument of a record processing macro call is the address of a Record Access Block. Certain fields within these blocks must contain appropriate values at the time a particular file or record processing macro call is issued. Following such macro calls, RMS-11 returns information to your program within fields of the same control block. The detailed descriptions of file and record processing macros in this chapter, therefore, list all control block fields used as input to a macro call and every field that may contain information returned to your program by RMS-11.

The remainder of this chapter is divided into three sections:

1. File and record operation macro conventions (i.e., macro formats, the RMS-11 calling sequence, completion routines, control block field usage, and status codes).

2. Performing file operations (i.e., $CREATE, $OPEN, $DISPLAY, $ERASE, $EXTEND, $CLOSE).

3. Performing record operations (i.e., record access streams, record operations and file sharing, current context of record operations, synchronous and asynchronous operations, accessing records, and record operation macros).

NOTE

When RMS-11 is executing, asynchronous system traps (ASTs) are disabled. When RMS-11 returns to your program, ASTs are enabled.

9.1 FILE AND RECORD OPERATION MACRO CONVENTIONS

While differing in function, file and record operation macros have the same general format. Furthermore, each macro call for an RMS-11 service requires the establishment of a standard calling sequence. Within this calling sequence, you can optionally specify the addresses of completion routines.

Before issuing the actual file or record operation macro call, your program must ensure that the control block (i.e., FAB or RAB) contains appropriate values in the fields that will be examined by RMS-11. Following execution of the macro call, your program should examine the status code returned by RMS-11 in the control block to ascertain the success or reason for failure of the operation. The following subsections, therefore, describe:

- The format of file and record processing macros
- The RMS-11 calling sequence
- Completion routine conventions
- Control block field usage
- Status codes

9.1.1 Format of File and Record Operation Macros

You can code file and record operation macros in either of the following two formats:

1. label:$macro

   or

2. label:$macro block[,error[,success]]

where

label — is an optional user-defined symbol referring to the macro.

$macro — is one of the file or record operation macros described in this chapter.

block — is the address of a File Access Block (for file operations) or a Record Access Block (for record operations.)

error — is the address of a user completion routine to be called if the requested operation fails.

success — is the address of a user completion routine to be called if the requested operation completes successfully. This argument is ignored for file processing macro calls.

In the second format, the presence of arguments causes the macro expansion to build an argument list on your program's stack. In the first format above, however, no arguments appear following the macro name. In this format, therefore, you must create the argument list in your program. The following subsection describes this process.

NOTE

The first format of RMS-11 file and record operation macros (i.e., without an argument list) generates less code and uses less stack space than the second format.

9.1.2 The RMS-11 Calling Sequence

At the time an RMS-11 routine is called, general register R5 must contain the address of a word-aligned formatted argument list. Figure 9-1 shows the format of this list:

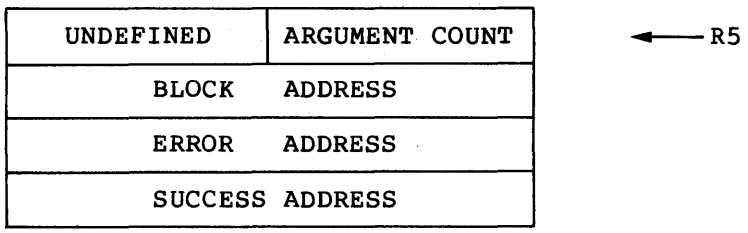

Figure 9-1 Argument List Format

RMS-11 interprets the fields within the argument list as follows:

Argument Count	is a binary value from 1 to 3 representing the number of arguments present in the argument list. This count field must equal 1 if the user does not specify completion routine addresses.
Block Address	is the address of a File Access Block (for file operations) or a Record Access Block (for record operations).
Error Address	is the address of a user completion routine to be called if the requested operation fails.
Success Address	is the address of a user completion routine to be called if the requested operation completes successfully. This argument is ignored for file processing macro calls.

The preceding argument list is automatically generated whenever you specify a file or record processing macro with arguments. When specifying these macros without arguments, your program must construct the argument list. For example, if you want to read a record from a file and specify both an error and a success completion routine, the equivalent of the following sequence must be coded:

```
        MOV #LIST,R5            ;ADDRESS OF ARGUMENT
                                ;LIST TO REGISTER 5
        $GET                    ;RECORD PROCESSING MACRO
                                ;TO READ A RECORD
        ...

LIST:.WORD      3               ;NUMBER OF ARGUMENTS
     .WORD      INRAB           ;RECORD ACCESS BLOCK ADDRESS
     .WORD      ERR1            ;ERROR ROUTINE ADDRESS
     .WORD      SUCC1           ;SUCCESS ROUTINE ADDRESS
```

If you wanted to specify a success routine without specifying an error routine, you would use the following code for the argument list:

```
LIST:.WORD      3               ;NUMBER OF ARGUMENTS
     .WORD      INRAB           ;RECORD ACCESS BLOCK ADDRESS
     .WORD      -1              ;-1 MEANS NO ERROR ROUTINE
     .WORD      SUCC1           ;SUCCESS ROUTINE ADDRESS
```

If neither a success nor an error routine is specified, the argument list is constructed as follows:

```
LIST:.WORD     1           ;NUMBER OF ARGUMENTS
     .WORD     INRAB       ;RECORD ACCESS BLOCK ADDRESS
```

Finally, if only an error routine is specified, you would construct the argument list as follows:

```
LIST:.WORD     2           ;NUMBER OF ARGUMENTS
     .WORD     INRAB       ;RECORD ACCESS BLOCK ADDRESS
     .WORD     ERR1        ;ERROR ROUTINE ADDRESS
```

Regardless of the macro format used to invoke an RMS-11 routine, RMS-11 preserves all general registers (R0-R5) across a macro call.

9.1.3 Completion Routine Conventions

The use of completion routines is always optional. If a corresponding address is present in the argument list, RMS-11 invokes an error or success completion routine based on the results of the operation attempted. When employing completion routines, you must be aware of conventions in the areas of:

- Register usage
- Issuing RMS-11 macro calls within completion routines
- Returning from a completion routine

The subsections that follow discuss each convention.

9.1.3.1 Register Usage Conventions Within Completion Routines - When RMS-11 calls a user-specified completion routine, the following register conventions are used:

1. General register R5 contains the address of the same argument list, or a copy of the argument list, that was part of the calling sequence to RMS-11 itself. Therefore, you can use the control block address at 2(R5) to access fields of the control block.

2. General registers R0-R4 are undefined.

9.1.3.2 Issuing RMS-11 Macro Calls Within Completion Routines - As part of a completion routine, your program can issue an additional RMS-11 file or record processing macro call. RMS-11 considers each such additional macro call as an extension of the original request that caused the completion routine to be invoked. The only restriction RMS-11 enforces is that a synchronous operation request cannot be issued within a completion routine called as a result of an asynchronous record operation.

9.1.3.3 **Returning From a Completion Routine** - To return control from a completion routine to RMS-11, your program must:

1. Restore the stack pointer SP to its original value at the time of entry to the completion routine. Your program must not attempt to cause control flow changes by modifying the stack.

2. Issue a $RETURN macro.

The format of the $RETURN macro is as follows:

```
label:$RETURN
```

where

label — is an optional user-defined macro referring to this $RETURN macro.

9.1.4 Control Block Field Usage

The fields within the control block associated with a particular macro call are the means by which you qualify or further define the requested file or record operation. For each particular operation, some number of control block fields will be used as input by RMS-11. The tables that accompany each macro description in this chapter enumerate these fields. You must ensure, before your program issues a macro call, that the appropriate fields in your control block contain the necessary values.

You have three choices on how to set values in control block fields:

1. Assembly-time initialization.

2. Assembly-time defaulting.

3. Runtime initialization.

At assembly-time, you can explicitly initialize a field or, when a default is defined, allow the assembly-time expansion of the block allocation macros to set a default value in the field. At runtime, you can initialize or alter a control block field through use of the $STORE, $SET, or $OFF macros. You must understand, however, that there are no runtime defaults for any field in any control block. If you fail to set (at assembly-time or runtime) every field in a control block that is defined as input to a particular operation, the operation may fail.

Consider the following example. You write a program that creates an indexed file and a sequential file. To save storage, you decide to allocate a single File Access Block. Therefore, your program will first create one file, close it, and then reuse the FAB to create the second file. You allocate a single key definition XAB (assuming your indexed file has only a single key). To create the indexed file first, you initialize all the appropriate fields of the FAB (e.g., ORG contains FB$IDX, RFM contains, perhaps, FB$VAR, etc.). You place the address of the key definition XAB in the XAB field of the FAB. Then you define the primary key of the file through the fields of the XAB and you set the NXT (next XAB) field in the XAB to zero to indicate that there are no additional XABs. Finally, you issue the $CREATE macro call and, if RMS-11 indicates successful completion, you close the new file by issuing a $CLOSE macro call.

Now your program is ready to create the sequential file. Using the $STORE macro, you alter the ORG field of the FAB so that it contains the value FB$SEQ. You further alter additional fields (RFM, FNA, FNS, MRS, etc.) so that they accurately describe your sequential file. However, you neglect to zero the XAB field in the FAB. When you issue the second $CREATE macro call, RMS-11 will find a valid address in the XAB field. It will then examine the XAB addressed by this field and find that it is a key definition XAB. Since sequential files do not have keys, RMS-11 will reject the create operation and return an error status code to your program.

As the preceding example shows, you must appropriately set every field in a control block that RMS-11 may use during execution of an operation. RMS-11 assumes that every value found in an input field was intentionally placed there for use during the current operation. Since there are no runtime defaults for control block fields, you are responsible for the contents of all input fields prior to issuing a file or record operation macro call.

9.1.5 Status Codes

Before returning to your program from a file or record operation macro call, RMS-11 always indicates the success or failure of the requested operation by setting the STS field of the control block associated with the call.

Through the use of the $COMPARE macro and the mnemonic status codes listed in Appendix A, your program can check for successful completion (SU$SUC) or particular error conditions.

While particular error status codes are mentioned in certain sections, there is no attempt to list all possible error codes that can occur for every individual file or record operation. Such lists would tend to duplicate most of the list found in Appendix A. In writing programs, therefore, you must examine Appendix A and determine which error conditions you particularly want to test for following a macro call. For example, in reading a file, you always want to be prepared for the ER$EOF error status code (end of file). Whether you test for other error codes, such as ER$RLK (bucket is locked by another program or another stream in your own program) depends on the requirements and logic of your program.

With certain error status codes, RMS-11 returns to your program additional information in the STV field of the control block associated with an operation. The descriptions of error status codes in Appendix A indicate those instances in which the STV field contains such information.

9.2 PERFORMING FILE OPERATIONS

A file operation macro call causes RMS-11 to perform some action related to an entire file. The macro call itself (e.g., $CREATE, $OPEN, $CLOSE) indicates the type of action desired. The fields of the File Access Block associated with the macro call identify the file or further qualify the requested action.

Table 9-1 summarizes the file operation macros provided by RMS-11:

Table 9-1
RMS-11 File Operation Macros

Macro Name	Description
$CREATE	Creates and opens a new RMS-11 file of any organization.
$OPEN	Opens an existing RMS-11 file making its contents available for processing.
$DISPLAY	Returns attributes of an RMS-11 file to the user program.
$ERASE	Deletes an RMS-11 file and removes its entry from a directory.
$EXTEND	Extends the allocated space of an RMS-11 file.
$CLOSE	Closes an open RMS-11 file.

The subsections that follow describe the action caused by each file operation macro and the control block fields that are input to and output from each operation.

9.2.1 $CREATE - Creating an RMS-11 File

The $CREATE macro creates a new RMS-11 file with the attributes you specify in the File Access Block and any Extended Attribute Blocks chained to the FAB. When key definition or allocation XABs are present, they must be grouped in densely ascending order (i.e., 0, 1, 2, 3, 4... in the REF or AID fields) without gaps or intervening XABs of other types. Otherwise, RMS-11 returns an ER$ORD error code. An illogical XAB type in the chain (e.g., a summary XAB or a key definition XAB for a sequential file) causes the ER$COD error condition. If a NAM Block is also chained to the FAB, RMS-11 sets the ESL field of the block and places the expanded file specification in the area addressed by the ESA field of the NAM block.

The $CREATE operation leaves the file open. Therefore, you must close the file by issuing a $CLOSE macro call.

NOTE

The $CREATE operation requires 1 BDB and 512 bytes of I/O buffer space, all of which are released when the operation completes. You reserve space for BDBs through the use of the P$BDB macro (refer to Chapter 3) and space for I/O buffers through the P$BUF macro (refer to Chapter 3) or the BPA and BPS fields of the FAB (refer to Chapter 5). The calculation of BDB and buffer space requirements is related to the

performance of record operations on the file. Since any record operations require at least 1 BDB and a buffer at least 1 virtual block in size, the BDB and buffer requirements for this file operation are typically met without specific attention on the part of the user.

The formats of the $CREATE macro are as follows:

1. label:$CREATE

2. label:$CREATE fab[,error]

where

label — is an optional user-defined symbol referring to the $CREATE macro.

fab — is the address of a File Access Block representing the file to be created.

error — is the address of a user completion routine to be called if the $CREATE operation fails.

Table 9-2 lists the fields of the File Access Block used during the $CREATE macro call.

Table 9-2
$CREATE FAB Fields

	Name	Description
Input	ALQ	Allocation quantity (this field is ignored if one or more allocation XABs are present).
	BKS	Bucket size (this field is ignored for indexed files if one or more allocation XABs are present).
	BLS	Block size (magnetic tape files only).
	BPA	Buffer pool address.
	BPS	Buffer pool size.
	DEQ	Default file extension quantity.
	DNA	Default name string address.
	DNS	Default name string size.
	FAC	File access (must contain at least the value FB$PUT).
	FNA	File name string address.
	FNS	File name string size.

(Continued on next page)

Table 9-2 (Cont.)
$CREATE FAB Fields

	Name	Description
Input (Cont.)	FOP	file processing options.
	FSZ	Fixed control area size (files with VFC format records only).
	LCH	Logical channel number.
	MRN	Maximum record number (relative files only).
	MRS	Maximum record size.
	NAM	Name block pointer.
	ORG	File organization.
	RAT	Record attributes.
	RFM	Record format.
	RTV	Retrieval window size.
	SHR	File sharing.
	XAB	Extended Attribute Block pointer.
Output	DEV	Device characteristics.
	IFI	Internal file identifier.
	STS	Completion status code.
	STV	Status value.

9.2.2 $OPEN - Opening an Existing File for Processing

The $OPEN macro makes an existing file available for processing by your program. RMS-11 returns the basic attributes of the file in the fields of the File Access Block associated with the request. If Extended Attribute Blocks are chained to the FAB, RMS-11 fills in the attribute information represented by each XAB (e.g., key definition XAB, protection XAB, etc.). When either key definition or allocation XABs are present, they must be grouped in ascending order (by REF or AID, respectively) but need not be dense. No other intervening types are permitted. If this sequencing is violated, RMS-11 returns an ER$ORD error. Further, RMS-11 does not verify that the number of key definition or allocation XABs does not exceed the number of actual keys or areas defined for the file. Any such excess XABs are ignored. An illogical XAB type in the chain (e.g., key definition XAB for a sequential file) is similarly ignored. Additionally, if a NAM Block is chained to the FAB, RMS-11 sets the ESL field in the NAM Block and places the expanded file specification of the open file in the location specified by the ESA field of the NAM Block.

NOTE

The $OPEN operation requires 1 BDB and 512 bytes of I/O buffer space, all of which are released when the operation completes. You reserve space for BDBs through the use of the P$BDB macro (refer to Chapter 3) and space for I/O buffers through the P$BUF macro (refer to Chapter 3) or the BPA and BPS fields of the FAB (refer to Chapter 5). The calculation of BDB and buffer space requirements is related to the performance of record operations on the file. Since any record operations require at least 1 BDB and a buffer at least 1 virtual block in size, the BDB and buffer requirements for this file operation are typically met without specific attention on the part of the user.

The formats of the $OPEN macro are:

1. label:$OPEN

2. label:$OPEN fab[,error]

where

label — is an optional user-defined symbol referring to the $OPEN macro.

fab — is the address of a File Access Block representing a file to be opened.

error — is the address of a user completion routine to be called if the $OPEN operation fails.

Table 9-3 lists the fields of the File Access Block used during the $OPEN macro call.

Table 9-3
$OPEN FAB Fields

	Name	Description
Input	BPA	Buffer pool address.
	BPS	Buffer pool size.
	DEQ	Default extend quantity.
	DNA	Default name string address.
	DNS	Default name string size.
	FAC	File access.

(Continued on next page)

Table 9-3 (Cont.)
$OPEN FAB Fields

	Name	Description
Input (Cont.)	FNA	File name string address.
	FNS	File name string size.
	FOP	File processing options.
	LCH	Logical channel number.
	NAM	Name block pointer.
	RTV	Retrieval window size.
	SHR	File sharing.
	XAB	Extended Attribute Block Pointer. This field is optional. If present, RMS-11 will fill the indicated block (and any XABs chained to the first block) with the file attribute information indicated by the type of block present.
Output	ALQ	Allocation quantity. Contains the highest numbered virtual block allocated to the file.
	BKS	Bucket size. Supplied for relative and indexed files and zeroed for sequential files. When multiple areas have been defined for an indexed file, BKS contains the size of the largest bucket in the file.
	BLS	Block size (sequential files on magnetic tape only).
	DEV	Device characteristics.
	FOP	File processing options. Contains FB$CTG if the file is contiguous.
	FSZ	Fixed control area size. Supplied only if record format is VFC.
	IFI	Internal file identifier.
	MRN	Maximum record number. Supplied only if file organization is relative.
	MRS	Maximum record size.
	ORG	File organization.
	RAT	Record attributes.
	RFM	Record format.
	STS	Completion status code.
	STV	Status value.

9.2.3 $DISPLAY - Obtaining Attributes of a File

The $DISPLAY macro retrieves file attribute information that you request and places the information in the fields of Extended Attribute Blocks. You must open the file in order to obtain attributes.

RMS-11 determines the type of attribute information desired through examination of the types of Extended Attribute Blocks associated with the File Access Block by the XAB field. When either key definition or allocation XABs are present, they must be grouped in ascending order (by REF or AID, respectively) but need not be dense. No other intervening types are permitted. If this sequencing is violated, RMS-11 returns an ER$ORD error. Further, RMS-11 does not verify that the number of key definition or allocation XABs does not exceed the number of actual keys or areas defined for the file. Any such excess XABs are ignored. An illogical XAB type in the chain (e.g., key definition XABs for a sequential file) are similarly ignored.

NOTE

For the duration of its operation, $DISPLAY requires 1 BDB (refer to the P$BDB macro description in Chapter 3) and 512 bytes of I/O buffer space (refer to the P$BUF macro description in Chapter 3 or the BPS field description in Chapter 5). These requirements are in addition to the BDB and buffer space requirements for any streams that are connected at the time the $DISPLAY call is issued.

The formats of the $DISPLAY macro are as follows:

1. label:$DISPLAY

2. label:$DISPLAY fab[,error]

where

label — is an optional user-defined symbol referring to the $DISPLAY macro.

fab — is the address of a File Access Block representing a file whose attributes are to be returned to the user program.

error — is the address of a user completion routine to be called if the $DISPLAY operation fails.

Table 9-4 lists the fields of the File Access Block used during the $DISPLAY macro call.

Table 9-4
$DISPLAY FAB Fields

	Name	Description
Input	XAB	Extended Attribute Block pointer.
	IFI	Internal file identifier.
Output	STS	Completion status code.
	STV	Status value. Contains a system error code or the address of the XAB that caused an error.

9.2.4 $ERASE - Deleting a File

The $ERASE macro deletes an RMS-11 file and removes its directory entry. The space occupied by the file is returned at a system dependent time. You can issue a $ERASE operation for a file currently accessed by another program or by your own program (so long as you are accessing it on a different logical channel). The file is not deleted, however, until all accessors close it. You cannot delete files on unit record or sequential (e.g., magnetic tape) devices.

NOTE

The $ERASE operation requires 1 BDB and 512 bytes of I/O buffer space, all of which are released when the operation completes. You reserve space for BDBs through the use of the P$BDB macro (refer to Chapter 3) and space for I/O buffers through the P$BUF macro (refer to Chapter 3) or the BPA and BPS fields of the FAB (refer to Chapter 5). The calculation of BDB and buffer space requirements is related to the performance of record operations on the file. Since any record operations require at least 1 BDB and a buffer at least 1 virtual block in size, the BDB and buffer requirements for this file operation are typically met without specific attention on the part of the user.

The formats of the $ERASE macro are:

1. label:$ERASE

2. label:$ERASE fab[,error]

where

label — is an optional user-defined symbol referring to the $ERASE macro.

fab — is the address of a file access block representing a file to be deleted.

error — is the address of a user completion routine to be called if the $ERASE operation fails.

Table 9-5 lists the fields of the File Access Block used during the $ERASE macro call.

Table 9-5
$ERASE FAB Fields

	Name	Description
Input	BPA	Buffer pool address.
	BPS	Buffer pool size.
	DNA	Default name string address.
	DNS	Default name string size.
	FNA	File name string address.
	FNS	File name string size.
	LCH	Logical channel number.
Output	STS	Completion status code.
	STV	Status value.

9.2.5 $EXTEND - Extending Allocated Space

The $EXTEND macro extends the amount of space allocated to an RMS-11 file. The file must be open and any record access streams connected to the file must be inactive before the extension is attempted. Before you open the file, the FAC field in your FAB must specify at least one type of write operation (i.e., FB$PUT, FB$UPD, or FB$DEL). You cannot extend a file residing on magnetic tape.

If no allocation XABs are present, RMS-11 extends, by default, area 0 of the file or (for sequential or relative files) the file itself. The allocation quantity field (ALQ) of the File Access Block must contain the number of virtual blocks to be added to the file. In the FOP field of the FAB, you can request a contiguous extent (FB$CTG). If RMS-11 cannot allocate a contiguous extent, the $EXTEND operation will fail.

If you created an indexed file using allocation XABs (refer to Section 7.6 in Chapter 7), you can selectively extend one or more areas of the file by providing appropriate allocation XABs as input to the $EXTEND operation. In this case, RMS-11 obtains the amount by which each area is to be extended from the ALQ field of each XAB. If you attempt to extend an area that has a currently unused extent, RMS-11 returns on ER$LEX error. Allocation XABs, when present, must be in ascending order by AID (area identification number) but need not be dense.

The formats of the $EXTEND macro are:

1. label:$EXTEND

2. label:$EXTEND fab[,error]

where

label — is an optional user-defined symbol referring to the $EXTEND macro.

fab — is the address of a file access block representing a file to be extended.

error — is the address of a user completion routine to be called if the $EXTEND operation fails.

Table 9-6 lists the field of the File Access Block used during the $EXTEND macro call.

Table 9-6
$EXTEND FAB Fields

	Name	Description
Input	ALQ	Allocation quantity. Ignored if one or more allocation XABs are present.
	FOP	File processing options. You can specify FB$CTG. This value will be ignored, however, if one or more allocation XABs are present.
	IFI	Internal file identifier.
	XAB	Extended Attribute Block Pointer (optional).
Output	ALQ	Allocation quantity. This field will contain the actual number of virtual blocks added to the file.
	STS	Completion status code.
	STV	Status value.

9.2.6 $CLOSE - Terminating File Processing

The $CLOSE macro closes an open RMS-11 file.

You should issue $DISCONNECT macro calls (refer to Section 9.3.1.2) for each record access stream associated with a file before issuing the $CLOSE macro. If you do not issue these $DISCONNECTs, RMS-11 will effectively disconnect all the file's record access streams. However, the $CLOSE operation will fail if there is an outstanding I/O request on any such stream. Further, the ISI (internal stream identifier) field in each RAB representing a stream will not be zeroed. Only the $DISCONNECT macro call zeroes this field. This action is desirable since it ends the association of a RAB with an internal RMS-11 structure in the space pool.

If RMS-11 returns a write error (ER$WER) in the STS field of the FAB, it has still closed and deaccessed the file.

The formats of the $CLOSE macro are as follows:

1. label:$CLOSE

2. label:$CLOSE fab[,error]

where

label — is an optional user-defined symbol referring to the $CLOSE macro.

fab — is the address of a File Access Block representing the file to be closed.

error — is the address of a user completion routine to be called if the $CLOSE operation fails.

Table 9-7 lists the fields of the File Access Block used during the $CLOSE macro call.

Table 9-7
$CLOSE FAB Fields

	Name	Description
Input	FOP	File processing options. You can specify FB$RWC (rewind tape volume on close).
	IFI	Internal file identifier.
Output	IFI	Internal file identifier. (Zeroed)
	STS	Completion status code.
	STV	Status value.

9.3 PERFORMING RECORD OPERATIONS

After you open a file through a $OPEN or $CREATE macro call, you can perform record operations on the file. To perform such record operations, you must understand in detail the following:

- Record Access streams
- Record operations and file sharing
- Current context of record operations
- Synchronous and asynchronous record operations
- Accessing records
- The actual record operation macros (e.g., $GET, $PUT, $UPDATE, etc.)

The following subsections describe each of these facilities.

9.3.1 Record Access Streams

To process records in a file, you must establish a record access stream. A record access stream is the logical association of a Record Access Block with a File Access Block. Once you have established a record access stream, you can issue operations on records in the file represented by the File Access Block.

RMS-11 permits only one record access stream for sequential files. Thus, when you open or create a sequential file, you can associate only one RAB with the FAB representing the file. When you open or create a relative or indexed file, RMS-11 permits you to establish multiple record access streams by associating more than one RAB with the same FAB.

When you establish a single record access stream for a file, your program uses the stream to issue a sequence of record operations. Within the stream, these record operations are executed serially. In other words, you can process only one record at a time. When you establish multiple record access streams for a file, your program can process more than one record of the file in parallel. Thus, such multiple streams represent independent and concurrently active sequences of record operations to the same file.

After you open a file by issuing a $OPEN or a $CREATE macro call, you establish a record access stream by:

1. Placing the address of the File Access Block in the FAB field of the appropriate Record Access Block. You can perform this action at runtime by using the $STORE macro or at assembly-time by using the R$FAB initialization macro.

2. Issuing a $CONNECT macro.

When you have completed the desired sequence of record operations, you terminate a record access stream by issuing a $DISCONNECT macro.

The following subsections describe the $CONNECT and $DISCONNECT macros.

9.3.1.1 **$CONNECT - Establishing a Record Access Stream** - The $CONNECT macro establishes a record access stream by associating a Record Access Block with a File Access Block. During the macro call, RMS-11 allocates I/O buffers for the stream. These buffers are allocated from the file's private buffer pool (if such a pool was described by the BPA and BPS fields of the FAB during the $OPEN or $CREATE operation) or from the centralized space pool. Additionally, RMS-11 allocates, within the centralized space pool, internal control structures needed to support the stream.

The formats of the $CONNECT macro are:

1. label:$CONNECT

2. label:$CONNECT rab[,error[,success]]

where

label — is an optional user-defined symbol referring to the $CONNECT macro.

rab — is the address of a Record Access Block to be associated with a File Access Block.

error — is the address of a user completion routine to be called if the $CONNECT operation fails.

success — is the address of a user completion routine to be called if the $CONNECT operation succeeds.

Table 9-8 lists the fields of the Record Access Block used during the $CONNECT operation.

Table 9-8
$CONNECT RAB Fields

	Name	Description
Input	FAB	File access block address.
	KRF	Key of reference. Needed for indexed files only.
	MBC	Multi-block count (sequential disk files only).
	MBF	Multi-buffer count.
	ROP	Record processing options.
	UBF	User record area address (see RBF field below).
Output	ISI	Internal stream identifier.
	RBF	Record buffer address. Supplied only if locate mode for a sequential file was specified in the ROP field. RBF is set equal to UBF.
	STS	Completion status code.
	STV	Status value.

9.3.1.2 **$DISCONNECT - Terminating a Record Access Stream** - The $DISCONNECT macro terminates a record access stream. The association of the specified Record Access Block to a File Access Block is ended. System buffers and internal control structures reserved for record processing by the stream are returned to the buffer pool.

NOTE

The $DISCONNECT macro does not have the effect of an implicit $REWIND operation (refer to Section 9.3.6.6) for magnetic tape files. If you want to ensure that a magnetic tape file will be positioned to the beginning of file for a subsequent $CONNECT to the same open file, you must issue an explicit $REWIND operation before issuing the $DISCONNECT for the current stream.

The formats of the $DISCONNECT macro are:

1. label:$DISCONNECT

2. label:$DISCONNECT rab[,error[,success]]

where

label — is an optional user-defined symbol referring to the $DISCONNECT macro.

rab — is the address of a Record Access Block whose association with a File Access Block is to be ended.

error — is the address of a user completion routine to be called if the $DISCONNECT operation fails.

success — is the address of a user completion routine to be called if the $DISCONNECT operation succeeds.

Table 9-9 lists the fields of the Record Access Block used during the $DISCONNECT operation.

Table 9-9
$DISCONNECT RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
Output	ISI	Internal stream identifier (zeroed).
	STS	Completion status code.
	STV	Status value.

9.3.2 Record Operations and File Sharing

RMS-11 allows your program to share access to a file with other concurrently executing programs. The manner in which a particular file can be shared is based on its organization. Whether a particular file is shared at runtime depends on information provided to RMS-11 by your program and other programs accessing the file. RMS-11 coordinates record operations to a shared file through a bucket locking mechanism. The following subsections, therefore, describe:

1. File organizations and file sharing.
2. Program sharing information.
3. Bucket locking.

9.3.2.1 File Organizations and File Sharing - With the exception of files on unit record devices and magnetic tape, which cannot be shared, every RMS-11 file can be shared by any number of programs that are reading, but not writing, the file. Relative and indexed files (but not sequential files) can be shared by multiple readers and one or more writers. Your program can read or write records in a relative or indexed file while other programs are similarly reading or writing records in the file. Thus, the information in such files can be changing while your program, or other programs, are accessing them.

9.3.2.2 Program Sharing Information - While a file's organization establishes whether it can be shared by readers only or by multiple readers and writers, your program specifies whether such sharing actually occurs at runtime. You control the sharing of a file through information your program provides RMS-11 in the File Access Block used to open the file.

Two fields in the File Access Block provide RMS-11 with file sharing information--the file sharing (SHR) field and the file access (FAC) field. The SHR field specifies what type of sharing you will allow. If the SHR field contains the value FB$WRI, you have indicated that you are willing to share the file with programs that are writing to the file. The absence of FB$WRI (i.e., SHR equals 0) indicates that you are willing to share the file with readers, but not writers. The FAC field declares the record operations that your program can itself perform on the file it is accessing. The presence of one or more of the values FB$PUT, FB$DEL, FB$UPD, or FB$TRN indicates that your program intends to write to the file.

The contents of the SHR and FAC fields in the FAB you use to open a file are critical in the sharing of a file for the following reasons:

1. If your program is the first to open a file, it can perform any record operation on the file as long as the organization of the file supports the operation and the operation is declared in your FAC field. However, the contents of your SHR field establish whether any other program desiring to write the file can also gain access.

2. If your program is not the first to access a file, RMS-11 allows you to open the file only if the contents of your FAC and SHR fields are compatible with the intentions of all programs that currently have access to the file. That is,

   a. If your FAC field indicates write operations and your SHR field indicates no one else can write, then:

      - All current accessors of the file must have specified FB$WRI in their SHR fields, and
      - All current accessors must have indicated read only operations in their FAC fields.

   b. If your FAC field indicates read only operations and your SHR field indicates no one else can write, then:

      - All current accessors of the file must have indicated read only operations in their FAC fields.

   c. If your FAC field indicates read only operations and your SHR field allows writers, then:

      - There cannot be any current accessor who is writing and has disallowed writers.

   d. If your FAC field indicates write operations and your SHR field allows writers, then:

      - There cannot be any current accessor who has disallowed writers.

9.3.2.3 **Bucket Locking** - RMS-11 uses a bucket locking facility to control operations to a relative or indexed file that is being accessed by one or more writers. The purpose of this facility is to ensure that a program can add, delete, or modify a record in a file without another program, or another record access stream within the same program, simultaneously accessing the same record.

RMS-11 employs bucket locking in either of the following instances:

1. The SHR field in the FAB of the first program to open a file specifies shared writing (i.e., FB$WRI).

2. The SHR field in the FAB of the first program to open a file specifies shared reading only (i.e., SHR contains 0) but the FAC field in the same FAB declares one or more output operations (i.e., FB$PUT, FB$UPD, or FB$DEL).

In the first instance, RMS-11 locks any bucket accessed by a successful $GET or $FIND operation issued from within any record access stream in any program processing the file. This lock prevents a second record access stream, in the same program or another program, from accessing any record in the same bucket. In the second instance, RMS-11 locks buckets accessed by $GET or $FIND operation issued by the program whose FAC field indicates write operations. These locks, however, can be encountered only by other record access streams within the same program.

RMS-11 retains the lock on a bucket until the record access stream that caused the locking issues another record operation. While the lock is in effect, RMS-11 returns an ER$RLK (target bucket locked) error status code to other record access streams attempting to access the bucket.

For greatest flexibility at runtime, you should always assume that any record your program attempts to access may be denied because of the target bucket locked (ER$RLK) condition. The bucket that you attempted to access can have been locked by another record access stream in your own program or a record access stream in another program. To deal with this condition, you should employ the following procedures when you write programs:

1. Never allow a lock to be retained on a bucket longer than is necessary. That is, after you issue a successful $GET or $FIND operation, you have caused RMS-11 to lock a bucket. You should issue a second record operation in the same record access stream so that RMS-11 will unlock the bucket. Any record operation (e.g., $PUT, $UPDATE, $DELETE, or another $GET or $FIND) will cause unlocking. Alternatively, you can explicitly unlock the bucket by issuing the $FREE macro (see below).

2. If you are using a single record access stream to access a file and you encounter the ER$RLK error, you can reissue the record operation that failed until RMS-11 indicates successful completion.

3. If you are using multiple record access streams to access a file, you must not merely reissue the record operation that failed. Since one of your own record access streams may have caused locking of the target bucket, you could place your program in an infinite loop if you continue to issue the same operation. Therefore, you should issue a $FREE (see below) for all other record access streams to the same file in your program. You can then safely reissue the original record operation until RMS-11 indicates successful completion.

The $FREE macro unlocks a bucket that RMS-11 has locked on behalf of the record access stream. If no bucket is locked, RMS-11 returns on ER$RNL (no bucket locked) error status code.

The formats of the $FREE macro are as follows:

1. label:$FREE

2. label:$FREE rab[,error[,success]]

where

label — is an optional user-defined symbol referring to the $FREE macro.

rab — is the address of a Record Access Block representing a record access stream.

error — is the address of a user completion routine to be called if the $FREE operation fails.

success — is the address of a user completion routine to be called if the $FREE operation succeeds.

Table 9-10 lists the fields of the Record Access Block used during the $FREE operation.

Table 9-10
$FREE RAB Fields

	Name	Description
Input	ISI	Internal stream identifier
Output	STS	Completion status code
	STV	Status value

9.3.3 Current Context of Record Operations

Transparent to your program, RMS-11 maintains current context information for each record access stream you establish. This context information identifies where, in a file, each record access stream is positioned at any point in time. As your program performs record operations on a record access stream, RMS-11 modifies, as necessary, the current context of that stream.

At any point in time, the current context of a record access stream is represented by, at most, two records:

1. The Current Record
2. The Next Record

The type of record operation you issue establishes whether or not, after the operation, there is a Current Record for the stream. Furthermore, the type of record operation and the access mode you use establish whether the identity of the Next Record is changed or unchanged. Table 9-11 summarizes the effect each successful record operation has on the current context of a stream. The subsections that follow describe the purpose and importance of the notions of Current and Next Records.

Table 9-11
Record Access Stream Context
After Record Operations

Record Operation	Access Mode	Current Record	Identity of Next Record
$CONNECT	-	None	First Record (for indexed files, established by index represented by KRF)
$CONNECT (ROP=RB$EOF)	-	None	End-of-file (sequential files, disk only)

(Continued on next page)

Table 9-11 (Cont.)
Record Access Stream Context
After Record Operations

Record Operation	Access Mode	Current Record	Identity of Next Record
$FIND	Sequential	New	New Current Record +1
$FIND	Random or RFA	New	Unchanged
$GET (not immediately preceded by $FIND)	Sequential	New	New Current Record +1
$GET (immediately preceded by $FIND)	Sequential	Unchanged	Current Record +1
$GET	Random or RFA	New	New Current Record +1
$PUT	Sequential	None	1. Sequential file - EOF. 2. Relative file - next relative record position. 3. Indexed file - undefined.
$PUT	Random	None	Unchanged
$UPDATE	-	None	Unchanged
$DELETE	-	None	Unchanged
$TRUNCATE	-	None	End-of-file
$REWIND	-	None	First record
$FREE	-	None	Unchanged
Unsuccessful record operations (except ER$RTB)	(any)	None	Unchanged

NOTES

1. Except for the $TRUNCATE operation, RMS-11 establishes the Current Record (if any) before establishing the identity of the Next Record.

2. The notation "+1" indicates the next sequential record as determined by the organization of the file. For indexed files, the current key of reference is part of this determination.

9.3.3.1 **Understanding the Current Record** - The Current Record associated with a record access stream represents the target record for an $UPDATE, $DELETE, or $TRUNCATE operation. RMS-11 rejects any $UPDATE, $DELETE, or $TRUNCATE operation you issue for a stream that does not have, at that point, a Current Record.

To establish (or maintain) a Current Record for a stream, you must successfully issue one of the following:

1. A $FIND operation.

2. A $GET operation (including $GET operations that return ER$RTB).

As Table 9-11 shows, either of these two operations, if successful, establishes or retains a Current Record as part of the stream's context. Note that a successful, sequential $GET operation that was immediately preceded by a successful $FIND does not change the Current Record. Your program simply reads the record located by the preceding $FIND but that record continues to be the stream's Current Record. All other types of $GET operations and all $FIND operations establish new Current Records.

In contrast to $FIND and $GET operations, all other record operations, upon completion, leave the stream without a Current Record. Further, any unsuccessful record operation (excluding a $GET that returns ER$RTB) leaves the stream without a Current Record. Therefore, any $UPDATE, $DELETE, or $TRUNCATE operation your program issues must be immediately preceded by a successful $FIND or $GET operation. If you do not follow this procedure, RMS-11 rejects the $UPDATE, $DELETE, or $TRUNCATE operation and returns the ER$CUR (No Current Record) error code.

9.3.3.2 **Understanding the Next Record** - RMS-11 utilizes the Next Record for sequential mode processing. A stream's Next Record represents the target record for:

- A sequential $GET operation (if the immediately preceding operation was not $FIND)

- A sequential $FIND operation

- A sequential $PUT operation to a sequential or relative file

For the preceding operations, RMS-11 uses its internal knowledge of file organization and structure to determine the next record to be processed. This look ahead ability significantly decreases sequential mode access times.

Initially, RMS-11 determines the Next Record as follows:

The $CONNECT operation leaves the Next Record as:

- The first record in a sequential disk file (unless the record processing options (ROP) field in the Record Access Block contains the value RB$EOF).

- End of file in a magnetic tape file that has been opened for $PUT operations.

- The first record in a magnetic tape file if this is the first $CONNECT for a file opened for read only. Note that if there has been a previous $CONNECT followed by a $DISCONNECT for the open tape file, the $REWIND operation should be used to position the stream to beginning of file.

- The first record in a relative file.

- The first record in the collating sequence of the specified key of reference in an indexed file.

Thereafter, RMS-11 alters the identity of Next Record as follows:

- The $GET operation in any access mode, and the $PUT and $FIND operations in sequential mode leave the identity of the Next Record as that of the record following the one on which the operation was performed. This record is:

  1. The next record within sequential or relative files, or

  2. The next record as determined by the collating sequence of the specified key of reference in indexed files.

- The $TRUNCATE operation, which is allowed only on sequential files, sets the Next Record to point to the end of file. RMS-11 does not write EOF indicators in the file as a result of a $TRUNCATE.

The following operations have no effect on the Next Record:

1. $UPDATE.

2. $DELETE.

3. $PUT or $FIND in random access mode.

4. All unsuccessful record operations (except ER$RTB).

9.3.4 Synchronous and Asynchronous Record Operations

Within each record access stream, your program can perform any record operation either synchronously or asynchronously. When a record operation is performed synchronously, RMS-11 returns control to your program only after the record operation request has been satisfied (e.g., a record has been read and passed to your program). When a record operation is performed asynchronously, RMS-11 can return control to your program before the record operation request has been satisfied. Your program, then, can utilize the time required for the physical transfer between the file and memory of the block or bucket containing the record to perform other computations.

To perform asynchronous record operations, you must:

1. Allocate, at assembly-time, an asynchronous Record Access Block to represent the stream in which you will issue asynchronous requests. You must allocate multiple asynchronous Record Access Blocks if you intend to issue asynchronous requests in parallel. (Refer to Section 6.1 in Chapter 6 for the details of allocating an asynchronous RAB.)

2. Ensure that the value RB$ASY is present in the record processing options (ROP) field of the asynchronous RAB before you issue the desired record operation. You can initialize the ROP field at assembly-time with this value or set the field at runtime with the $SET macro. Further, during execution, you can switch between synchronous and asynchronous operations in a stream by using the $SET and $OFF macros to set and reset the value RB$ASY in the ROP field. Note, however, that RMS-11 ignores the value RB$ASY if the RAB is not asynchronous.

When you issue an asynchronous record operation, you can specify a completion routine to be called if the operation succeeds. You can also specify an error completion routine to be called if the operation fails. Within such completion routines, you can issue additional operations. These additional operations, however, should also be asynchronous operations. If you issue a synchronous operation from an asynchronous completion routine, you may receive an ER$AST error if the completion routine was called at AST level in the task.

While an asynchronous operation is in progress in a record access stream, you must:

1. Never modify the contents of the Record Access Block representing the stream.

2. Never issue a second record operation request in the same stream until the first operation is completed. RMS-11 returns an ER$RSA (record stream active) error status code if you issue a record operation to a stream which has a record operation in progress.

To determine when an asynchronous record operation has completed, you issue the $WAIT macro. Upon completion of the asynchronous request for the associated RAB, RMS-11 returns control to your program at the point following the $WAIT macro (after calling the appropriate completion routine, if specified).

The formats of the $WAIT macro are as follows:

1. label:$WAIT

2. label:$WAIT rab

where

label is an optional user-defined symbol referring to the $WAIT macro.

rab is the address of an asynchronous Record Access Block representing a record access stream with an asynchronous request in progress.

Table 9-12 lists the fields of the Record Access Block used during the $WAIT operation.

Table 9-12
$WAIT RAB Fields

	Name	Description
Input	ISI	Internal stream identifier
Output	(varies)	(dependent on operation waited upon)

9.3.5 Accessing Records

To process a record in a file, you must specify an access mode in the Record Access Block representing a stream connected to the file. Your program can use one of two record transfer modes to access the subject record in memory after it has been read from the file or before your program writes it to the file. The following subsections, therefore, describe:

- Specifying an access mode
- Specifying a record transfer mode

9.3.5.1 **Specifying an Access Mode** - You use the record access mode (RAC) field of the Record Access Block to specify the access mode that RMS-11 is to employ for a particular record operation. During the execution of your program, you can switch access modes within a stream by changing the contents of this field.

Within the RAC field, you can specify one of three values:

1. RB$SEQ - sequential access mode.
2. RB$KEY - random access mode.
3. RB$RFA - record's file address access mode.

You can specify sequential access mode with any file organization. Random access mode, however, is restricted to relative and indexed files. RFA access mode is limited to retrieval operations (i.e., $GET, $FIND) to files residing on disk devices.

RMS-11 examines the contents of the RAC field in a RAB only during the execution of a $GET, $FIND, or $PUT operation. The $UPDATE, $DELETE, and $TRUNCATE operations do not require an access mode specification. You cannot issue these operations unless you have already accessed a record by issuing a $GET or $FIND operation.

9.3.5.2 **Specifying a Record Transfer Mode** - While the RAC field specifies how RMS-11 accesses a record in a file on behalf of your program, the record processing options (ROP) field in the RAB allows you to specify how your program (for $GET operations) or RMS-11 (for $PUT or $UPDATE operations) accesses the target record in memory. In the ROP field, you can specify one of two record transfer modes - move mode or locate mode. During execution of your program, you can switch record transfer modes within a stream by changing the contents of this field.

Within the ROP field, you indicate:

- Move mode through the absence of the bit value RB$LOC
- Locate mode through the presence of the bit value RB$LOC

Move mode requires that RMS-11 move individual records between I/O buffers and the user program. For $GET operations, RMS-11 reads a virtual block (sequential files) or bucket (relative or indexed files) into an I/O buffer. RMS-11 then locates the desired record in the buffer and moves it to a program-specified location. For $PUT or $UPDATE operations, your program first builds a record in any desired program location. Your program then stores the address and size of

the record in the RAB and issues the appropriate macro call. During execution of the macro call, RMS-11 moves the record from its specified location to an I/O buffer. When the buffer is filled, RMS-11 writes it to the file.

Locate mode enables your program to access records directly in an I/O buffer. Therefore, there is normally no need for RMS-11 to move records between I/O buffers and the user program. RMS-11 does not permit the use of locate mode if a file was opened for update operations (e.g., FAC contains the value FB$UPD). If a file has not been opened for update operations, then locate mode is supported for $GET operations for all file organizations. For $PUT operations, locate mode is supported for sequentially organized files only.

The record transfer mode you select for a particular record operation determines the use of the following fields of the RAB:

1. Record address (RBF) and record size (RSZ).

2. User record area address (UBF) and user record area size (USZ).

The use of these fields is described in the following subsections.

9.3.5.2.1 **The RBF and RSZ Fields of the RAB** - The RBF (record address) and RSZ (record size) fields of the RAB always describe the location in memory and the size of the target record for a $GET, $PUT, or $UPDATE operation. These fields are set as follows:

1. For $PUT and $UPDATE operations, your program must always ensure that RBF and RSZ describe the location and size of the record to be written.

2. After a successful $GET operation, RMS-11 always sets RBF to the address of the retrieved record and RSZ to the size of the record.

Additionally, when your program is writing records to a sequential file in locate mode, RMS-11 returns an address in the RBF field after each $PUT operation. This address is the location where your program can build the next record. It is not required that this location be used. If it is used, your program need only set the RSZ field to describe the size of the next record to be written. If it is not used, your program must, again, set both RBF and RSZ.

9.3.5.2.2 **The UBF and USZ Fields of the RAB** - The UBF (user record area address) and USZ (user record area size) fields described a fixed work area in your program. Regardless of record transfer mode, you must always provide this work area. Further, it should be large enough to contain the largest record in the file accessed by the stream. RMS-11 uses the work area described by UBF and USZ as follows:

1. For $GET operations in move mode, RMS-11 moves the retrieved record from the I/O buffer to the location described by UBF. RBF (set equal to UBF) and RSZ describe the moved record. If the work area is not large enough to contain the entire record, RMS-11 moves as much of the record as possible (i.e., up to the amount specified in USZ), updates the current context of the stream, and returns the ER$RTB error code.

2. For $GET operations to any file in locate mode, RMS-11 will normally set RBF to the address of the record in an I/O buffer. However, if RMS-11 determines that locate mode cannot be used for a particular operation, RMS-11 will actually use move mode. That is, the record will be moved to the location described by UBF when move mode must be used, RMS-11 may return ER$RTB, as described above, if the user record area is too small.

3. For $UPDATE operations and $PUT operations in move mode, RMS-11 ignores the UBF and USZ fields. You must describe the record to be written in the RBF and RSZ fields.

4. For $PUT operations on sequential files in locate mode, RMS-11 uses the UBF and USZ fields as follows:

   - If the space remaining in the I/O buffer after the $PUT operation has been performed is equal to or greater than USZ, RMS-11 sets RBF to the location within the buffer where your program can build the next record.

   - If the space remaining in the I/O buffer is less than USZ, RMS-11 sets RBF equal to UBF.

9.3.6 Record Operation Macros

Table 9-13 summarizes the record processing macros provided by RMS-11:

Table 9-13
RMS-11 Record Processing Macros

Macro Name	Description
$FIND	Locates a record in a file and returns its RFA.
$GET	Retrieves a record from a file.
$PUT	Writes a new record into a file.
$UPDATE	Rewrites an existing record within a file.
$DELETE	Deletes a record from a relative or indexed file.
$REWIND	Positions to the beginning of a file.
$TRUNCATE	Truncates a sequential file.
$FLUSH	Writes out modified I/O buffers.
$NXTVOL	Causes processing of a magnetic tape file to continue on the next volume of a volume set.

The subsections that follow describe each of the record processing macros listed in the preceding table. Appendix A contains a complete list of completion status codes that can result from the issuance of these and other RMS-11 macros.

9.3.6.1 **$FIND - Locating and Obtaining the RFA of a Record** - The $FIND macro locates a specified record in a file and returns its record's file address into the RFA field of the RAB. Additionally, RMS-11 sets the record pointer and, for $FIND operations in sequential access mode, the next record pointer. The record pointer after a $FIND specifies which record will be the subject of the next sequential $GET operation or a $DELETE, $UPDATE, or $TRUNCATE operation.

The main uses of the $FIND operation are as follows:

1. Skipping records when in sequential access mode (by issuing successive $FIND operations).
2. Establishing a random starting point in a file for sequential access.
3. Establishing a current record for a $DELETE, $UPDATE, or $TRUNCATE operation.

The formats of the $FIND macro are as follows:

1. label:$FIND
2. label:$FIND rab[,error[,success]]

where

label — is an optional user-defined symbol referring to the $FIND macro.

rab — is the address of a Record Access Block containing the specification of a record to be found.

error — is the address of a user completion routine to be called if the $FIND operation fails.

success — is the address of a user completion routine to be called if the $FIND operation succeeds.

Table 9-14 lists the fields of the Record Access Block used during the $FIND operation.

Table 9-14
$FIND RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	KBF	Key buffer address. In combination with KSZ, KBF describes a record identifier for random access to a relative or indexed file.
	KRF	Key of reference. Used only for random access indexed files.
	KSZ	Key size.
	RAC	Record access mode.

(Continued on next page)

Table 9-14 (Cont.)
$FIND RAB Fields

	Name	Description
Input (Cont.)	RFA	Record's file address. Used for RFA access only.
	ROP	Record processing options.
Output	BKT	Bucket code. When you access a relative file in sequential access mode, RMS-11 sets this field equal to the relative record number of the found record.
	RFA	Record's file address.
	STS	Completion status code.
	STV	Status value.

The following subsections describe characteristics of the $FIND operation for the sequential, relative, and indexed file organizations.

9.3.6.1.1 **$FIND and the Sequential File Organization** - Either sequential or RFA access mode can be used with the $FIND operation to a sequential file. RFA access, however, is permitted only for sequential files on disk devices.

9.3.6.1.2 **$FIND and the Relative File Organization** - For the relative file organization, you can employ any of the three supported access modes with the $FIND operation -- sequential, random, or RFA.

When you specify sequential access mode, RMS-11 will locate the next existing record in the file. If no record exists in any remaining record positions, RMS-11 will return the ER$EOF error code (End of File) in the STS field of the RAB.

Normally, you would not use RFA access mode with the $FIND macro since the output of the operation would be the identical record's file address used as input. However, a $FIND operation in RFA access mode could return a ER$DEL (Record Deleted) error code. This error code indicates that the record described by the user-supplied record's file address once existed in the file but was subsequently deleted.

Finally, random access mode can also be used for $FIND operations to relative files. Such an operation can result in an ER$RNF (Record Not Found) error if no record exists in the record position you specified through the KBF and KSZ fields of the RAB.

9.3.6.1.3 **$FIND and the Indexed File Organization** - For the indexed file organization, you can use any of the three supported access modes -- sequential, random, and RFA. In random access mode, RMS-11 uses the key of reference field (KRF) of the RAB to determine the index to be used to locate the record.

In a $FIND operation in sequential access mode, RMS-11 will locate the next record in the sequence established by the index associated with a particular key of reference. RMS-11 uses the key of reference used by the most recent successful $GET or sequential $FIND operation or the key of reference contained in the KRF field at $CONNECT time. If a different index is desired for sequential processing, you should first issue a $REWIND operation specifying the new key of reference or issue a $GET in random access mode with the desired key of reference.

$FIND operations in random access mode require the KRF field. You can specify any valid key of reference. If no record exists in the file that will satisfy the $FIND request, RMS-11 returns an ER$RNF (Record Not Found) error.

During $FIND operations in RFA access mode, RMS-11 ignores the KRF field. The ER$DEL error code may be returned if the desired record once existed in the file but had been deleted subsequently.

9.3.6.2 **$GET - Retrieving a Record** - The $GET macro retrieves a record from any of the RMS-11 file organizations. The RBF and RSZ fields always describe the record retrieved after a successful operation. Further, RMS-11 sets the record pointer to the record's file address of the retrieved record and returns this value in the RFA field of the Record Access Block. These fields are valid only until the next operation on the RAB is issued.

The formats of the $GET macro are as follows:

1. label:$GET

2. label:$GET rab[,error[,success]]

where

label — is an optional user-defined symbol referring to the $GET macro.

rab — is the address of a Record Access Block containing the specification of the record to be accessed.

error — is the address of a user completion routine to be called if the $GET operation fails.

success — is the address of a user completion routine to be called if the $GET operation succeeds.

Table 9-15 lists the fields of the Record Access Block used during the $GET operation.

Table 9-15
$GET RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	KBF	Key buffer. In combination with KSZ, KBF describes a record identifier for random access to a relative or indexed file.

(Continued on next page)

Table 9-15 (Cont.)
$GET RAB Fields

	Name	Description
Input (Cont.)	KRF	Key of reference. Used only for random access to indexed files.
	KSZ	Key size.
	RAC	Record access mode.
	RFA	Record's file address. Used for RFA access only.
	RHB	Record header buffer. Used for VFC format records. If this field equals 0, RMS-11 skips the fixed control area portion of the record.
	ROP	Record processing options.
	UBF	User record area address.
	USZ	User record area size.
Output	BKT	Bucket code. When you access a relative file in sequential access mode, RMS-11 sets this field equal to the relative record number of the record retrieved.
	RBF	Record address. This field contains the address of the retrieved record.
	RFA	Record's file address.
	RSZ	Record size. This field contains the size of the record whose address is in RBF.
	STS	Completion status code.
	STV	Status value. See note below.

NOTE

After a successful $GET operation from a unit record or terminal device, the low order byte of the STV field is used to report the terminating character for the input record. This byte is set as follows:

Octal Contents	Terminating Character
15	CR
33	ESC
32	CTRL/Z
0	other

Except when the low order byte of STV is 0, the terminating character is never in the record described by the RBF and RSZ fields of the RAB.

The following subsections describe the characteristics of the $GET operation for the sequential, relative, and indexed file organizations.

9.3.6.2.1 **$GET and the Sequential File Organization** - Either sequential or RFA access mode can be used with the $GET operation to a sequential file. RFA access, however, is permitted only for sequential files on disk devices.

9.3.6.2.2 **$GET and the Relative File Organization** - For the relative file organization, you can employ any of the three supported access modes -- sequential, random, or RFA.

When you specify sequential access mode, RMS-11 returns the next existing record in the file (or the Current Record if the immediately preceding operation was a successful $FIND). If no record exists in any remaining record positions, RMS-11 returns the ER$EOF error code (End of File) in the STS field of the RAB.

When you use random access mode, RMS-11 returns an ER$RNF (Record Not Found) error if no record exists in the record position you specified through the KBF and KSZ fields of the RAB.

RMS-11 may return an ER$DEL (Record Deleted) error code for $GET operations in RFA access mode. This error code indicates that the desired record once existed in the file but was subsequently deleted.

9.3.6.2.3 **$GET and the Indexed File Organization** - For the indexed file organization, you can employ any of the three supported access modes -- sequential, random, and RFA.

In a $GET operation (not preceded by a $FIND) in sequential access mode, RMS-11 retrieves the next record in the sequence established by the index associated with a particular key of reference. RMS-11 uses the key of reference of the most recent successful $GET or $FIND operation or the key of reference contained in the KRF field at $CONNECT time. If a different index is desired for sequential processing, you should first issue a $REWIND operation specifying the new key of reference or issue a $GET or $FIND in random access mode with the desired key of reference.

$GET operations in random access mode require the KRF field. You can specify any desired key of reference. If no record exists in the file that will satisfy the $GET request, RMS-11 returns an ER$RNF (Record Not Found) error.

During $GET operations in RFA access mode, RMS-11 ignores the KRF field. The ER$DEL error code may be returned if the desired record has been deleted.

9.3.6.3 **$PUT - Writing a New Record to a File** - The $PUT macro writes a new record into any RMS-11 file organization. The RBF and RSZ fields must describe the record to be written. RFA access mode cannot be used. Note that $PUT operations in random access mode do not change the next record pointer.

The following are the formats of the $PUT macro:

1. label:$PUT

2. label:$PUT rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $PUT macro.

rab is the address of a Record Access Block containing the specification of the record to be written.

error is the address of a user completion routine to be called if the $PUT operation fails.

success is the address of a user completion routine to be called if the $PUT operation succeeds.

Table 9-16 lists the fields of the Record Access Block used during the $PUT operation.

Table 9-16
$PUT RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	KBF	Key buffer. Used only for random access to relative files.
	KSZ	Key size. Used for random access to relative files only.
	RAC	Record access mode.
	RBF	Record address.
	RHB	Record header buffer. Used for VFC records only. If this field equals 0, RMS-11 zeroes the fixed control area portion of the record written.
	ROP	Record processing options.
	RSZ	Record size.
	UBF	User record area. Used for locate mode for sequential files only (refer to Section 9.3.5.2.2).
	USZ	User record area size. Used for locate mode for sequential files only.
Output	BKT	Bucket code. When you access a relative file in sequential access mode, RMS-11 sets this field equal to the relative record number of the record written.

(Continued on next page)

Table 9-16 (Cont.)
$PUT RAB Fields

	Name	Description
Output (Cont.)	RBF	Record address. This field indicates where the next record may be built. (locate mode on sequential files only)
	RFA	Record's file address.
	STS	Completion status code.
	STV	Status value.

The following subsections describe the characteristics of the $PUT operation for the sequential, relative, and indexed file organizations.

9.3.6.3.1 **$PUT and the Sequential File Organization** - You must ensure that the value RB$SEQ is present in the record access field (RAC) of the RAB at the time a $PUT macro call is issued for a sequential file. Further, RMS-11 does not permit new records to be added to a file at any place but the end of file. Therefore, if you intend to add records to a sequential file that already contains records, either the value RB$EOF (position to end-of-file) must be present in the record options field (ROP) of the RAB at the time you issue the $CONNECT macro call for the stream or you must position to end-of-file through $FINDs and/or $GETs prior to issuing any $PUT requests.

You cannot write any record whose length is greater than the maximum record size specified when the file was created.

9.3.6.3.2 **$PUT and the Relative File Organization** - You can use either sequential or random access mode during $PUT operations to a relative file. For both access modes, three restrictions apply. First, no record can be written to the file whose length is greater than the maximum record size specified when the file was created. Second, the target record position cannot already contain a record. Third, the target record position cannot represent a relative offset from the beginning of the file whose value is greater than the maximum record number, if such a number was specified at the time the file was created.

9.3.6.3.3 **$PUT and the Indexed File Organization** - You can use either sequential or random access mode during $PUT operations to an indexed file. For both access modes, two restrictions apply to the length of the record to be written. First, the length of the record cannot exceed the maximum record size specified when the file was created. Second, every record written must be large enough to contain a complete primary key field. It is not required, however, that records be large enough to contain all defined alternate key fields. If such alternate key fields are partially or completely missing because of record length, RMS-11 makes no entries for the new record in the associated alternate indexes.

$PUT operations to an indexed file do not require a key value or a key of reference. RMS-11 determines where to write the record by examining the contents of the primary key field in the record. Before writing the record, RMS-11 compares the key values (primary and alternate) in the record with the key values of records already existing in the file. This comparison determines if the writing of the record would result in the presence of duplicate key values among records of the file. If duplicates would occur, RMS-11 verifies the defined characteristics for the key field(s) being duplicated. If duplicates are not allowed for a particular key field, RMS-11 rejects the operation with an ER$DUP error code. However, if duplicates are allowed, RMS-11 performs the $PUT operation and returns a success code of SU$DUP. Subsequent sequential $GET operations on a given index will always retrieve records with identical key values in the order in which the records were written.

When you issue a series of $PUT operations in sequential access mode, RMS-11 checks that the primary key in each record is equal to (if duplicate primary key values are allowed) or greater than the primary key of the previous record written. An ER$SEQ (key out of sequence) error status code is returned if the primary key of the current record fails this check. This checking is suppressed, however, if the immediately preceding operation was not a successful $PUT in sequential access mode.

9.3.6.4 **$UPDATE - Rewriting an Existing Record** - The $UPDATE macro rewrites an existing record within a file. The Current Record associated with the stream will be rewritten. Therefore, the operation immediately preceding a $UPDATE must be a successful $GET or $FIND. Otherwise, RMS-11 returns an ER$CUR (No Current Record) error code. The address and size of the replacement record must be in the RBF and RSZ fields respectively. Errors indicating an illegal input value in the RAB (e.g., ER$RSZ - illegal record size) do not effect the original record in the file. Other types of errors (e.g., ER$WER - file write error), however, can mean that the original record is lost.

The following are the formats of the $UPDATE macro:

1. label:$UPDATE

2. label:$UPDATE rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $UPDATE macro.

rab is the address of a Record Access Block.

error is the address of a user completion routine to be called if the $UPDATE operation fails.

success is the address of a user completion routine to be called if the $UPDATE operation succeeds.

Table 9-17 lists the fields of the Record Access Block used during the $UPDATE operation.

Table 9-17
$UPDATE RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	RBF	Record address.
	RHB	Record header buffer. Used for VFC format records only. If no address is specified in this field, RMS-11 leaves the fixed control area portion of the original record unaltered.
	ROP	Record processing options. Note that locate mode is not supported for $UPDATE operations.
	RSZ	Record size.
Output	RFA	Record's file address.
	STS	Completion status code.
	STV	Status value.

The following subsections describe characteristics of the $UPDATE operation for the sequential, relative, and indexed file organizations.

9.3.6.4.1 **$UPDATE and the Sequential File Organization** - RMS-11 does not permit the $UPDATE operation for sequential files residing on magnetic tape or unit record devices. Furthermore, for disk files, the format of the records in the file cannot be stream and you cannot change the length of the record being rewritten.

9.3.6.4.2 **$UPDATE and the Relative File Organization** - If the format of records is variable, the length of the replacement record can differ from the length of the original record. In all instances, however, you cannot issue a $UPDATE macro call specifying a replacement record whose length is greater than the maximum record size defined when the file was created.

9.3.6.4.3 **$UPDATE and the Indexed File Organization** - On an $UPDATE operation to an indexed file that allows duplicate primary keys, you cannot change the length of the original record (RMS-11 returns ER$RSZ is you attempt such a change). For indexed files that do not allow duplicate primary keys, however, the length of the replacement record to be written by the $UPDATE macro call can differ from the length of the original record. Two restrictions, however, apply to the length of the replacement record. First, the length of the replacement record cannot exceed the maximum record size defined when the file was

created. Second, every replacement record must be large enough to contain a complete primary key field. It is not required, however, that the replacement record be large enough to contain all defined alternate key fields. If an alternate key field is partially or completely missing in the replacement record but was present in the original record, the key field must have the characteristic that key values can change. This is also true if the replacement record contains a key field missing in the original record.

Before writing the record, RMS-11 compares the key values of the original record with the key values of the replacement record. This comparison takes into account the defined characteristics of each key. For example, if a particular key is not allowed to change, RMS-11 will reject the $UPDATE operation with a ER$CHG error code if the replacement record contains an altered value in the associated key field. Similarly, if duplicates are not allowed for a particular key, RMS-11 rejects the operation with a ER$DUP error code if writing the replacement record would cause duplicate values for the particular key. Conversely, if duplicates are allowed and writing the record results in the presence in the file of duplicate values for a particular key, RMS-11 performs the write operation and returns a success code of SU$DUP.

9.3.6.5 **$DELETE - Deleting a Record** - The $DELETE macro deletes an existing record from a relative or indexed file. This macro is an illegal operation for records in a sequential file.

The $DELETE operation always applies to the Current Record. Therefore, the operation immediately preceding a $DELETE must be a successful $GET or $FIND. Otherwise, RMS-11 returns an ER$CUR (No Current Record) error code.

The formats of the $DELETE macro are as follows:

1. label:$DELETE

2. label:$DELETE rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $DELETE macro.

rab is the address of a Record Access Block.

error is the address of a user completion routine to be called if the $DELETE operation fails.

success is the address of a user completion routine to be called if the $DELETE operation succeeds.

Table 9-18 lists the fields of the Record Access Block used during the $DELETE operation.

Table 9-18
$DELETE RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	ROP	Record processing options.
Output	STS	Completion status code.
	STV	Status value.

9.3.6.6 **$REWIND - Positioning to the Beginning of a File** - The $REWIND macro sets the current context of a stream to the beginning of a file. Following the operation, there is no Current Record. The Next Record is the first record in the file (for indexed files, the KRF field establishes the index to be used).

The following are the formats of the $REWIND macro:

1. label:$REWIND

2. label:$REWIND rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $REWIND macro.

rab is the address of a Record Access Block associated with a file.

error is the address of a user completion routine to be called if the $REWIND operation fails.

success is the address of a user completion routine to be called if the $REWIND operation succeeds.

Table 9-19 lists the fields of the Record Access Block used during the $REWIND operation.

Table 9-19
$REWIND RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	KRF	Key of reference. Used for indexed files only.
Output	STS	Completion status code.
	STV	Status value.

9.3.6.7 **$TRUNCATE - Truncating a Sequential File** - The $TRUNCATE macro truncates a sequential file. This operation is illegal for relative and indexed files.

$TRUNCATE deletes the Current Record and all records following that record. The immediately preceding operation must be a successful $GET or $FIND. Otherwise, RMS-11 returns an ER$CUR (No Current Record) error code.

RMS-11 declares an end-of-file at the position formerly occupied by the Current Record. Additionally, the $TRUNCATE operation causes the Next Record to point to this end-of-file. Therefore, you can extend the file by issuing $PUT operations in sequential mode following the $TRUNCATE operation.

The following are the formats of the $TRUNCATE macro:

1. label:$TRUNCATE

2. label:$TRUNCATE rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $TRUNCATE macro.

rab is the address of a Record Access Block associated with a sequential file.

error is the address of a user completion routine to be called if the $TRUNCATE operation fails.

success is the address of a user completion routine to be called if the $TRUNCATE operation succeeds.

Table 9-20 lists the fields of the Record Access Block used during the $TRUNCATE operation.

Table 9-20
$TRUNCATE RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
Output	STS	Completion status code.
	STV	Status value.

9.3.6.8 **$FLUSH - Writing Out Modified I/O Buffers** - The $FLUSH macro writes out all modified I/O buffers associated with a record access stream, thus ensuring that all record activity up to a point in time is actually reflected in the file. If the file is relative or indexed, any bucket currently locked by the stream remains locked.

The formats of the $FLUSH macro are as follows:

1. label:$FLUSH

2. label:$FLUSH rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $FLUSH macro.

rab is the address of a Record Access Block representing a stream to be flushed.

error is the address of a user completion routine to be called if the $FLUSH operation fails.

success is the address of a user completion routine to be called if the $FLUSH operation succeeds.

Table 9-21 lists the fields of the Record Access Block used during the $FLUSH operation.

Table 9-21
$FLUSH RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
Output	STS	Completion status code.
	STV	Status value.

9.3.6.9 **$NXTVOL - Continue Processing on Next Volume** - The $NXTVOL macro can be used only when the stream is accessing a file on magnetic tape. You issue this macro when you want to continue processing the file on the next volume of a volume set before the end of the current volume is reached. RMS-11 will then open the first file section on the next volume. File sections occur when a file is written on more than one volume. The portion of the file on each of the volumes constitutes a file section. For input files, the following processing occurs when you issue a $NXTVOL macro:

1. All records in I/O buffers for the current file section are skipped.

2. If the current volume is the last volume in the set, i.e., there is no next volume, RMS-11 reports end-of-file to your program.

3. If another file section exists, the current volume is rewound and the next volume is mounted. A request to the operator is printed if necessary.

4. The header label (HDR1) of the first file section is read and checked.

5. If all required fields check, the operation continues.

6. If any check fails, the operator is requested to mount the correct volume.

For output files, the following processing occurs.

1. I/O buffers that are currently in memory are written on the current file section (i.e., an implicit $FLUSH is performed).

2. The current file section is closed with EOV1 and EOV2 labels and the volume is rewound.

3. The next volume is mounted.

4. A file with the same name and the next higher section number is opened for write. The file set identifier is identical with the volume identifier of the first volume in the volume set.

The formats of the $NXTVOL macro are as follows:

1. label:$NXTVOL

2. label:$NXTVOL rab[,error[,success]]

where

label is an optional user-defined symbol referring to the $NXTVOL macro.

rab is the address of a Record Access Block associated with a file on a magnetic tape volume set.

error is the address of a user completion routine to be called if the $NXTVOL operation fails.

success is the address of a user completion routine to be called if the $NXTVOL operation succeeds.

Table 9-22 lists the fields of the Record Access Block used during the $NXTVOL operation.

Table 9-22
$NXTVOL RAB Fields

	Name	Description
Input	ISI	Internal stream identifier.
	ROP	Record processing options.
Output	STS	Completion status code.
	STV	Status value.

APPENDIX A

COMPLETION STATUS CODES

This appendix describes completion status codes that can be returned by RMS-11 to your program.

All RMS-11 file and record operations return a completion status code into the status field (STS) of the control block (i.e., FAB or RAB) associated with the operation. A symbolic name is defined for each such code. The symbolic names for successful completion status codes take the following form:

```
SU$xxx
```

where

xxx	is a mnemonic value describing the successful operation.

Symbolic names for error completion status codes take the form:

```
ER$xxx
```

where

xxx	is a mnemonic value representing the reason the operation failed.

For certain error conditions, RMS-11 uses the status value (STV) field to communicate additional information to your program. The tables in this appendix list all instances in which a particular symbolic value in the STS field indicates the presence of further information in the STV field.

NOTE

When the tables in this appendix indicate that the STV field contains an ACP error code, you should refer to the description of such codes in Appendix I of the IAS/RSX-11 I/O Operations Reference Manual.

A limited number of severe error conditions cause RMS-11 to invoke a fatal error crash routine. Section A.3 of this appendix describes these conditions and the crash routine itself.

The sections that follow describe, respectively, successful completion status codes, error completion status codes, and the RMS-11 fatal error crash routine.

A.1 SUCCESSFUL COMPLETION STATUS CODES

Table A-1 describes successful completion status codes returned by RMS-11 routines.

Table A-1
Successful Completion Status Codes

Symbolic Name	Decimal Value	Description
SU$SUC	1	Operation successful.
SU$DUP	2	A record written into an indexed file as a result of a $PUT or $UPDATE operation contains at least one key value that was already present in another record.
SU$IDX	3	During a $PUT or $UPDATE operation on an indexed file, the record was successfully written. The record can be subsequently retrieved but RMS-11 was not able to optimize the structure of the index at the time the record was inserted. Several indirections will occur, therefore, on retrieval. In some instances, RMS-11 may also return an error code (e.g., ER$RLK) in the STV field of the control block.
SU$RRV	4	During a $PUT or $UPDATE operation on an indexed file, the record was successfully written. However, RMS-11 was unable to update one or more Record Retrieval Vectors (RRVs) and the records associated with the RRVs cannot be retrieved using alternate indexes or RFA addressing mode.

A.2 ERROR COMPLETION STATUS CODES

Table A-2 describes error completion status codes returned by RMS-11 routines.

Table A-2
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$ABO	177760	-16	ER$STK or ER$MAP	Operation aborted: out of stack save area or in core data structures corrupted.
ER$ACC	177740	-32	ACP error code	Files-11 ACP could not access the file.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$ACT	177720	-48		File activity precludes action (e.g., attempting to close a file with outstanding asynchronous record operation).
ER$AID	177700	-64	XAB address	Bad area identification number (AID) field in allocation XAB (i.e., out of sequence).
ER$ALN	177660	-80	XAB address	Illegal value in alignment boundary type (ALN) field of allocation XAB.
ER$ALQ	177640	-96	(XAB address)	Value in allocation quantity (ALQ) field in FAB (or allocation XAB) exceeds maximum or, during an explicit $EXTEND operation, equals zero.
ER$ANI	177620	-112		Records in a file on ANSI labeled magnetic tape are variable length but not in ANSI D format.
ER$AOP	177600	-128	XAB address	Illegal value in allocation options (AOP) field in allocation XAB.
ER$AST	177560	-144		Invalid operation at AST level: attempting to issue a synchronous operation from an asynchronous record operation completion routine.
ER$ATR	177540	-160	ACP error code	Read error on file header attributes.
ER$ATW	177520	-176	ACP error code	Write error on file header attributes.
ER$BKS	177500	-192		Bucket size (BKS) field in FAB contains value exceeding maximum.
ER$BKZ	177460	-208	XAB address	Bucket size (BKZ) field in allocation XAB contains value exceeding maximum.
ER$BLN	177440	-224		Block length (BLN) field in a FAB or RAB is incorrect.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$BOF	177430	-232		Beginning of file detected on $SPACE operation to magnetic tape file.
ER$BPA	177420	-240		Private buffer pool address not a double word boundary.
ER$BPS	177400	-256		Private buffer pool size not a multiple of 4.
ER$BUG	177360	-272		Internal error detected in RMS-11 (refer to Section A.3 of this Appendix); no recovery possible; contact a Software Specialist.
ER$CCR	177340	-288		Can't connect RAB (i.e., only one record access stream permitted for sequential files).
ER$CHG	177320	-304		$UPDATE attempting to change a key field that does not have the change attribute.
ER$CHK	177300	-320		Index file bucket check-byte mismatch. The bucket has been corrupted. No recovery possible for the bucket.
ER$CLS	177260	-336	RSTS/E error code	Close function failed (RSTS/E operating system only).
ER$COD	177240	-352	XAB address	Invalid COD field in XAB or XAB type is illegal for the organization or operation.
ER$CRE	177220	-368	ACP error code	Files-11 ACP could not create file.
ER$CUR	177200	-384		No current record: operation not immediately preceded by a successful $GET or $FIND.
ER$DAC	177160	-400	ACP error code	Files-11 ACP deaccess error during $CLOSE
ER$DAN	177140	-416	XAB address	Invalid area number in DAN field of key definition XAB.
ER$DEL	177120	-432		Record accessed by RFA access mode has been deleted.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$DEV	177100	-448		1. Syntax error in device name. 2. No such device. 3. Inappropriate device for operation (e.g., attempting to create an indexed file on magnetic tape).
ER$DIR	177060	-464		Syntax error in directory name.
ER$DME	177040	-480		Dynamic memory exhausted: insufficient space in central space pool or private buffer pool.
ER$DNF	177020	-496		Directory not found.
ER$DNR	177000	-512		Device not ready.
ER$DPE	176770	-520	ACP error code	Device positioning error.
ER$DUP	176740	-544		Duplicate key detected, duplicates allowed attribute not set for one or more key fields.
ER$ENT	176720	-560	ACP error code	Files-11 ACP enter function failed.
ER$ENV	176700	-576		Environment error: operation or file organization not specified in ORG$ macro.
ER$EOF	176660	-592		End of file.
ER$ESS	176640	-608		Expanded string area in NAM block too short.
ER$EXP	176630	-616		File expiration date not reached.
ER$EXT	176620	-624	ACP error code	File extend failure.
ER$FAB	176600	-640		Not a valid FAB: BID field does not contain FB$BID. Refer to Section A.3 of this Appendix.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$FAC	176560	-656		1. Record operation attempted was not declared in FAC field of FAB at open time. 2. Invalid contents in FAC field. 3. FB$PUT not present in FAC for $CREATE operation.
ER$FEX	176540	-672		File already exists (attempted $CREATE operation).
ER$FID	177530	-680		Invalid file id.
ER$FLG	176520	-688	XAB address	Invalid combination of values in FLG field of key definition XAB (e.g., no duplicates and keys can change).
ER$FLK	176500	-704		File locked by another user -- you cannot access the file because your sharing specification cannot be met.
ER$FND	176460	-720	ACP error code	Files-11 ACP Find function failed.
ER$FNF	176440	-736		File not found.
ER$FNM	176420	-752		Syntax error in file name.
ER$FOP	176400	-768		Invalid file options.
ER$FUL	176360	-784		Device full: can't create or extend file.
ER$IAN	176340	-800	XAB address	Invalid area number in IAN field of key definition XAB.
ER$IDX	176320	-816		Index not initialized (this code can only occur in the STV field when STS contains ER$RNF).
ER$IFI	176300	-832		Invalid IFI field in FAB.
ER$IMX	176260	-848	XAB address	Maximum number (254) of key definition or allocation XABs exceeded or multiple summary, protection, or date XABs present during operation.
ER$INI	176240	-864		$INIT or $INITIF macro call never issued.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$IOP	176220	-880		Illegal operation; examples include: 1. Attempting a $TRUNCATE operation to a non-sequential file. 2. Attempting an $ERASE or $EXTEND operation to a magnetic tape file. 3. Issuing a block mode operation (e.g., $READ or $WRITE) to a stream not connected for block operations. 4. Issuing a record operation (e.g., $GET, $PUT) to a stream connected for block mode operations.
ER$IRC	176200	-896		Illegal record encountered in sequential file: invalid count field.
ER$ISI	176160	-912		Invalid internal stream identifier (ISI) field in RAB (field may have been altered by user) or $CONNECT never issued for stream.
ER$KBF	176140	-928		Key buffer address (KBF) field equals 0.
ER$KEY	176120	-944		Record identifier (i.e., the 4-byte location addressed by KBF) for random operation to relative file is 0 or negative.
ER$KRF	176100	-960		Invalid key of reference (KRF) in RAB: 1) As input to random $GET or $FIND operation, or 2) As input to $CONNECT or $REWIND (in this case, ER$KRF is returned for the first record operation following the $CONNECT or $REWIND.
ER$KSZ	176060	-976		Key size equals zero or too large (indexed file) or not equal to 4 (relative file).
ER$LAN	176040	-992	XAB address	Invalid area number in LAN field of key definition XAB.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$LBL	176020	-1008		Magnetic tape is not ANSI labeled.
ER$LBY	176000	-1024		Logical channel busy.
ER$LCH	175760	-1040		Invalid value in logical channel number (LCH) field of FAB.
ER$LEX	175750	-1048	XAB address	Attempt to extend an area containing an unused extent.
ER$LOC	175740	-1056	XAB address	Invalid value in LOC field of allocation XAB.
ER$MAP	175720	-1072		In core data structures (e.g., I/O buffers) corrupted. This code can only occur in the STV field when STS contains ER$ABO Refer also to Section A.3 of this Appendix.
ER$MKD	175700	-1088	ACP error	Files-11 ACP could not mark code file for deletion.
ER$MRN	175660	-1104		1. Maximum record number field contains a negative value during $CREATE of relative file. 2. Record identifier (pointed to by KBF) for random operation to relative file exceeds maximum record number specified when file created.
ER$MRS	175640	-1120		Maximum record size (MRS) field contains 0 during $CREATE operation and: 1. Record Format is fixed, or 2. File organization is relative.
ER$NAM	175620	-1136		Odd address in Name Block address (NAM) field in FAB on $OPEN, $CREATE, or $ERASE.
ER$NEF	175600	-1152		Not at end-of-file: attempting a $PUT operation to a sequential file when stream is not positioned to EOF.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$NID	175560	-1168		Can't allocate internal index descriptor: insufficient room in space pool while attempting to open an indexed file.
ER$NPK	175540	-1184		No primary key definition XAB present during $CREATE of indexed file.
ER$OPN	175520	-1200	RSTS/E error code	Open function failed (RSTS/E operating system only).
ER$ORD	175500	-1216	XAB address	XABs in chain not in correct order: 1. Allocation or key definition XABs not in ascending (or densely ascending) order. 2. XAB of another type intervenes in key definition or allocation XAB sub-chain.
ER$ORG	175460	-1232		Invalid value in file organization (ORG) field of FAB.
ER$PLG	175440	-1248		Error in file's prologue: file is corrupted and must be reconstructed.
ER$POS	175420	-1264	XAB address	Key position (POS) field in key definition XAB contains a value exceeding maximum record size.
ER$PRM	175400	-1280	XAB address	File header contains bad date and time information (retrieved by RMS-11 because a date and time XAB is present during a $OPEN or $DISPLAY operation); file may be corrupted.
ER$PRV	175360	-1296		Privilege violation: access to the file denied by the operating system.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$RAB	175340	-1312		Not a valid RAB: BID field does not contain RB$BID. Refer to Section A.3 of this Appendix.
ER$RAC	175320	-1328		1. Illegal values in record access mode (RAC) field of RAB. 2. Illogical value in RAC field (e.g., RB$KEY with a sequential file).
ER$RAT	175300	-1344		1. Illegal values in record attributes (RAT) field of FAB during $CREATE. 2. Illogical combination of attributes (e.g., FB$CR and FB$FTN) in RAC field during $CREATE.
ER$RBF	175260	-1360		Record address (RBF) field in RAB contains an odd address (block mode access only).
ER$RER	175240	-1376	ACP error code	File read error.
ER$REX	175220	-1392		Record already exists: during a $PUT operation in random mode to a relative file, an existing record found in the target record position.
ER$RFA	175200	-1408		Invalid RFA in RFA field of RAB during RFA access.
ER$RFM	175160	-1424		1. Invalid record format in RFM field of FAB during $CREATE. 2. Specified record format is illegal for file organization.
ER$RLK	175140	-1440		Target bucket locked by another task or another stream in the same program.
ER$RMV	175120	-1456	ACP error code	Files-11 ACP Remove function failed.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$RNF	175100	-1472	(ER$IDX)	Record identified by KBF/KSZ fields of RAB for random $GET or $FIND operation does not exist in relative or indexed file (for indexed files only, STV may contain ER$IDX). Record may never have been written or may have been deleted.
ER$RNL	175060	-1488		$FREE operation issued but no bucket was locked by stream.
ER$ROP	175040	-1504		Record options (ROP) field contains illegal values or illogical combination of values.
ER$RPL	175020	-1520	ACP error code	Error while reading prologue.
ER$RRV	175000	-1536		Invalid RRV record encountered in indexed file; file may be corrupted.
ER$RSA	174760	-1552		Record stream active, i.e., in asynchronous environment, attempting to issue a record operation to a stream that has a request outstanding.
ER$RSZ	174740	-1568		Record size specified in RSZ of RAB during $PUT or $UPDATE is invalid: 1. RSZ equals zero. 2. RSZ exceeds maximum record size (MRS) specified when file created. 3. RSZ not equal to size of Current Record for $UPDATE operation to a sequential file on disk. 4. RSZ does not equal MRS (for fixed format records).
ER$RTB	174720	-1584	Actual record size	Record too big for user's buffer: RMS-11 could not move entire record retrieved by $GET operation to user work area (UBF/USZ). Note that this error does not destroy the current context of the stream. Rather, the stream's context is updated as if the operation had been completely successful.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$SEQ	174700	-1600		During $PUT operation, key of record to be written is not equal to or greater than key of previous record (and RAC field contains RB$SEQ).
ER$SHR	174660	-1616		Illogical value in SHR field of FAB (e.g., FB$WRI specified for sequential file).
ER$SIZ	174640	-1632	XAB address	Invalid SIZ field in key definition XAB during $CREATE (e.g., specified size exceeds maximum record size).
ER$STK	174620	-1648		During asynchronous record operation, RMS-11 has found that the stack is too big to be saved (this code can only occur in the STV field when STS contains ER$ABO).
ER$SYS	174600	-1664	Directive or QIO status code	System directive error.
ER$TRE	174560	-1680		Index tree error: indexed file is corrupted.
ER$TYP	174540	-1696		Syntax error in file type (e.g., more than 3 characters specified).
ER$UBF	174520	-1712		Invalid address in UBF field of RAB: 1. UBF contains 0, or 2. UBF not word aligned (for block mode access only).
ER$USZ	174500	-1728		Invalid USZ field in RAB (i.e., USZ contains 0).
ER$VER	174460	-1744		Syntax error in file version number.
ER$VOL	174440	-1760	XAB address	Invalid VOL field in allocation XAB (i.e., VOL does not contain 0).
ER$WER	174420	-1776	ACP error code	File write error.

(Continued on next page)

Table A-2 (Cont.)
Error Completion Status Codes

Symbolic Value	Octal Value	Decimal Value	STV	Description
ER$WLK	174410	-1784		Device is write locked.
ER$WPL	174400	-1792	ACP error code	Error while writing prologue.
ER$XAB	174360	-1808	(XAB address)	XAB field in FAB (or NXT field in XAB) contains an odd address.

A.3 FATAL ERROR CRASH ROUTINE

RMS-11 issues a BPT instruction whenever it encounters inconsistent internal f FAB or RAB). This action is taken only when RMS-11 cannot continue processing, since to do so might cause damage to user files or the user's task image. As an example, when the problem is caused by an invalid FAB or RAB, RMS-11 cannot return an error status code in STS since it has no recognizable user control block to work with.

The BPT instruction generated as a result of fatal errors is in the R0RMSA module of RMS-11. The following is the state of the general registers at the time this instruction is issued:

R0 = RMS-11 error code
R1 = Entry SP value
R2 = Entry return PC
R3 = Address of system impure area

General registers R1 and R2 are always valid if the crash routine is invoked by a fatal user call error. When the crash routine is invoked by inconsistent internal conditions, the contents of general registers R1 and R2 may be meaningless if RMS-11 was executing an asynchronous RAB operation.

The following subsections summarize, respectively, the fatal user call errors and the RMS-11 inconsistent internal conditions that can cause invocation of the fatal error crash routine.

A.3.1 Fatal User Call Errors

When the fatal error crash routine is invoked because of a user call error, general register R0 contains one of the following error codes:

- ER$FAB
- ER$RAB

These error codes indicate that the user called RMS-11 using a control block that was not a valid FAB (for file operations such as $OPEN, $CREATE, etc) or RAB (for record operations such as $CONNECT, $GET, $PUT, etc.). This condition can occur for any one of the following reasons:

1. The address of the FAB or RAB is 0.

2. The address of the FAB or RAB is odd.

3. The control block's BID field does not contain the proper block identifier code (i.e., FB$BID for FABs and RB$BID for RABs).

A.3.2 RMS-11 Inconsistent Internal Conditions Errors

When the crash routine is invoked because of RMS-11 inconsistent internal conditions, general register R0 contains one of the following error codes:

- ER$BUG
- ER$MAP

These error codes indicate internal problems with RMS-11 and are considered fatal. They can be caused by improper coding by the user (e.g., destroying some internal RMS-11 data base), but are also used to detect RMS-11 bugs. When one of the above error codes is encountered, the user should provide, if possible, the following information to DEC with an SPR:

1. The contents of the general registers.

2. The first ten words, at a minimum, or all words upon the system stack.

3. The operation the program was performing (e.g., $OPEN, $GET, $PUT).

4. The organization of the file being processed.

5. A load map of the task.

6. If running on RSX-11M, a post-mortem dump.

APPENDIX B

PERFORMING BLOCK I/O

In addition to sequential, random, and RFA access, RMS-11 provides a fourth access mode known as block access. Block access allows you to bypass entirely the record processing capabilities of RMS-11. Through macros described in this appendix, you can directly read or write the virtual blocks of a file.

NOTE

Many elements of the internal structure of RMS-11 files are not normally visible to user programs. Through the use of block access, however, you can gain visibility of these elements. Extreme caution must be exercised, therefore, when altering the contents of the virtual blocks of an RMS-11-structured sequential, relative, or indexed file.

B.1 SPECIFYING BLOCK ACCESS

To use block access, you must allocate 1 buffer descriptor block (BDB) for each stream connected simultaneously for block access (RMS-11 permits a single block access stream for sequential files and multiple streams for relative and indexed files). These BDBs are specified in the P$BDB macro in the space pool declaration section (POOL$B .... POOL$E). At $OPEN or $CREATE time, the value FB$REA must be present in the FAC field of the FAB if the file will be read in block access mode and the value FB$WRT must be present in the FAC field if the file will be written in block access mode. If you are creating a file in block mode, the ORG field of the FAB must contain FB$SEQ and the RFM field must contain FB$UDF. Finally, a 512 byte buffer (either in the space pool or in a user-specified location described by the BPA and BPS fields of the FAB) must be present for the $OPEN or $CREATE macro call to execute successfully.

Once a file has been opened for block access, RMS-11 will not allow you to connect any streams to the file for record access. Only block access can be performed on the file. Each block access stream is activated by a $CONNECT macro call and terminated by a $DISCONNECT macro call.

B.2 $READ - RETRIEVING VIRTUAL BLOCKS

The $READ macro retrieves a user-specified number of bytes from a file beginning on a specified virtual block boundary. You must supply a word-aligned work area (UBF) into which RMS-11 is to move the blocks of the file. You indicate the even number of bytes to be transferred to the UBF location in the USZ field and the starting virtual block number in the file in the BKT field. After completion of the transfer, RMS-11 uses the RBF and RSZ fields to describe the location and number of bytes actually transferred.

The formats of the $READ macro are as follows:

1. label:$READ

2. label:$READ rab[,error[,success]]

where

label — is an optional user-defined symbol referring to the $READ macro.

rab — is the address of a Record Access Block containing the specification of the block(s) to be retrieved.

error — is the address of a user completion routine to be called if the $READ operation fails.

success — is the address of a user completion routine to be called if the $READ operation succeeds.

Table B-1 lists the fields of the Record Access Block used during the $READ operation.

Table B-1
$READ RAB Fields

	Name	Description
Input	BKT	Bucket number. Must contain virtual block number of first block to be read.
	ISI	Internal stream identifier.
	UBF	User work area address (word-aligned).
	USZ	User work area size (must be an even number). This field will control the total amount of data transferred. Multiple virtual blocks can be retrieved from a disk file by specifying the appropriate multiple of 512 bytes in this field.
Output	RBF	Record address. This field is set equal to UBF.
	RSZ	Record size. Actual number of bytes transferred (not including terminator character - refer to STV below).

(Continued on next page)

Table B-1 (Cont.)
$READ RAB Fields

	Name	Description
Output (Cont.)	STS	Completion status code. Note that if this field contains ER$EOF, the RSZ field will still describe a number of bytes transferred.
	STV	Status value. After successful $READ operations from a unit record or terminal device, STV is used to report the terminating character for the input block. Refer to the $GET operation in Section 9.3.6.2 in Chapter 9.

B.3 $WRITE - WRITING VIRTUAL BLOCKS

The $WRITE macro writes a user-specified number of bytes beginning on a specified block boundary to any of the RMS-11 file organizations. The user describes the blocks to be written in the RBF and RSZ fields. The BKT field must contain the virtual block number of the first block in the file to be written.

The formats of the $WRITE macro are as follows:

1. label:$WRITE

2. label:$WRITE rab,[,error[,success]]

where

label — is an optional user-defined symbol referring to the $WRITE macro.

rab — is the address of a Record Access Block containing the specification of the block(s) to be written.

error — is the address of a user completion routine to be called if the $WRITE operation fails.

success — is the address of a user completion routine to be called if the $WRITE operation succeeds.

Table B-2 lists the fields of the Record Access Block used during the $WRITE operation.

Table B-2
$WRITE RAB Fields

	Name	Description
Input	BKT	Bucket number. Must contain virtual block number of first block to be written.
	ISI	Internal stream identifier.
	RBF	Record address (word aligned). Address within user program of first byte of one or more blocks to be written.
	RSZ	Record size (must be an even number). This field will control the total amount of data transferred. Multiple virtual blocks can be written to a disk file by specifying the appropriate multiple of 512 bytes in this field. Partial blocks can be written but the contents of the unwritten portion of a block on disk are undefined.
Output	STS	Completion status code.
	STV	Status value. Actual number of bytes transferred.

B.4 $SPACE - FORWARD AND BACKWARD SPACING OF MAGNETIC TAPE FILES

When you open a file on magnetic tape in block mode, you can use the $SPACE macro call to space forward or backward in the file. RMS-11 returns an ER$IOP (illegal operation) if the file does not reside on magnetic tape or has not been opened for block I/O.

Table B-3 lists the fields of the Record Access Block used during the $SPACE operation.

Table B-3
$SPACE RAB Fields

	Name	Description
Input	BKT	Bucket number. Only the low order 16 bits of this field are examined. RMS-11 interprets these bits as representing a signed 15 bit integer. A positive integer represents the number of blocks the file is to be forward spaced. A negative integer represents the number of blocks the file is to be backspaced.
	ISI	Internal stream identifier.
	ROP	Record processing options. Can contain RB$ASY.
Output	STS	Completion status code.
	STV	Status value. Number of blocks spaced.

APPENDIX C

MAGNETIC TAPE HANDLING

The only form of magnetic tape structure supported by RMS-11 is the standard ANSI structure. This appendix describes the processing of magnetic tape files and the ANSI labeling and structuring format supported by host operating systems.

C.1 MAGNETIC TAPE FILE PROCESSING

IAS and RSX-11M support the standard ANSI magnetic tape structure as described in the June 19, 1974 proposed revision to "Magnetic Tape Labels and File Structure for Information Interchange," ANSI X3.27-1969. Any of the following file/volume combinations can be used:

1. Single file on a single volume,
2. Single file on more than one volume,
3. Multiple files on a single volume,
4. Multiple files on more than one volume.

Items 2 and 4 above constitute a volume set.

The sequence in which volume and file labels are used and the format of each label type is defined in Sections C.2 and C.3.

C.1.1 Access to Magnetic Tape Volumes

Magnetic tape is a sequential access, single-directory storage medium. Only one user can have access to a given volume set at a time. No more than one file in a volume set can be open at a time. Access protection is performed on a volume set basis. On volumes produced by DIGITAL systems, user access rights are determined by the contents of the owner identification field as described in Section C.2.1.1. Volumes produced by nonDIGITAL systems are restricted to read-only access unless explicitly overridden at MOUNT time.

C.1.2 Rewinding Volume Sets

A magnetic tape volume set can be rewound during a $OPEN, $CREATE, or $CLOSE macro call. Regardless of the method used to rewind the volume set, the following procedures are performed by the system:

1. All mounted volumes are rewound to BOT.

2. If the first volume in the set is not mounted, the unit to be used is placed offline.

3. If the volume is not already mounted and if the rewind was requested by a $OPEN or $CREATE macro call, a request to mount the first volume is printed on the operator's console.

4. If the rewind was requested by a $CLOSE macro call, no mount message is issued until the next volume is needed.

C.1.3 Positioning to the Next File Position

The FB$POS option in the file options (FOP) field of the FAB can be used to indicate that the file to be created is to be written immediately after the end of file labels of the most recently closed file. Any subsequent files in the volume set are lost.

If the rewind-on-open option (FB$RWO) also is specified, the file is created after the VOL1 label on the first volume of the set. All files that were previously contained in the volume set are lost.

To create a file in the next file position, the FB$POS option must be set in the FOP field. The default action of the file system is to position at the logical end of the volume set to create the file.

When the default is used, no check is made for the existence of a file with the same name in the volume set. Therefore, a program written to use magnetic tape normally should specify FB$POS.

C.1.4 Single File Operations

Single file operations are performed by specifying the FB$RWO and FB$RWC options for the $CREATE or $OPEN macro call. Using this approach, scratch tape operations can be performed as follows:

1. Create the first file with rewind specified,

2. Write the data records and close the file with rewind,

3. Open the first file again for input,

4. Read and process the data,

5. Close the file with rewind,

6. Create the second file with rewind specified,

7. Write the data records,

8. Close the file with rewind and perform any additional processing.

C.1.5 Multiple File Operations

A multiple file volume is created by creating, writing, and then closing a series of files without specifying the FB$RWO option. The sequential processing of files on the volume can be accomplished by not specifying the FB$RWC (rewind-on-close) option when opening the files.

Opening a file for extend (i.e., FB$PUT is specified in the FAC field of the FAB) is legal only for the last file on the volume set.

The following tape operations are performed to create a multiple file tape volume:

1. Create a file for output with rewind,
2. Write data records and close the file,
3. Create the next file with no rewind,
4. Write the data records and close the file,
5. Repeat for as many files as desired.

Files on tape can be opened in a nonsequential order, but increased processing and tape positioning time is required. Nonsequential access of files in a multiple volume set is not recommended.

C.2 VOLUME AND FILE LABELS

Tables C-1, C-2, and C-3 present the format of volume labels and file header labels.

C.2.1 Volume Label Format

Table C-1
Volume Label Format

Character Position	Field Name	Length in Bytes	Contents
1-3	Label identifier	3	VOL
4	Label number	1	1
5-10	Volume identifier	6	Volume label. Any alphanumeric or special character in the center four columns of the ASCII code table.

(Continued on next page)

Table C-1 (Cont.)
Volume Label Format

Character Position	Field Name	Length in Bytes	Contents
11	Accessibility	1	Any alphanumeric or special character in the center four columns of the ASCII code table. A space indicates no restriction. All volumes produced by IAS or RSX-11 have a space in this position.
12-37	Reserved	26	Spaces
38-51	Owner identification	14	The contents of this field are system-dependent and are used for volume protection purposes. See Section C.2.1.1 below.
52-79	Reserved	28	Spaces
80	Label standard version	1	1

C.2.1.1 **Contents of Owner Identification Field** - The owner identification field is divided into the following three subfields and a single pad character:

1. System identification (positions 38 through 40),
2. Volume protection code (positions 41 through 44),
3. UIC (positions 45 through 50),
4. Pad character of one space (position 51).

The system identification consists of the following character sequence.

D%x

x is the machine code and can be one of the following:

8 - PDP-8
A - DECsystem-10
B - PDP-11
F - PDP-15

The D%x characters provide an identification method so that the remaining data in the owner identification field can be interpreted. In the case of tapes produced on PDP-11 systems, the system identification is D%B and the volume protection code and UIC are interpreted as described below.

The volume protection code in positions 41 through 44 defines access protection for the volume for four classes of users. Each class of user has access privileges specified in one of the four columns as follows.

Position	Class
41	System (UIC no greater than [7,255])
42	Owner (group and member numbers match)
43	Group (group number matches)
44	World (any user not in one of the above)

One of the following access codes can be specified for each character position.

Code	Privilege
0	No access
1	Read only
2	Extend (append) access
3	Read/extend access
4	Total access

The UIC is specified in character positions 45 through 50. The first three characters are the group code in decimal. The next three are the user code in decimal.

The last character in the owner identification field is a space.

The following is an example of the owner identification field.

Owner identifier - D%B1410051102

1. The file was created on a PDP-11.

2. System and group have read access.
   Owner has total access.
   All others are denied access.

3. The UIC is [051,102].

C.2.2 User Volume Labels

User volume labels never are written or passed back to the user. If present, they are skipped.

C.2.3 File Header Labels

The following information should be kept in mind when creating file header labels:

- The Files-11 naming convention uses a subset (Radix-50) of the available ANSI character set for file identifiers.
- One character in the file identifier, the period (.), is fixed by Files-11.
- A maximum of 13 of the 17 bytes in the file identifier are processed by Files-11.
- It is strongly recommended that all file identifiers be limited to the Radix-50 PDP-11 character set and that no character other than the period (.) be used in the file type delimiter position for data interchange between PDP-11 and DECsystem-10 systems.

- For data interchange between DIGITAL and nonDIGITAL systems, the conventions listed above should be followed. If they are not, refer to Section C.2.3.1.

Tables C-2 and C-3 describe the HDR1 and HDR2 labels respectively.

Table C-2
File Header Label (HDR1)

Character Position	Field Name	Length in Bytes	Content
1-3	Label identifier	3	HDR
4	Label number	1	1
5-21	File identifier	17	Any alphanumeric or special character in the center four columns of the ASCII code table.
22-27	File set identifier	6	Volume identifier of the first volume in the set of volumes.
28-31	File section number	4	Numeric characters. This field starts at 0001 and is increased by 1 for each additional volume used by the file.
32-35	File sequence number	4	File number within the volume set for this file. This number starts at 0001.
36-39	Generation number	4	Numeric characters.
40-41	Generation version	2	Numeric characters.
42-47	Creation date	6	yyddd (with leading space) or 00000 (with leading space) if no date.
48-53	Expiration date	6	Same format as creation date.
54	Accessibility	1	Space
55-60	Block count	6	000000
61-73	System code	13	The three letters DEC followed by name of system that produced the volume. See Section C.2.1.1. Examples: DECFILE11A DECSYSTEM10 Pad name with spaces.
74-80	Reserved	7	Spaces

(Continued on next page)

Table C-3
File Header Format (HDR2)

Character Position	Field Name	Length in Bytes	Content
1-3	Label identifier	3	HDR
4	Label number	1	2
5	Record format	1	F - fixed length D - variable length S - spanned (not supported) U - undefined
6-10	Block length	5	Numeric characters
11-15	Record length	5	Numeric characters
16-50	System-dependent information	35	Positions 16 through 36 are spaces. Position 37 defines carriage control and can contain one of the following: A - first byte of record contains FORTRAN control characters, space - line feed/carriage return is to be inserted between records, M - the record contains all form control information. If DEC appears in positions 61 through 63 of HDR1, position 37 must be as specified above. Positions 38 through 50 contain spaces.
51-52	Buffer offset	2	Numeric characters. 00 on tapes produced by Files-11. Not supported on input to Files-11.
53-80	Reserved	28	Spaces

C.2.3.1 **File Identifier Processing by RMS-11** - The following steps describe the processing of a file identifier by RMS-11.

1. The first nine characters at a maximum are processed by an ASCII to Radix-50 converter. The filename scan continues until one of the following occurs:

   A conversion failure,
   9 characters are converted,
   A period (.) is encountered.

2. If the period is encountered, the next three characters after the period are converted and treated as the file type. If a failure occurs or all nine characters are converted, the next character is examined for a period. If it is a period, it is skipped and the next three characters are converted and treated as the file type.

3. The version number is derived from the generation number and the generation version number as follows.

   (generation number - 1)*100 + generation version + 1

At file output, the file identifier is handled as follows.

1. The filename is placed in the first positions in the file identifier field. It can occupy up to nine positions. It is followed by a period.

2. The file type of up to three characters is placed after the period. The remaining spaces are padded with spaces.

3. The version number is then placed in the generation and generation version number fields as described in the following formulas.

   a. generation number = $\frac{\text{version \# - 1 + 1}}{100}$

   b. generation version # = $\frac{\text{version \# - 1}}{\text{Modulo 100}}$

NOTE

In both calculations, remainders are ignored.

The following are examples.

VERSION #	GENERATION #	GENERATION VER #
1	1	0
50	1	49
100	1	99
101	2	0
1010	11	9

C.2.4 End-of-Volume Labels

End-of-volume labels are identical to the file header labels with the following exceptions:

1. Character positions 1 through 4 contain EOV1 and EOV2 instead of HDR1 and HDR2, respectively.

2. The block count field contains the number of records in the last file section on the volume.

C.2.5 File Trailer Labels

End-of-file labels (file trailer labels) are identical with file header labels with the following exceptions:

1. Columns 1 through 4 contain EOF1 and EOF2 instead of HDR1 and HDR2, respectively,

2. The block count contains the number of data blocks in the file.

C.2.6 User File Labels

User file labels never are written or passed back to the user. If present, they are skipped.

C.3 FILE STRUCTURES

The file structures illustrated below are the types of file and volume combinations that the file processor produces. Additional sequences can be read and processed by the file processor.

If HDR2 is not present, the data type is assumed to be fixed (F) and the block size and record size are assumed to be the default value for the file processor. 512 decimal bytes is the default for both block and record size. The minimum block size and fixed length record size is 18 bytes. The maximum block size is 8192 bytes.

The meaning of graphics used in the file structure illustrations is as follows.

1. * indicates a tape mark,

2. BOT indicates beginning of tape,

3. EOT indicates end of tape,

4. , indicates the physical record delimiter.

C.3.1 Single File Single Volume

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**

C.3.2 Single File Multi-Volume

```
BOT,VOL1,HDR1,HDR2*---DATA---*EOV1,EOV2**
BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**
```

C.3.3 Multi-File Single Volume

```
BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2*HDR1,HDR2*---DATA--*EOF1,EOF2**
```

C.3.4 Multi-File Multi-Volume

```
BOT,VOL1,HDR1,HDR2*--DATA--*EOF1,EOF2*HDR1,HDR2*--DATA--*EOV1,EOV2**
BOT,VOL1,HDR1,HDR2*--DATA--*EOF1,EOF2*HDR1,HDR2*--DATA--*EOF1,EOF2**
```

C.4 END OF TAPE HANDLING

End of tape is handled automatically by the magnetic tape file processor. Files are continued on the next volume providing the volume is already mounted or mounted upon request. A request for the next volume is printed on CO.

C.5 ANSI MAGNETIC TAPE FILE HEADER BLOCK (FCS COMPATIBLE)

Figure C-1 illustrates the format of a file header block that is returned by the file header READ ATTRIBUTE command for ANSI magnetic tape. The header block is constructed by the magnetic tape primitive from data within the tape labels.

ANSI MAGTAPE FCS-COMPATIBLE FILE
HEADER BLOCK

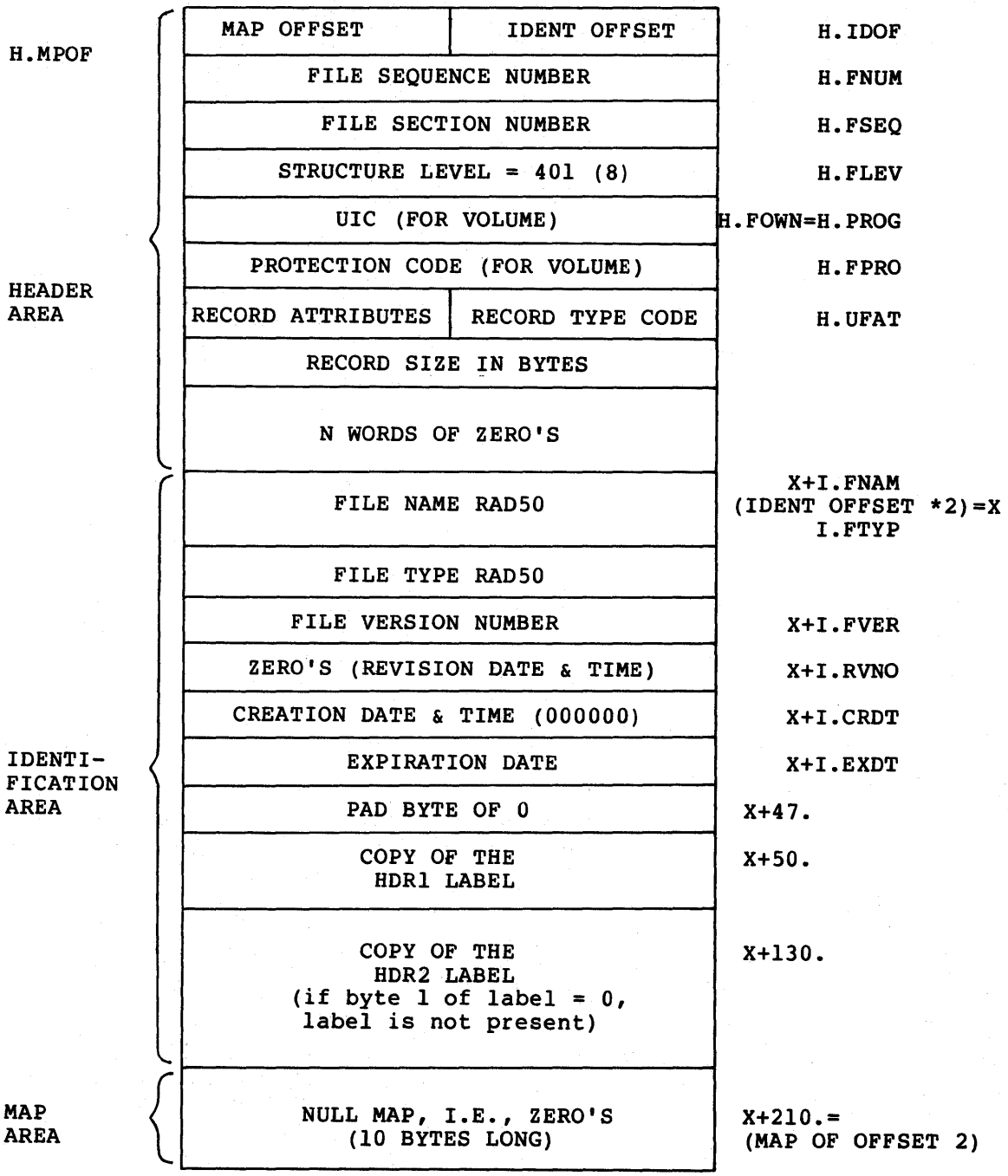

Figure C-1
ANSI Magnetic Tape File Header Block
(FCS Compatible)

APPENDIX D

FORMULAS

This appendix contains formulas that can be used to calculate the following:

1. Number of records per block in sequential files.
2. Number of user data records per bucket in relative or indexed files.
3. Number of entries per bucket in the index levels of all indexes and the data level of alternate indexes in an indexed file.

D.1 SEQUENTIAL FILES - AVERAGE RECORDS PER BLOCK

Table D-1
Average Records Per Block
in Sequential Files

	Sequential File Medium	
Record Format	Disk*	Magnetic Tape
Fixed	$\frac{512}{recsz}$	$\frac{BLS}{recsz}$
Variable	$\frac{512}{(recsz+2)}$	$\frac{BLS}{(recsz+4)}$
VFC	$\frac{512}{(recsz+FSZ+2)}$	$\frac{BLS}{(recsz+FSZ+4)}$

* Records in a sequential file on disk are word-aligned.

where

recsz — is the actual record size (for fixed format) or average record size (for variable or VFC) in bytes.

BLS — is the magnetic tape block size as specified in the BLS field of the FAB at the time the file was created.

FSZ — is the fixed control area size as specified in the FSZ field of the FAB at the time the file was created.

D.2 RELATIVE AND INDEXED FILES - AVERAGE DATA RECORDS PER BUCKET

Table D-2
Average User Data Records per Bucket

	File Organization	
Record Format	Relative	Indexed (data level of primary index)
Fixed	$\frac{DBKT}{(recsz+1)}$	$\frac{(DBKT-15)}{(recsz+7)}$
Variable	$\frac{DBKT}{(recsz+3)}$	$\frac{(DBKT-15)}{(recsz+9)}$
VFC	$\frac{DBKT}{(recsz+FSZ+3)}$	

where

DBKT — is the bucket size in bytes (a multiple of 512).

recsz — is:

1. The actual record size for fixed format records.
2. The maximum record size for variable or VFC format records in relative files.
3. The average record size for variable format records in indexed files.

FSZ — is the fixed control area size as specified in the FSZ field of the FAB at the time the file was created.

D.3 INDEXED-FILES - AVERAGE ENTRIES PER INDEX AND ALTERNATE KEY DATA LEVEL

Table D-3
Average Entries Per Index and Alternate Key Data Level Bucket

	Bucket Type		
	Index Level (primary and alternate	Data Level (alternate key - no duplicates allowed)	Data Level (alternate key - duplicates allowed
Average Entries per Bucket	$\frac{(IBKT-15)}{(keysz+3)}$	$\frac{(DBKT-15)}{(keysz+9)}$	$\frac{(DBKT-15)}{(keysz+8+(5*dups))}$

where

IBKT	is the size in bytes of buckets in the index level (a multiple of 512).
DBKT	is the size in bytes of buckets in the data level of the alternate key index (a multiple of 512).
keysz	is the key size in bytes.
dups	is the number of records with identical values for a particular instance of a key value.
3,9,5	are average values representing actual ranges of, respectively, 2 to 4, 8 to 10, and 4 to 6.

APPENDIX E

SAMPLE CODE SEGMENTS

The sample code segments in this appendix demonstrate the copying of records from an existing sequential file to a new sequential file.

```
; DEMO.RMS

; PROGRAM TO COPY RECORDS FROM A SEQUENTIAL FILE NAMED
; FILE1.DAT  TO A NEW SEQUENTIAL FILE NAMED  FILE2.DAT

; STEP 1:  ACCESS THE NECESSARY RMS MACROS

        .MCALL  $INIT,ORG$,FAB$B,RAB$B,POOL$B,$CREATE,$OPEN,$CLOSE
        .MCALL  $CONNECT,$GET,$PUT,$FETCH,$STORE,$COMPARE

; STEP 2:  DEFINE CONTROL BLOCKS AND FILE NAME STRINGS

        ;THE FAB FOR THE INPUT FILE, WHICH ALREADY EXISTS, WILL BE
        ;FILLED WITH MOST OF THE ESSENTIAL INFORMATION CONCERNING
        ;FILE1.DAT WHEN THE FILE IS OPENED. THE PROGRAM NEED ONLY
        ;SPECIFY ENOUGH INFORMATION TO OPEN THE FILE.

FAB1:   FAB$B                   ;FAB FOR FILE1.DAT
        F$FNA   NAME1           ;ADDRESS OF NAME STRING
        F$FNS   9               ;STRING IS 9 CHARACTERS LONG
        F$LCH   1               ;ACCESS ON CHANNEL 1
        FAB$E                   ;END OF FAB1

NAME1:  .ASCII  /FILE1.DAT/     ;NAME STRING FOR FAB1
        ;HERE, AND WITH FILE2, WE ASSUME THAT THE FILE EXISTS
        ;ON THE SYSTEM DISK UNDER THE ACCOUNT ON WHICH WE ARE
        ;LOGGED IN.
        .EVEN                   ;(CONTROL BLOCKS MUST BE WORD ALIGNED)

        ;THE FAB FOR THE OUTPUT FILE, WHICH DOES NOT YET EXIST, MUST
        ;BE FILLED WITH THE INFORMATION NECESSARY TO CREATE IT (AS IN
        ;THE PREVIOUS CASE, SOME FIELDS SIMPLY CONTAIN DEFAULT VALUES
        ;AND ARE NOT EXPRESSED EXPLICITLY). SOME OF THIS INFORMATION
        ;WILL DEPEND ON THE CHARACTERISTICS OF FILE1.DAT, AND MUST BE
        ;FILLED IN AT RUN-TIME AFTER FILE1.DAT HAS BEEN OPENED.

FAB2:   FAB$B                   ;FAB FOR FILE2.DAT
        F$FNA   NAME2           ;ADDRESS OF NAME STRING
        F$FNS   9               ;STRING IS 9 CHARACTERS LONG
        F$LCH   2               ;ACCESS ON CHANNEL 2
        F$FAC   FB$PUT          ;WRITE ACCESS REQUIRED
        FAB$E                   ;END OF FAB2

NAME2:  .ASCII  /FILE2.DAT/     ;NAME STRING FOR FAB2
```

```
        .EVEN

RAB1:   RAB$B                   ;RAB FOR FILE1.DAT
        R$FAB   FAB1            ;ADDRESS OF OWNER (FAB)
        R$RAC   RB$SEQ          ;SPECIFY SEQUENTIAL ACCESS
        R$UBF   RECBUF          ;ADDRESS OF RECORD BUFFER FOR $GETS
        R$USZ   500             ;SIZE OF THIS BUFFER (500. BYTES)
        R$RHB   HEDBUF          ;ADDRESS OF RECORD HEADER BUFFER
                                ;(NECESSARY FOR VFC RECORDS ONLY)
        RAB$E                   ;END OF RAB1

RAB2:   RAB$B                   ;RAB FOR FILE2.DAT
        R$FAB   FAB2            ;ADDRESS OF OWNER
        R$RAC   RB$SEQ          ;SPECIFY SEQUENTIAL ACCESS
        R$RBF   RECBUF          ;ADDRESS OF RECORD BUFFER FOR $PUTS
        R$RSZ   500             ;SIZE OF THIS BUFFER (500. BYTES)
        R$RHB   HEDBUF          ;ADDRESS OF RECORD HEADER BUFFER
                                ;(NECESSARY FOR VFC RECORDS ONLY)
        RAB$E                   ;END OF RAB2

; STEP 3:  ALLOCATE THE BUFFERS SPECIFIED ABOVE

RECBUF: .BLKW   250.

HEDBUF: .BLKW   128.

; STEP 4:  GENERATE RMS INTERNAL SPACE POOL

        POOL$B                  ;BEGIN POOL SPECIFICATION
        P$FAB   2               ;A FAB FOR EACH FILE
        P$RAB   2               ;A RAB FOR EACH FAB
        P$BDB   2               ;AN I/O BUFFER FOR EACH RAB
        P$BUF   1024            ;MINIMUM BUFFER SIZE IS 512. BYTES
        POOL$E                  ;END OF POOL SPECIFICATION

; STEP 5:  DEFINE THE RMS FUNCTIONALITY REQUIRED

        ORG$    SEQ,<CRE,GET,PUT>       ;SEQUENTIAL FILES ONLY, $FIND,
                                        ;$UPDATE, $DELETE NOT REQUIRED

; STEP 6:  PROVIDE A GENERAL ERROR ROUTINE TO HANDLE UNEXPECTED ERRORS
;          WHICH MIGHT OCCUR (WHAT IF FILE1.DAT DID NOT EXIST, OR CON-
;          TAINED A RECORD LARGER THAN 500 BYTES?). THIS IS AN ALTER-
;          NATIVE TO THE 'COMPLETION ROUTINE' FUNCTION PROVIDED BY RMS.

ERROR:  ;(CODE WHICH WILL HANDLE THE ERROR, PROMPT AT THE TERMINAL FOR
        ;FURTHER INSTRUCTIONS, ETC.)

; STEP 7:  WRITE THE PROGRAM

START:  $INIT                   ;INITIALIZE RMS.
        $OPEN   #FAB1           ;OPEN FILE1.DAT,
        MOV     #FAB1,R0        ;SET UP FOR $COMPARE:
        $COMPARE #0,STS,R0      ;NEGATIVE STS VALUE IMPLIES OPEN FAILURE
        BLT     1$              ;BRANCH IF SUCCESSFUL
        JSR     PC,ERROR        ;OTHERWISE EXECUTE ERROR ROUTINE.
1$:     MOV     #FAB2,R1        ;COMPLETE INITIALIZATION OF FAB2:
        $FETCH  R2,RAT,R0       ;GET RAT FIELD FROM FAB1
        $STORE  R2,RAT,R1       ;AND MOVE IT INTO FAB2;
        $FETCH  R2,RFM,R0       ;DO THE SAME WITH THE RFM FIELD;
        $STORE  R2,RFM,R1
        $FETCH  R2,FSZ,R0       ;FSZ IS PERTINENT ONLY IF FILE1.DAT
        $STORE  R2,FSZ,R1       ;MAY CONTAIN VFC RECORDS.
        $FETCH  R2,MRN,R0       ;YOU MAY OR MAY NOT WISH TO TRANSFER
```

```
        $STORE  R2,MRN,R1       ;MRN, MRS, AND FOP. IF MRN IS COPIED,
        $FETCH  R2,MRS,R0       ;REMEMBER THAT IT IS A 2-WORD FIELD,
        $STORE  R2,MRS,R1       ;AND WILL DESTROY THE CONTENTS OF R3.

        ;INITIALIZATION OF FAB2 SHOULD NOW BE COMPLETE EXCEPT FOR ANY
        ;SPECIAL CASES (FOR EXAMPLE, IF FILE2.DAT IS ON MAGNETIC TAPE,
        ;YOU MAY WISH TO SET THE BLS FIELD).

        $FETCH  R2,ALQ,R0
        $STORE  R2,ALQ,R1
        $CREATE R1              ;NOW CREATE FILE2.DAT (R1=FAB2):
        $COMPARE #0,STS,R1      ;CHECK FOR FAILURE
        BLT     2$              ;BRANCH IF SUCCESSFUL
        JSR     PC,ERROR        ;OTHERWISE EXECUTE ERROR ROUTINE.
2$:     MOV     #RAB1,R0        ;CONNECT THE RABS.
        MOV     #RAB2,R1
        $CONNECT R0
        $COMPARE #0,STS,R0
        BLT     3$              ;BRANCH IF SUCCESSFUL
        JSR     PC,ERROR
3$:     $CONNECT R1
        $COMPARE #0,STS,R1
        BLT     4$
        JSR     PC,ERROR
4$:     $GET    R0              ;GET A RECORD FROM FILE1.DAT.
        $COMPARE #ER$EOF,STS,R0         ;WERE WE AT END-OF-FILE?
        BEQ     DONE            ;IF SO, CLEAN UP AND EXIT.
        $COMPARE #0,STS,R0      ;SOME OTHER ERROR?
        BLT     5$              ;BRANCH IF SUCCESSFUL
        JSR     PC,ERROR        ;IF SO, HANDLE IT.
5$:     $FETCH  R2,RSZ,R0       ;COPY RECORD LENGTH FROM
        $STORE  R2,RSZ,R1       ;RAB1 TO RAB2.
        $PUT    R1              ;OUTPUT THE RECORD TO FILE2.DAT.
        $COMPARE #0,STS,R1
        BLT     6$              ;BRANCH IF SUCCESSFUL
        JSR     PC,ERROR
6$:     BR      4$              ;LOOP UNTIL DONE.
DONE:   MOV     #FAB1,R0        ;BACK TO THE FABS FOR $CLOSE.
        MOV     #FAB2,R1
        $CLOSE  R0
        $COMPARE #0,STS,R0
        BLT     1$              ;BRANCH IF SUCCESSFUL
        JSR     PC,ERROR
1$:     $CLOSE  R1
        $COMPARE #0,STS,R1
        BLT     2$
        JSR     PC,ERROR
2$:
        ;RMS IS NOW DONE:  FILE2.DAT NOW CONTAINS ALL THE DATA RECORDS
        ;OF FILE1.DAT, PLUS WHATEVER OTHER INFORMATION (MRS, MRN, ETC.)
        ;YOU CHOSE TO DUPLICATE. INSERT YOUR OWN EXIT CODE AND EXIT.

        .END    START
```

APPENDIX F

ASSEMBLING AND TASK BUILDING

When you assemble a program that uses RMS-11 facilities, you must reference the following file as a macro library in your command line:

```
[1,1]RMSMAC.MLB
```

When you task build without RMS-11 overlays, you must include in the task builder command line or user ODL root statement a reference to the following file as an object library:

```
[1,1]RMSLIB.OLB
```

When you task build with RMS-11 overlays, you must:

1. Edit a private copy of the RMS-11 prototype ODL to include the needed portions of RMS-11. The prototype ODL is in the following file:

   ```
   [1,1]RMS11.ODL
   ```

2. Include references in the root statement of your ODL to the following factors:

   a. RMSROT
   b. RMSALL

   RMSALL can be referenced as a co-tree. If RMSALL is a co-tree, you must specify the full search option to the task builder.

3. Reference, in your overlay description language, your private copy of the edited RMS11.ODL as an indirect file.

INDEX

IAS/RSX-11M
RMS-11 MACRO Programmer's
Reference Manual
AA-0002A-TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

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
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital equipment corporation

Printed in U.S.A.