pdp11

IAS/RSX-11
I/O Operations
Reference Manual

Order No. DEC-11-OIORA-A-D

digital

IAS/RSX-11
I/O Operations
Reference Manual

Order No. DEC-11-OIORA-A-D

IAS Version 1

RSX-11M Version 2

RSX-11D Version 6A (Version 6.1)

digital equipment corporation · maynard. massachusetts

First Printing, December 1975

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.

10/76-14

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

0.1 MANUAL OBJECTIVES AND READER ASSUMPTIONS

The purpose of this manual is to familiarize the users of an RSX-11D, RSX-11M, or IAS operating system with the file management facilities provided with the system. Since the file control services described herein pertain to both MACRO-11 and FORTRAN programs, the reader is assumed to be familiar with the manuals describing these program development tools. Also, since the development of programs in an RSX-11 or IAS environment necessarily involves the use of the Task Builder, the reader is likewise assumed to be familiar with this system program. Unless otherwise noted, the term RSX-11 refers to both RSX-11D and RSX-11M.

0.2 STRUCTURE OF THE DOCUMENT

Chapter 1 briefly describes the file control services available for IAS/RSX-11 users and defines some of the terminology that is pertinent to discussions throughout the manual. This chapter is vital to understanding the balance of the manual.

Chapter 2, perhaps the most important in the manual, describes the actions the user must take at assembly-time to prepare adequately for all intended file I/O processing. This chapter describes the data structures and working storage areas that the user must define within his program in order to use any of the file control services provided by the system. Unless the user is thoroughly familiar with the content of this chapter, he is advised to defer a reading of subsequent chapters, since all that follows is dependent upon a complete working understanding of the material in Chapter 2.

Chapter 3 describes the run-time macro calls which allow the user to manipulate files and to perform I/O operations.

Chapter 4 describes a set of run-time routines used to perform functions related to controlling files, such as reading and writing directory entries, renaming or extending files, etc.

Chapter 5 describes the structure of files supported by the IAS and RSX-11 systems. In this context, the structure of files for disks, DECtapes, and magnetic tapes are covered.

Chapter 6 describes two collections of object library routines called the Get Command Line Routine (GCML) and the Command String Interpreter (CSI). These routines may be linked with the user task to perform operations associated with the dynamic input of command lines. Such input consists of file specifications which identify and control the files to be processed by the user program.

Chapter 7 describes the queuing of files for printing. This facility is available at both the MACRO level and subroutine level.

Finally, a number of appendices are provided which supply detailed information of further interest.

Appendix A and Appendix B outline in detail the file descriptor block and the filename block, respectively, two structures forming a significant part of the descriptive material in Chapter 2. Appendix C summarizes a number of I/O-related system directives that form a part of the total resource management capabilities of the RSX-11 or the IAS Executive. Through simplified sample programs, Appendix D illustrates the use of the macro calls that create and initialize the file descriptor block. These sample programs also include some of the macro calls that are used for processing files.

Appendix E illustrates the structure of index files, while Appendix F describes in detail the format and content of a file header block. The format and content of magnetic tape labels (not used in RSX-11M) are similarly described in Appendix G. The format and content of the statistics block are described in Appendix H.

The error codes returned by the system are listed in Appendix I and field size symbols are listed in Appendix J.

0.3 ASSOCIATED DOCUMENTS

Other manuals closely allied to the purposes of this document are described briefly in the IAS, RSX-11D, and RSX-11M/RSX-11S Documentation Directories. The Documentation Directories define the intended readership of each manual in the appropriate set and provide a brief synopsis of each manual's contents. The directories and order numbers are listed below:

IAS Documentation Directory, Order No. DEC-11-OIDDA-A-D

RSX-11D Documentation Directory, Order No. DEC-11-OXUGA-C-D

RSX-11M/RSX-11S Documentation Directory, Order No. DEC-11-OMUGA-B-D

CHAPTER 1

FILE CONTROL SERVICES

IAS and RSX-11 file control services (FCS) enable the user to perform record-oriented and block-oriented I/O operations and to perform additional functions required for file control, such as open, close, wait, and delete operations. To invoke FCS functions, the user issues macro calls to specify desired file control operations. The FCS macros are called at assembly-time to generate code for specified functions and operations. The macro calls provide the system-level file control primitives with the necessary parameters to perform the file access operations requested by the user (see Figure 1-1).

FCS is basically a set of routines that are linked with the user program at task-build time from a system global area (IAS and RSX-11D) or resident system library (RSX-11M); or a system object module library. These routines, consisting of pure, position-independent code, provide a user interface to the file system, enabling the user to read and write files on file-structured devices and to process files in terms of logical records.

Logical records are regarded by the user program as data units that are structured in accordance with application requirements, rather than existing merely as physical blocks of data on a particular storage medium.

FCS provides the capability to write a collection of data (consisting of distinct logical records) to a file in a way that enables the data to be retrieved at will. Data can be retrieved from the file without having to know the exact format in which it was written to the file.

FCS thus provides a sense of transparency to the user so that records can be read or written in logical units that are consistent with his application requirements.

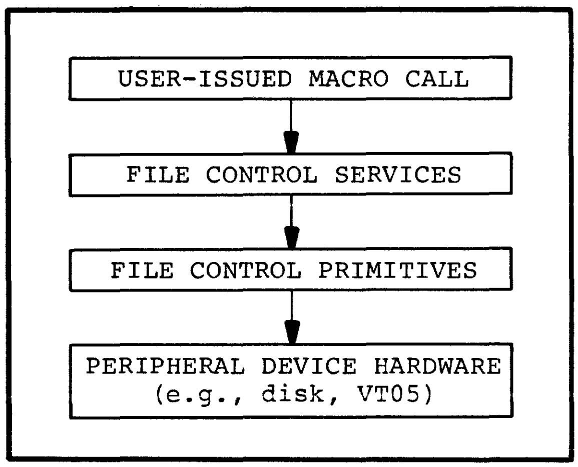

Figure 1-1
File Access Operation

FCS provides an extensive set of macros to simplify the user's interface to the system's I/O facilities. These macros create and maintain certain data structures that are required in performing all file I/O operations. The required structures include the following:

1. A file descriptor block (FDB) that contains execution-time information necessary for the processing of a file.

2. A dataset descriptor that is accessed by FCS to obtain ASCII file information required in opening a specified file.

3. A default filename block that is accessed by FCS to obtain default file information required in opening a specified file. This structure is accessed when complete file information is not specified in the dataset descriptor.

The file descriptor block is described in detail in Appendix A and Appendix B. The dataset descriptor and the default filename block are treated in detail in section 2.4.

1.1 FILE ACCESS METHODS

IAS and RSX-11 support both sequential and direct access to files. The sequential access method is device-independent, i.e., sequential access can be used for both record-oriented and file-structured devices (e.g., card reader and disk, respectively). The direct access method can be used only for file-structured devices.

1.2 FILE STORAGE REGION (FSR)

The file storage region (FSR) is an area allocated in the user program as working storage for performing record I/O operations (see section 1.5). The FSR consists of two program sections which are always contiguous to each other. These program sections exist for the following purposes:

$$FSR1 - This area of the FSR contains the block buffers and the block buffer headers for record I/O processing. The user determines the size of this area at assembly-time by issuing the FSRSZ$ macro call (see section 2.6.1). The number of block buffers and associated headers is based on the number of files that the user intends to open simultaneously for record I/O operations.

$$FSR2 - This area of the FSR contains impure data that is used and maintained by FCS in performing record I/O operations. Portions of this area are initialized at task-build time, and other portions are maintained by FCS.

The size of the FSR can be changed, if desired, at task-build time. Section 2.7 presents the procedures which provide this flexibility to the programmer.

The data flow during record I/O operations is depicted in Figure 1-2. Note that blocks of data are transferred directly between the FSR block buffer and the device containing the desired file. The blocking and deblocking of records during input is accomplished in the FSR block buffer, and the building of records is likewise accomplished in the FSR block buffer during output. Note also that FCS serves as the user interface to the FSR block buffer pool. All record I/O operations, which are initiated through GET$ and PUT$ macro calls, are totally synchronized by FCS.

Record I/O operations are described in greater detail in section 1.5.

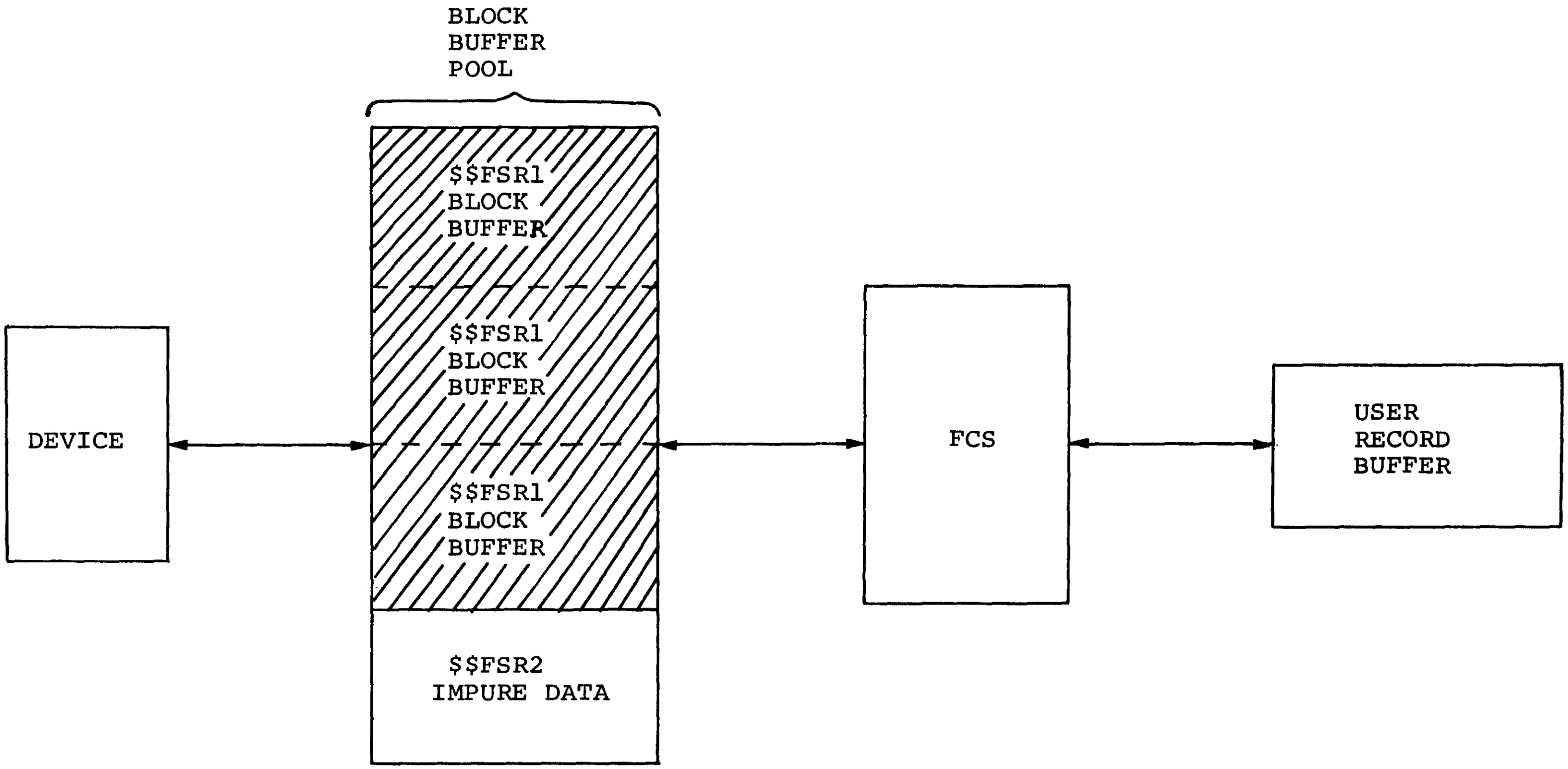

Figure 1-2
Record I/O Operations

1.3 DATA FORMATS FOR FILE-STRUCTURED DEVICES

Data is transferred between peripheral devices and memory in blocks. A data file consists of virtual blocks, each of which may contain one or more logical records. In FCS terms, a virtual block in a file consists of 512(10) bytes.

Records in a virtual block can be either fixed or variable in length. The term "fixed-length" refers to records which are equal and non-varying in length; conversely, the term "variable length" refers to records which are not equal in length. The first two bytes of a variable-length record contain a value defining the length of that record (in bytes), excluding the record length bytes.

Both fixed and variable length records are aligned on a word boundary. Any extra byte that results from an odd-length record is simply ignored. (The extra byte is not necessarily a 0 byte.)

Virtual blocks and logical records within a file are numbered sequentially, each starting at one (1). A virtual block number is a file relative value, while a logical block number is a volume relative value.

1.4 BLOCK I/O OPERATIONS

The READ$ and WRITE$ macro calls (see sections 3.15 and 3.16, respectively) allow the user to read and write virtual blocks of data from and to a file without regard to logical records within the file. Block I/O operations provide a very efficient means of processing file data, since such operations do not involve the blocking and deblocking of records within the file. Also, in block I/O operations, the user may read or write files in an asynchronous manner, i.e., control may be returned to the user program before the requested I/O operation is completed.

When block I/O is used, the number of the virtual block to be processed is specified as a parameter in the appropriate READ$/WRITE$ macro call; the virtual block so specified is processed directly in a buffer reserved by the user in his own memory space.

As implied above, the user is responsible for synchronizing all block I/O operations. Such asynchronous operations may be coordinated through an event flag (see section 2.8.1) specified in the READ$/WRITE$ macro call. The event flag is used by the system to signal the completion of a specified block I/O transfer, enabling the user to coordinate those block I/O operations which are dependent on each other.

1.5 RECORD I/O OPERATIONS

The GET$ and PUT$ macro calls (see sections 3.9 and 3.12, respectively) are provided for processing record-oriented files. Using the FSR block buffers (see section 1.2), GET$ and PUT$ operations perform the necessary blocking and deblocking of records within the virtual blocks of the file, allowing the user to read or write individual records.

In preparing for record I/O operations, the user must specify the format of the records. For example, he must specify whether the records are fixed or variable in length, or whether records that are to be output to a carriage-control device are to contain carriage-control information (either at the beginning of the record or embedded within the record).

For sequential access files, I/O operations can be performed for both fixed- and variable-length records. For direct access files, I/O operations can be performed only for fixed-length records.

In contrast to block I/O operations, all record I/O operations are synchronous, i.e., control is returned to the user program only after the requested I/O operation is completed.

Because GET$/PUT$ operations process logical records within a virtual block, only a limited number of GET$ or PUT$ operations result in an actual I/O transfer, e.g., when the end of a data block is encountered. Therefore, all GET$/PUT$ I/O requests will not necessarily involve an actual physical transfer of data.

1.6 DATA TRANSFER MODES

When record I/O is used, a program can gain access to a record in either of two ways after the virtual block has been transferred into the FSR from a file:

1. In move mode. By specifying that individual records are to be moved from the FSR block buffer to a user-defined record buffer (see Figure 1-2).

2. In locate mode. By referencing a location in the file descriptor block (see section 1.9) which contains a pointer to the desired record within the FSR block buffer.

1.6.1 Move Mode

Move mode requires that data be moved between the FSR block buffer and a user-defined record buffer. For input, data is first read into the FSR block buffer from a peripheral device and then moved to the user record buffer for processing. For output, the user program first builds a record in the user record buffer; FCS then moves the record to the FSR block buffer, from whence it is written to a peripheral device when the entire block is filled.

Move mode simulates the reading of a record directly into a user record buffer, thereby making the blocking and deblocking of records transparent to the user.

1.6.2 Locate Mode

Locate mode enables the user to access records directly in the FSR block buffer. Consequently, there is normally no need to transfer data from the FSR block buffer to the user record buffer. To access records directly in the FSR block buffer, the user refers to locations in the file descriptor block (see section 1.9) which contain values defining the length and the address of the desired record within the FSR block buffer. These values are present in the file descriptor block as a result of FCS macro calls issued by the user.

Program overhead is reduced in locate mode, since records can be processed directly within the FSR block buffer. Moving data to the user record buffer in locate mode is required only when the last record of a virtual block crosses block boundaries.

1.7 MULTIPLE BUFFERING FOR RECORD I/O (IAS AND RSX-11D ONLY)

By supporting multiple buffers for record I/O, FCS provides the capability for IAS and RSX-11D users to read data into buffers in anticipation of user program requirements and to write the contents of buffers while the user program is building records for output. The user can thus overlap the internal processing of data with file I/O operations, as illustrated in Figure 1-3.

When read-ahead multiple buffering is used, the file must be sequentially accessed to derive full benefit from multiple buffering. For write-behind multiple buffering, any file access method can be used with full benefit.

When multiple buffering is used, sufficient space in the FSR must be allocated for the total number of block buffers in use at any given time. The FSRSZ$ macro call (see section 2.6.1) is used to accomplish the allocation of space for FSR block buffers.

Time →					
Single Buffer	process record	write record	process record	write record	...
Multiple Buffer	process record	write record process record	process record write record	write record process record	...

Figure 1-3
Single Buffering Versus Multiple Buffering

1.8 SHARED ACCESS TO FILES

FCS permits shared access to files according to established conventions. Two macro calls, among several available in FCS for opening files, may be issued to invoke these conventions. The OPNS$x macro call (see section 3.2) is used specifically to open a file for shared access. The OPEN$x macro call (see section 3.1), on the other hand, invokes generalized open functions which have shared access implications only in relation to other I/O requests then issued. Both macro calls take an alphabetic suffix which specifies the type of operation being requested for the file, as follows:

R - Read existing file.

W - Write (create) a new file.

M - Modify existing file without extending its length.

U - Update existing file and extend its length, if necessary.

A - Append data to end of existing file.

The suffix R applies to the reading of a file, while the suffixes W, M, U, and A all apply to the writing of a file. These macro calls and the shared access conditions which they invoke are summarized below.

The OPNS$x and OPEN$x macro calls may be used as follows for shared access to files:

1. When the OPNS$R macro call is issued, read access to the file is granted unconditionally, regardless of the presence of a concurrent write-access request to the file. (The OPNS$R macro call permits concurrent write access to the file while it is being read.) A subsequent write-access request for this same file will be honored, provided that only one such request is active at any given time. Thus, several active read-access requests and one write-access request may be present for the same file.

   Other concurrent OPNS$x macro calls are equivalent to their OPEN$x counterpart, since only one writer of a file is permitted under any circumstances.

2. When the OPEN$R macro call is issued, read access to the file is granted, provided that no write-access request for that file is active. (The OPEN$R macro call does not permit concurrent write access to the file while it is being read.)

Note from the above that there can be several concurrent readers of a file, but only one writer of that same file. Readers of a shared file should be aware that the file may yield inconsistent data from request to request if that file is also being written.

Shared access during reading does not necessarily imply the presence of read requests from several separate tasks. The same task, for example, may open the same file using different logical unit numbers.

1.9 FILE DESCRIPTOR BLOCK (FDB)

The file descriptor block (FDB) contains information used by FCS in opening and processing files. One FDB is required for each file that is to be opened simultaneously by the user program. The user initializes some portions of the FDB with assembly-time or run-time macro calls, and FCS maintains other portions. Each FDB has five sections that contain user or system-initialized information:

- . File Attribute Section;
- . Record or Block Access Section;
- . File-Open Section;
- . Block Buffer Section; and the
- . Filename Block Portion of the FDB.

The information stored in the FDB depends upon the characteristics of the file to be processed. The FDB and the macro calls that cause values to be stored in this structure are described in detail in section 2.2. Appendix A describes the format and the content of the FDB in detail.

1.10 DATASET DESCRIPTOR AND DEFAULT FILENAME BLOCK

Normally, either a dataset descriptor or a default filename block is specified for each file that the user intends to open. These data structures provide FCS with the file specifications required for opening a file.

Although either one or the other is usually defined, both can be specified for the same file. The dataset descriptor and the default filename block are summarized below and described in detail in section 2.4.1 and 2.4.2, respectively.

When a file is being opened using information already present in the filename block, neither the dataset descriptor nor the default filename block is accessed by FCS for required file information. This method of file access, which is termed "opening a file by file ID," is a very efficient means of opening files. Section 2.5 describes this process in detail.

1.11 KEY TERMS USED THROUGHOUT THIS MANUAL

Listed below are the terms used throughout this manual which have specific meanings in the context of FCS operations.

FILE DESCRIPTOR BLOCK (FDB) -- The tabular data structure that provides FCS with information needed to perform I/O operations on a file. The space for this data structure is allocated in the user program by issuing the FDBDF$ macro call (see section 2.2.1.1). Each file to be opened simultaneously by the user program must have an associated FDB. Portions of the FDB are user-defined and others are maintained by FCS. Assembly-time or run-time macro calls are provided for user initialization of the FDB. The format and content of the FDB are detailed in Appendix A.

FILENAME BLOCK -- The portion of the FDB that contains the various elements of a file specification (i.e., directory, filename, file type, file version number, device, and unit) for use by the FCS file-processing routines. Initially, as a file is opened, FCS fills in the filename block with user-specified information taken from the dataset descriptor and/or the default filename block (see below). The methods of creating file specifications for initializing the filename block are described in detail in section 2.4; the format and content of the filename block itself are described in Appendix B.

DEFAULT FILENAME BLOCK -- The default filename block, an area allocated within the user program by issuing the NMBLK$ macro call (see section 2.4.2), contains the various elements of a file specification. The default filename block is a user-created structure, while the filename block within the FDB is maintained by FCS. The user creates the default filename block to supply file specifications to FCS that are not otherwise available through the dataset descriptor (see below). In other words, from information defined in the default filename block, FCS creates a parallel structure in the FDB that serves as the execution-time repository for information that FCS requires in opening and operating on files.

Thus, the terms "default filename block" and "filename block" refer to separate and distinct data structures. These distinctions should be kept clearly in mind whenever these terms appear in the manual. Though created and used differently, these areas are structurally identical.

DATASET DESCRIPTOR -- The dataset descriptor is a 6-word block in the user program containing the sizes and the addresses of ASCII data strings that together constitute a file specification (see below). This data structure, which is also created by the user, is described in detail in section 2.4.1. Unless the filename block in the FDB has been saved, dataset descriptor and/or default filename block information must be provided to FCS before the specified file can be opened.

DATASET DESCRIPTOR POINTER -- An address value that points to the 6-word dataset descriptor within the user program. This address value is stored in the FDB, allowing FCS to access a user-created file specification in the dataset descriptor.

FILE SPECIFICATION -- Any system or user program having a requirement to refer to files does so through a file specification. Such information names a file and allows it to be explicitly referenced by any task. A file specification, whether for input or output, contains specific information which must be made available to FCS before that file can be opened. The term "file specifier," is sometimes used as a synonym for "file specification."

FILE STORAGE REGION (FSR) -- The file storage region (see section 1.2) is an area of memory reserved by the user for use in record I/O operations. This area is allocated by issuing the FSRSZ$ macro call in the user program (see section 2.6.1).

1.12 SYSTEM CHARACTERISTICS

Listed below are the important characteristics of FCS that should be borne in mind in order to use its I/O facilities properly:

1. READ$/WRITE$ operations are asynchronous; the user is responsible for coordinating all block I/O activity. In contrast, GET$/PUT$ operations are synchronized entirely by FCS; control is not returned to the user program until the requested GET$/PUT$ operation is completed.

2. FCS macro calls save and restore all registers, with the following exceptions:

   a. The file-processing macro calls (see Chapter 3) place the FDB address in R0.

   b. Many of the file-control routines (see Chapter 4) return requested information in the general registers.

3. The FDBDF$ macro call (see section 2.2.1.1) is issued to allocate space for an FDB. Once the FDB is allocated, necessary information can be placed in this data construct through any logical combination of assembly-time and/or run-time macro calls (see sections 2.2.1 and 2.2.2, respectively). Certain information must be present in the FDB before FCS can open and operate on a specified file.

4. For each assembly-time FDB initalization macro call, a corresponding run-time macro call is provided that supplies identical information. Although both sets of macro calls (see Table 2-1) place the same information in the FDB, each set does so in a different way. The assembly-time calls generate .BYTE or .WORD directives which create specific data, while the run-time calls generate MOV or MOVB instructions which place desired information in the FDB during program execution.

5. If an error condition is detected during any of the file processing operations described in Chapter 3, or during the execution of several of the file-control routines (see section 4.1), the C-bit (carry condition code) in the Processor Status Word is set, and an error indicator is returned to FDB offset location F.ERR.

   If the address of a user-coded error-handling routine is specified as a parameter in any of the file-processing macro calls, a JSR PC instruction to the error-handling routine is generated. The routine is then executed if the C-bit in the Processor Status Word is set.

CHAPTER 2

PREPARING FOR I/O

The MACRO-11 programmer must establish the proper data base and working storage areas within his program in order to perform input/output operations. The following actions must be performed:

- A file descriptor block (FDB) must be defined for each file that is to be opened simultaneously by the user program (see section 2.2).

- A dataset descriptor and/or a default filename block (see section 2.4.1 or 2.4.2, respectively) must also be defined if the user intends to access these structures to provide required file specifications to FCS.

- A file storage region (FSR) must be established within the program if the user intends to employ record I/O in processing files (see section 2.6). (The initialization procedures for FORTRAN programs are described in detail in the FORTRAN-IV User's Guide.)

This chapter describes the macro calls that must be invoked to provide the necessary file processing information for the FDB. Such information is placed in the FDB in one of three ways:

1. By the assembly-time FDB initialization macro calls (see section 2.2.1).

2. By the run-time FDB initialization macro calls (see section 2.2.2).

3. By the file-processing macro calls (see Chapter 3).

Data supplied during the assembly of the source program establishes the initial values in the FDB. Data supplied at run-time can either initialize additional portions of the FDB or change values established at assembly-time. Likewise, the data supplied through the file-processing macro calls can either initialize portions of the FDB or change previously-initialized values.

Table 2-1 lists the macro calls that generate FDB information.

Table 2-1
Macro Calls Generating FDB Information

Assembly-Time FDB Macro Calls	Run-Time FDB Macro Calls	File-Processing Macro Calls
FDBDF$ (Required)	FDAT$R	OPEN$ (All Variations)
FDAT$A	FDRC$R	CLOSE$
FDRC$A	FDBK$R	GET$ (All Variations)
FDBK$A	FDOP$R	PUT$ (All Variations)
FDOP$A	FDBF$R	READ$
FDBF$A		WRITE$
		DELET$
		WAIT$

2.1 .MCALL DIRECTIVE - LISTING NAMES OF REQUIRED MACRO DEFINITIONS

All the assembly-time, run-time, and file-processing macro calls (see Table 2-1 above) that the user intends to issue in his program must first be listed as arguments in an .MCALL directive. So doing allows the required macro definitions to be read in from the system macro library during assembly.

The .MCALL directive and associated arguments must appear in the program prior to the issuance of any macro call in the execution code of the program. If the list of macro names is lengthy, several .MCALL directives, each appearing on a separate source line, must be specified to accommodate the entire list of macro names. The number of such names that may appear in any given .MCALL statement is limited only by the availability of space within that 80-byte source line.

The .MCALL directive takes the following general form:

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

where: arg1, etc. represents a list of symbolic names identifying the macro definitions required in the assembly of the user program. If more than one source line is required to list the names of all desired macros, each additional line so required must begin with an .MCALL directive.

For clarity of functional use, the assembly-time, run-time, and file-processing macro names may be listed in each of three separate .MCALL statements. The macro names may also be listed alphabetically for readability, or they may be intermixed, irrespective of functional use. All these options are matters of preference and have no effect whatever on the retrieval of macro definitions from the system macro library.

For those users planning to invoke the command line processing capabilities of the Get Command Line Routine (GCML) and the Command String Interpreter (CSI), all the names of the associated macros must also be listed as arguments in an .MCALL directive. GCML and CSI, ordinarily employed in system or application programs for convenience in dynamically processing file specifications, are described in detail in Chapter 6.

The .MCALL directive is described in further detail in the IAS/RSX-11 MACRO-11 Reference Manual. The sample programs in Appendix D also illustrate the use of the .MCALL directive. Note that these directives appear as the very first statements in the preparatory coding of these programs.

The object routines described in Chapter 4 should not be confused with the macro definitions available from the system macro library. The file-control routines, constituting a body of object modules, are linked into the user program at task-build time from the system object library (SY:[1,1]SYSLIB.OLB). The reader should consult section 4.1 for a description of these routines.

The following statements are representative of the use of the .MCALL directive:

```
.MCALL  FDBDF$,FDAT$A,FDRC$A,FDOP$A,NMBLK$,FSRSZ$,FINIT$
.MCALL  OPEN$R,OPEN$W,GET$,PUT$,CLOSE$
```

2.2 FILE DESCRIPTOR BLOCK (FDB)

The file descriptor block (FDB) is the data structure that provides the information needed by FCS for all file I/O operations. Two sets of macro calls are available for FDB initialization: one set is used for assembly-time initialization (see next section), and the other set is used for run-time initialization (see section 2.2.2). Run-time macros are used to supplement and/or override information specified during assembly. Appendices A and B illustrate all the sections of the FDB in detail.

2.2.1 Assembly-Time FDB Initialization Macros

Assembly-time initialization requires that the FDBDF$ macro call be issued (see section 2.2.1.1) to allocate space for and to define the beginning address of the FDB. Additional macro calls can then be issued to establish other required information in this structure. The assembly-time macros which accomplish these functions are described in the following sections. These macro calls take the general form shown below:

```
mcnam$A  p1,p2,...,pn
```

where:

mcnam$A — represents the symbolic name of the macro.

p1,p2, ...,pn — represents the string of initialization parameters associated with the specified macro. A parameter may be omitted from the string by leaving its field between delimiting commas null. Assume, for example, that a macro call may take the following parameters:

```
FDOP$A  2,DSPT,DFNB
```

Assume further that the second parameter field is to be coded as a null specification. In this case, the statement is coded as follows:

```
FDOP$A  2,,DFNB
```

Also, a trailing comma need not be inserted to reflect the omission of a parameter beyond the last explicit specification. For example, the following macro call:

```
FDOP$A  2,DSPT,DFNB
```

need not be specified in the following manner

```
FDOP$A  2,DSPT,
```

if the last parameter (DFNB) is omitted. Rather, such a macro call is specified as follows:

```
FDOP$A  2,DSPT
```

If any parameter is not specified, i.e., if any field in the macro call contains a null specification, the corresponding cell in the FDB is not initialized and thus remains zero (0).

Multiple values may be specified in a parameter field of certain macro calls. Such values are indicated by placing an exclamation point (!) between the values, indicating a logical OR operation to the MACRO-11 assembler. The use of multiple values in this manner is pointed out in the body of this manual where such specifications apply.

Throughout the descriptions of the assembly-time macros in the following sections and elsewhere in this manual, symbols of the form F.xxx or F.xxxx are referenced (e.g., F.RTYP). These symbols are defined as offsets from the beginning address of the FDB, allowing specific locations within the FDB to be referenced. Thus, the programmer can reference or modify information within the FDB without having to calculate word or byte offsets to specific locations.

Using such symbols in system/user software also has the additional advantage of permitting the relative position of cells within the FDB to be changed (in a subsequent release, for example) without affecting the user's current programs or the coding style employed in developing new programs.

2.2.1.1 FDBDF$ - Allocate File Descriptor Block (FDB) - The FDBDF$ macro call is specified in a MACRO-11 program to allocate space within the program for a file descriptor block (FDB). This macro call must be specified in the source program once for each input or output file that is to be opened simultaneously by the user program in the course of execution. Any associated assembly-time macro calls (see sections 2.2.1.2 through 2.2.1.6) must then be specified immediately following the FDBDF$ macro if the user desires to accomplish the initialization of certain portions of this FDB during assembly.

The FDB allocation macro takes the following form:

```
label:  FDBDF$
```

where: label — represents a user-specified symbol that names this particular FDB and defines its beginning address. This label has particular significance in all I/O operations that require access to the data structure allocated through this macro call. FCS accesses the fields within the FDB relative to the address represented by this symbol.

The following examples are representative of FDBDF$ macro calls as they might appear in a source program:

```
FDBOUT: FDBDF$                  ;ALLOCATES SPACE FOR AN FDB NAMED
                                ;"FDBOUT" AND ESTABLISHES THE
                                ;BEGINNING ADDRESS OF THE FDB.

FDBIN:  FDBDF$                  ;ALLOCATES SPACE FOR AN FDB NAMED
                                ;"FDBIN" AND ESTABLISHES THE
                                ;BEGINNING ADDRESS OF THE FDB.
```

As noted earlier, the source program must embody one FDBDF$ macro call logically similar to those above for each file that is to be accessed simultaneously by the user program. FDB's can be re-used for many different files, as long as the file currently using the FDB is closed before the next file is opened. The only requirement is that an FDB must be defined for every file that is to be opened simultaneously.

2.2.1.2 FDAT$A - Initialize File Attribute Section of FDB - The FDAT$A macro call is used to initialize the file attribute section of the FDB when a new output file is to be created. If the file to be processed already exists, the FDAT$A initialization macro is not required, since FCS obtains the necessary information from the first 14 bytes of the user file attribute section of the specified file's header block (see Appendix F). This macro call has the following format:

```
FDAT$A  rtyp,ratt,rsiz,cntg,aloc
```

where:

rtyp — represents a symbolic value that defines the type of records to be built as the new file is created. Either one of two values must be specified, as follows:

R.FIX - Indicates that fixed-length records are to be written in creating the file.

R.VAR - Indicates that variable-length records are to be written in creating the file.

This parameter initializes FDB offset location F.RTYP. Since the symbols R.FIX and R.VAR initialize the same location in the FDB, these values are mutually exclusive. Either one or the other, but not both, may be specified.

ratt — represents symbolic values that may be specified to define the attributes of the records as the new file is created. The following symbolic values may be specified, as appropriate, to define the desired record attributes:

FD.FTN - Indicates that the first byte in each record is to contain a FORTRAN carriage-control character.

FD.CR - Indicates that the record is to be preceded by a <LF> character and followed by a <CR> character when the record is written to a carriage-control device, e.g., a line printer or a terminal.

FD.BLK - Indicates that records are not allowed to cross block boundaries.

These parameters initialize the record attribute byte (offset location F.RATT) in the FDB. The values FD.FTN and FD.CR are mutually exclusive and must not be specified together. Apart from this restriction, the combination (logical OR) of multiple parameters specified in this field must be separated by an exclamation point (e.g., FD.CR!FD.BLK).

rsiz — represents a numeric value that defines the size (in bytes) of fixed-length records to be written to the file. This value, which initializes FDB offset location F.RSIZ, need not be specified if R.VAR has been specified as the record type parameter above (for variable-length records). If R.VAR is specified, FCS maintains a value in FDB offset location F.RSIZ that defines the size (in bytes) of the largest record currently written to the file. Thus, whenever an existing file containing variable-length records is opened, the value in F.RSIZ defines the size of the largest record within that file. By examining the value in this cell, a program can dynamically allocate record buffers for its open files.

cntg — represents a signed numeric value that defines the number of blocks that will be allocated for the file as it is created. The signed values have the following significance:

Positive Value - Indicates that the specified number of blocks is to be allocated contiguously at file-create time, and, further, that the file is to be contiguous.

Negative Value - Indicates that the two's complement of the specified number of blocks is to be allocated at file-create time, not necessarily contiguously, and, further, that the file is to be noncontiguous.

This parameter, which has 15 bits of magnitude (plus a sign bit), initializes FDB offset location F.CNTG.

If the user has a firm idea as to the desired length of the file, it is more efficient to allocate the required number of blocks at file-create time through this parameter, rather than requiring FCS to extend the file, if necessary, during the writing of the file (see aloc parameter below).

If this parameter is not specified, then the file is created as an empty file, i.e., no space is allocated within the file as it is created.

Issuing the CLOSE$ macro call at the completion of file processing resets the value in F.CNTG to zero (0). Thus, the usual procedure is to initialize this location at run-time just before opening the file. This action is especially necessary if the FDB is to be re-used.

aloc — represents a signed numeric value that defines the number of blocks by which the file will be extended if FCS determines that file extension is necessary during the writing of the file. When the end of allocated space in the file is reached during writing, the signed value provided through this parameter causes file extension to occur, as follows:

Positive Value - Indicates that the specified number of blocks is to be allocated contiguously as additional space within the file and, further, that the file is to be noncontiguous.

Negative Value - Indicates that the two's complement of the specified number of blocks is to be allocated noncontiguously as additional space within the file and, further, that the file is to be noncontiguous.

This parameter, which also has 15 bits of magnitude (plus a sign bit), initializes FDB offset location F.ALOC. If this optional parameter is not specified, file extension occurs as follows:

1. If the number of virtual blocks yet to be written is greater than one (1), the file is extended by the exact number of blocks required to complete the writing of the file.

2. If only one additional block is required to complete the writing of the file, the file is extended in accordance with the volume's default extend value.

In IAS, RSX-11D, and RSX-11M, the volume default extend size is established through the INITIALIZE, INITVOLUME, or MOUNT command respectively. These initialization commands are described in the IAS System Management Guide, the RSX-11D User's Guide, or the RSX-11M Operator's Procedures Manual. The MOUNT command for IAS is described in the IAS User's Guide. The volume default extend size cannot be established at the FCS level; this value must be established when the volume is initially mounted.

The following statement is representative of an FDAT$A macro call. This statement initializes the FDB in preparation for the creation of a new file containing fixed-length, 80-byte records that will be allowed to cross block boundaries.

```
FDAT$A  R.FIX,,80.
```

In the above example, the record attribute (ratt) parameter has been omitted, as indicated by the second comma (,) in the parameter string. Also, the cntg and aloc parameters have been omitted. Their omission, however, occurs following the last explicit specification, and their absence need not be indicated by trailing commas in the parameter string. Since the aloc parameter has been omitted, file extension (if it becomes necessary) will be accomplished in accordance with the current default extend size in effect for the associated volume.

If more than one record attribute is specified in the ratt parameter field, such specifications must be separated by an exclamation point (!), as shown below:

```
FDAT$A  R.VAR,FD.CR!FD.BLK
```

The above macro call will enable a file of variable-length records to be created. The records will contain vertical formatting information for carriage-control devices; the records will not be allowed to cross block boundaries.

2.2.1.3 FDRC$A - Initialize Record Access Section of FDB - The FDRC$A macro call is used to initialize the record access section of the FDB and to indicate whether record or block I/O operations are to be used in processing the associated file.

If record I/O operations (GET$ and PUT$ macro calls) are to be used, the FDRC$A or the FDRC$R macro call (see section 2.2.2) establishes the FDB information necessary for record-oriented I/O. If block I/O operations (READ$ and WRITE$ macro calls) are to be used, however, the FDBK$A macro call (see section 2.2.1.4) or the FDBK$R macro call (see section 2.2.2) must also be specified in order to establish other values in the FDB required for block I/O. In this case, portions of the record access section of the FDB are physically overlaid with parameters from the FDBK$A/FDBK$R macro call.

Prior to issuing the OPEN$x macro call to initiate file operations, the FDB must be appropriately initialized to indicate whether record or block I/O operations are to be used in processing the associated file.

The FDRC$A macro call takes the following format:

```
        FDRC$A  racc,urba,urbs
```

where: racc represents symbolic values that specify how FCS is to handle file data. This parameter initializes the record access byte (offset location F.RACC) in the FDB. The first value below applies only for block I/O (READ$/WRITE$) operations; all remaining values are specific to record I/O (GET$/PUT$) operations:

FD.RWM - Indicates that READ$/WRITE$ (block I/O) operations are to be used in processing the file. If this value is not specified, GET$/PUT$ (record I/O) operations are used by default.

Specifying FD.RWM necessitates issuing an FDBK$A or an FDBK$R macro call in the program to initialize other offsets in the block access section of the FDB. Note also that the READ$ or WRITE$ macro call allows the complete specification of all the parameters required for block I/O operations.

FD.RAN - Indicates that random access mode is to be used in processing the file. If this value is not specified, sequential access mode is used by default.

FD.PLC - Indicates that locate mode is to be used in processing the file. If this value is not specified, move mode is used by default.

FD.INS - This value, which applies only for sequential files (and therefore cannot be specified jointly with the FD.RAN parameter above), indicates that a PUT$ operation performed within the body of the file shall not truncate the file.

Should the user wish to perform a PUT$ operation within the body of a file, the .POINT routine described in section 4.8.1 may be called. This routine, which permits a limited degree of random access to a file, positions the file to a user-specified byte within a virtual block in preparation for the PUT$ operation.

If FD.INS is not specified, a PUT$ operation within the file truncates the file at the point of insertion, i.e., the PUT$ operation moves the logical end-of-file (EOF) to a point just beyond the inserted record. However, no deallocation of blocks within the file occurs.

Regardless of the setting of the FD.INS bit, a PUT$ operation that is in fact beyond the current logical end of the file will reset the logical end of the file to a point just beyond the inserted record.

urba — represents the symbolic address of a user record buffer that is to be used for GET$ operations in move and locate modes, and for PUT$ operations in locate mode. This parameter initializes FDB offset location F.URBD+2.

urbs — represents a numeric value that defines the size (in bytes) of the user record buffer to be employed for GET$ operations in move and locate modes, and for PUT$ operations in locate mode. This parameter initializes FDB offset location F.URBD.

The user allocates and labels a record buffer in his program through a .BLKB or .BLKW directive. The address and the size of this area is then passed to FCS as the urba and the urbs parameters above. For example, a user record buffer may be defined through a statement that is logically equivalent to that shown below:

```
RECBUF: .BLKB    82.
```

where "RECBUF" is the address of the buffer and 82(10) is its size (in bytes).

Under certain conditions, the user need not allocate a record buffer or specify the buffer descriptors (urba and urbs) for GET$ or PUT$ operations. These conditions are described in detail in sections 3.9.2 and 3.12.2, respectively.

The following statement is representative of an FDRC$A macro call that is issued for a file that may be accessed in random mode:

```
        FDRC$A  FD.RAN,BUF1,160.
```

The address of the user record buffer is specified through the symbol BUF1, and the size of the user record buffer (in bytes) is defined by the numeric value 160(10).

If more than one value is specified in the record access (racc) field, multiple values must be separated by an exclamation point (!), as shown below:

```
        FDRC$A  FD.RAN!FD.PLC,BUF1,160.
```

In addition to the functions described for the first example, this example specifies that locate mode is to be used in processing the associated file. Note that the multiple parameters specified in the first field are separated by an exclamation point (!).

2.2.1.4 FDBK$A - Initialize Block Access Section of FDB - The FDBK$A macro call is used to initialize the block access section of the FDB when block I/O operations (READ$ and WRITE$ macro calls) are to be used for file processing. Initializing the FDB with this macro call allows the user to read or write virtual blocks of data within a file.

As noted in the preceding section, issuing the FDBK$A macro call implies that the FDRC$A macro call has also been specified, since it is through the FD.RWM parameter of the FDRC$A macro call that the initial declaration of block I/O operations is accomplished. Thus, for block I/O operations, the FDRC$A macro call must be specified, as well as any one of the following macro calls, to appropriately initialize the block access section of the FDB: FDBK$A, FDBK$R, READ$, or WRITE$.

Issuing the FDBK$A macro call causes certain portions of the record access section of the FDB to be overlaid with parameters necessary for block I/O operations. Thus, the terms "record access section" and "block access section" refer to a shared physical area of the FDB which is functional for either record or block I/O operations.

When block I/O operations are desired, the FDB must be properly initialized through the FDBK$A or the FDBK$R macro call prior to issuing a generalized OPEN$x macro call which references that FDB. If record I/O operations are to be employed, the FDBK$A or the FDBK$R macro call must not be issued.

The FDBK$A macro call is specified in the following format:

```
FDBK$A  bkda,bkds,bkvb,bkef,bkst,bkdn
```

where:

bkda — represents the symbolic address of an area in user memory space that is to be employed as a buffer for block I/O operations. This parameter initializes FDB offset location F.BKDS+2.

bkds — represents a numeric value that specifies the size (in bytes) of the block to be read or written when a block I/O request (READ$ or WRITE$ macro call) is issued. This parameter initializes FDB offset location F.BKDS. The maximum block size that can be specified through this parameter is equal to one virtual block, i.e., 512(10) bytes.

bkvb — represents a dummy parameter for compatibility with the FDBK$R macro call. The bkvb parameter is not specified in the FDBK$A macro call for the reasons stated in Item 4 of section 2.2.2.1. In short, assembly-time initialization of FDB offset locations F.BKVB+2 and F.BKVB with the virtual block number is meaningless, since any version of the generalized OPEN$x macro call resets the virtual block number in these cells to one (1) as the file is opened. Therefore, these cells can be initialized only at run-time through either the FDBK$R macro call (see section 2.2.2) or the I/O-initiating READ$ and WRITE$ macro calls (see sections 3.15 and 3.16, respectively).

This dummy parameter need be reflected as a null specification (with a comma) in the parameter string only in the event that an explicit parameter follows. This null specification is required in order to maintain the proper positionality of any remaining field(s) in the parameter string.

bkef — represents a numeric value that specifies an event flag to be used during READ$/WRITE$ operations to indicate the completion of a block I/O transfer. This parameter initializes FDB offset location F.BKEF; if not specified, event flag 32(10) is used by default.

The function of an event flag is described in further detail in section 2.8.1.

bkst — represents the symbolic address of a 2-word I/O status block in the user program. If specified, this optional parameter initializes FDB offset location F.BKST.

The I/O status block, if it is to be used, must be defined and appropriately labeled at assembly-time. Then, if the bkst parameter is specified, information is returned by the system to the I/O status block at the completion of the block I/O transfer. This information reflects the status of the requested operation. If this parameter is not specified, no information is returned to the I/O status block.

If an error condition occurs during a READ$ or WRITE$ operation that would normally be reported as a negative value in the first byte of the I/O status block, then this occurrence is not reported unless an I/O status block address is specified. Thus, the user is advised to specify this parameter to allow the return of block I/O status information and to facilitate normal error reporting.

The creation and function of the I/O status block are described in greater detail in section 2.8.2.

bkdn — represents the symbolic address of an optional user-coded AST service routine. If present, this parameter causes the AST service routine to be initiated at the specified address upon completion of block I/O; if not specified, no AST trap occurs. This parameter initializes FDB offset location F.BKDN.

Considerations relevant to the use of an AST service routine are presented in section 2.8.3.

The following example shows an FDBK$A macro call which utilizes all available parameter fields for initializing the block access section of the FDB:

```
FDBK$A  BKBUF,240.,,20.,ISTAT,ASTADR
```

In this macro call, the symbol BKBUF identifies a block I/O buffer reserved in the user program that will accommodate a 240(10)-byte block. The virtual block number is null (for the reasons stated in the description of this parameter above), and the event flag to be set upon block I/O completion is 20(10). Finally, the symbol ISTAT specifies the address of the I/O status block, and the symbol ASTADR specifies the entry-point address of the AST service routine.

2.2.1.5 FDOP$A - Initialize File Open Section of FDB - The FDOP$A macro call is used to initialize the file-open section of the FDB. In addition to a logical unit number, either a dataset descriptor pointer and/or a default filename block address is normally specified for each file that is to be opened. The latter two parameters provide FCS with the linkage necessary to retrieve file specifications from these user-created data structures in the program.

Although both a dataset descriptor pointer (dspt) and the address of a default filename block (dfnb) may be specified for a given file, one or the other must be present in the FDB before that file can be opened. If, however, certain information is already present in the filename block as the result of prior program action, neither the dataset descriptor nor the default filename block is accessed by FCS, and the file is opened through a process called "opening a file by file ID." This process, which is a very efficient method of opening a file, is described in detail in section 2.5.

The dspt and dfnb parameters represent address values which point to user-defined data structures in the program. These data structures, which are described in detail in section 2.4, provide file specifications to the FCS file-processing routines.

The FDOP$A macro call takes the following form:

```
FDOP$A  lun,dspt,dfnb,facc,actl(1)
```

where:

lun
: represents a numeric value which specifies a logical unit number. This parameter initializes FDB offset location F.LUN. All I/O operations performed in conjunction with this FDB are done through the specified logical unit number (LUN). Every active FDB must have a unique LUN.

  The logical unit number specified through this parameter may be any value from one (1) through the largest value specified to the Task Builder through the UNITS directive. This directive specifies the number of logical units to be used by the task (see the Task Builder Reference Manual of the host operating system).

dspt
: represents the symbolic address of a 6-word block in the user program containing the dataset descriptor. This user-defined data structure consists of a 2-word device descriptor, a 2-word directory descriptor, and a 2-word filename descriptor, as outlined in section 2.4.1.

  The dspt parameter initializes FDB offset location F.DSPT. This address value, called the dataset descriptor pointer, is the linkage address through which FCS accesses the fields in the dataset descriptor.

---

(1) The actl parameter does not apply to RSX-11M.

When the Command String Interpreter (CSI) is used to process command string input, a file specification is returned to the calling program in a format identical to that of the manually-created dataset descriptor. The use of CSI as a dynamic command line processor is described in detail in section 6.2.

dfnb represents the symbolic address of the default filename block. This structure is allocated within the user program through the NMBLK$ macro call (see section 2.4.2). When specified, the dfnb parameter initializes FDB offset location F.DFNB, allowing FCS to access the fields of the default filename block in building the filename block in the FDB.

Specifying the dfnb parameter in the FDOP$A (or the FDOP$R) macro call assumes that the NMBLK$ macro call has been issued in the program. Furthermore, the symbol specified as the dfnb parameter in the FDOP$A (or the FDOP$R) macro call must correspond exactly to the symbol specified in the label field of the NMBLK$ macro call.

facc represents any one or any appropriate combination of the following symbolic values indicating how the specified file is to be accessed:

FO.RD - Indicates that an existing file is to be opened for reading only.

FO.WRT - Indicates that a new file is to be created and opened for writing.

FO.APD - Indicates that an existing file is to be opened for append.

FO.MFY - Indicates that an existing file is to be opened for modification.

FO.UPD - Indicates that an existing file is to be opened for update and, if necessary, extended.

FA.NSP - Indicates, in combination with FO.WRT, above, that an old file having the same file specification is not be to superseded by the new file.

FA.TMP - Indicates, in combination with FO.WRT above, that the created file is to be a temporary file.

FA.SHR - Indicates that the file is to be opened for shared access.

The facc parameter initializes FDB offset location F.FACC. The symbolic values FO.xxx, described above, represent the logical or of bits in FDB location F.FACC.

The information specified by this parameter can be overridden by an OPEN$ macro call, as described in Section 3.7. It is overridden by an OPEN$x macro call.

actl applies only to IAS and RSX-11D and represents a symbolic value that is used to specify the following control information in FDB location F.ACTL:

1. Magnetic tape position,

2. Whether a disk file that is opened for write is to be locked if it is not properly closed, e.g., the task terminates abnormally,

3. Number of retrieval pointers to allocate for a disk file window.

Normallly, FCS supplies default values for F.ACTL. However, if FA.ENB is specified in combination with any of the symbolic values described below, FCS uses the information in F.ACTL. FA.ENB must be specified with the desired values to override the defaults. The following are the defaults for location F.ACTL.

For file creation, magnetic tapes are positioned to the end of the volume set.

At file open and close, tapes are not rewound.

A disk file that is opened for write is locked if it is not properly closed.

The volume default is used for the file window.

The values listed below can be used in conjunction with FA.ENB.

FA.POS - Is meaningful only for output files and is specified to cause a magnetic tape to be positioned just after the most recently closed file for the creation of a new file. Any files that exist after that point are lost. If rewind is specified, it takes precedence over FA.POS, thus causing the tape to be positioned just after the VOL1 label for file creation. See Section 5.2.3.

FA.RWD - Is specified to cause a magnetic tape to be rewound when the file is opened or closed.

Examples of the use of FA.ENB with FA.POS and FA.RWD are provided in Section 5.2.8.

FA.DLK - Is specified to cause a disk file not to be locked if it is not properly closed.

The number of retrieval pointers for a file window can be specified in the low-order byte of F.ACTL.

> The system normally provides 7 retrieval pointers automatically. Retrieval pointers are used to point to contiguous blocks of the file on disk. Access to fragmented files may be optimized by increasing the number of retrieval pointers, i.e., by increasing the size of the window. Likewise, additional memory can be freed by reducing the number of pointers for files with little or no fragmentation, e.g., contiguous files.

As noted, if neither the dspt nor the dfnb parameter is specified, corresponding offset locations F.DSPT and F.DFNB contain zero (0). In this case, no file is currently associated with this FDB. Any attempt to open a file with this FDB will result in an open failure. Either offset location F.DSPT or F.DFNB must be initialized with an appropriate address value before a file can be opened using this FDB. Normally, these cells are initialized at assembly-time through the FDOP$A macro call; they may also be initialized at run-time through the FDOP$R or the generalized OPEN$x macro call (see section 3.1).

The following examples are representative of the FDOP$A macro call as it might appear in the source program:

```
FDOP$A  1,,DFNB

FDOP$A  2,OFDSPT

FDOP$A  2,OFDSPT,DFNB

FDOP$A  1,CSIBLK+C.DSDS

FDOP$A  1,,DFNB,,FA.ENB!16.(1)
```

Note in the first example that the dataset descriptor pointer (dspt) is null, requiring that FCS rely on the run-time specification of the dataset descriptor pointer for the FDB or the use of the default filename block for required file information.

In the second example, a dataset descriptor pointer (OFDSPT) has been specified, allowing FCS to access the fields in the dataset descriptor for required file information.

The third example specifies both a dataset descriptor pointer and a default filename block address, causing FDB offset locations F.DSPT and F.DFNB, respectively, to be initialized with the appropriate values. In this case, FCS can access the dataset descriptor and/or the default filename block for required file information. By convention, FCS first seeks such information in the dataset descriptor; if all the required information is not present in this data structure, FCS attempts to obtain the missing information from the default filename block.

The fourth example shows a macro call which takes as its second parameter a symbolic value which causes FDB offset location F.DSPT to be initialized with the address of the CSI dataset descriptor. This structure is created in the CSI control block through the invocation of the CSI$ macro call. All considerations relevant to the use of CSI as a dynamic command line processor are presented in section 6.2.

---

(1) This example does not apply to RSX-11M.

The last example illustrates the use of the parameter actl to increase the number of retrieval pointers in the file window to 16. FA.ENB is specified to cause the contents of F.ACTL, rather than the defaults, to be used.

In all the examples above, the value specified as the first parameter supplies the logical unit number to be used for all I/O operations involving the associated file.

2.2.1.6 FDBF$A - Initialize Block Buffer Section of FDB - The FDBF$A macro call is used to initialize the block buffer section of the FDB when record I/O operations (GET$ and PUT$ macro calls) are to be used for file processing. Initializing the FDB with this macro call allows FCS to control the necessary blocking and deblocking of individual records within a virtual block as an integral function of processing the file.

The FDBF$A macro call takes the following format:

```
FDBF$A  efn,ovbs,mbct,mbfg
```

where:

efn
: represents a numeric value which specifies the event flag to be used by FCS in synchronizing record I/O operations. This numeric value initializes FDB offset location F.EFN. This event flag is used internally by FCS; it must not be set, cleared, or tested by the user.

  If this parameter is not specified, event flag 32(10) is used by default. A null specification in this field is indicated by inserting a leading comma in the parameter string.

ovbs
: represents a numeric value which specifies an FSR block buffer size (in bytes) which overrides the standard block size for the particular device associated with the file. This parameter is specified only when a non-standard block size is desired. The numeric value so specified initializes FDB offset location F.OVBS.

  An override block size is allowed only for record-oriented devices (such as line printers) and sequential devices (such as magnetic tape units). For block-oriented devices, the override block size is ignored. In IAS and RSX-11D, for spooled output to a record-oriented device, a buffer less than 512(10) bytes in length must not be allocated.

  Issuing the CLOSE$ macro call (see section 3.8) resets offset location F.OVBS in the associated FDB to zero (0). Therefore, this location should typically be initialized at run-time just before opening the file, particularly if an OPEN$x/CLOSE$ sequence for the file is performed more than once.

  The standard block size in effect for a particular device may be obtained through an I/O-related system directive called Get Lun Information (GLUN$). This directive is described in detail in the Executive Reference Manual of the host operating system. The standard block size for a device is established at system-generation time.

mbct
: represents a numeric value which specifies the multiple buffer count, i.e., the number of buffers to be used by FCS in processing the associated

file. This parameter initializes FDB offset location F.MBCT. If this value is greater than one (1), multiple buffering is effectively declared for file processing. In this case, FCS will employ either read-ahead or write-behind operations, depending on which of two symbolic values is specified as the mbfg parameter (see below).

If the mbct parameter is specified as null or zero (0), FCS uses the default buffer count contained in symbolic location .MBFCT in $$FSR2 (the program section in the FSR containing impure data). This cell normally contains a default buffer count of one (1). If desired, this value can be modified, as noted in the discussion following the mbfg parameter below.

If, in specifying the FSRSZ$ macro call (see section 2.6.1), sufficient memory space has not been allocated to accommodate the number of buffers established by the mbct parameter, FCS allocates as many buffers as will fit in the available space. Insufficient space for at least one buffer causes FCS to return an error code to FDB offset location F.ERR.

The user can initialize the buffer count in F.MBCT through either the FDBF$A or the FDBF$R macro call. The buffer count so established is not altered by FCS and, once set, need not be of further concern to the user.

mbfg represents a symbolic value that specifies the type of multiple buffering to be employed in processing the file. Either of two values may be specified to initialize FDB offset location F.MBFG:

FD.RAH - Indicates that read-ahead operations are to be used in processing the file.

FD.WBH - Indicates that write-behind operations are to be used in processing the file.

These parameters are mutually exclusive, i.e., one or the other, but not both, may be specified.

Specifying this parameter assumes that the buffer count established in the mbct parameter above is greater than one (1). If multiple buffering has thus been declared,, the omission of the mbfg parameter causes FCS to use read-ahead operations by default for all files opened using the OPEN$R macro call; similarly, write-behind operations are used by default for all files opened using other forms of the OPEN$x macro call.

If these default buffering conventions are not desired, the user can alter the value in the F.MBFG dynamically at run-time. This is done by issuing the FDBF$R macro call, which takes as the mbfg parameter the appropriate control flag (FD.RAH or FD.WBH). This action must be taken, however, before opening the file.

Offset location F.MBFG in the FDB is reset to zero (0) each time the associated file is closed.

NOTE

For RSX-11M, the normally released version of FCS uses single buffering and simply ignores the multiple-buffering parameters (mbct and mbfg) in the FDBK$A/FDBK$R macro call. A multiple-buffered version of FCS is available in the library file SY:[1,1]DBLBUFLIB.OLB. Thus, for multiple buffering, the system must contain the appropriate routines in a resident library or the user must link his program with the DBLBUFLIB object library file.

For IAS and RSX-11D, resident and nonresident libraries support the multibuffered version of FCS.

As noted in the description of the mbct parameter above, the default buffer count can be changed, if desired, by modifying a location in $$FSR2, the second of two program sections comprising the FSR. A location defined as .MBFCT in $$FSR2 normally contains a default buffer count of one (1). This default value may be changed, as follows:

1. Apply a global patch to .MBFCT at task-build time to specify the desired number of buffers.

2. For MACRO-11 programs, use the EXTSCT Task Builder directive (see section 2.7.1) to allocate more space for the FSR block buffers; for FORTRAN programs, use the ACTFIL Task Builder directive (see section 2.7.2) to allocate more space for the FSR block buffers.

Because the above procedure alters the default buffer count for all files to be processed by the user program, it may be desirable to force single buffering for any specific file(s) that would not benefit from multiple buffering. In such a case, the buffer count in F.MBCT for a specific file may be set to one (1) by issuing the following macro call for the applicable FDB:

```
        FDBF$A  ,,1
```

The value "1" specifies the buffer count (mbct) for the desired file and is entered into offset location F.MBCT in the applicable FDB. Note in the example above that the event flag (efn) and the override block buffer size (ovbs) parameters are null; these null values are used for illustrative purposes only and should not be interpreted as conditional specifications for establishing single-buffered operations.

The following examples are representative of the FDBF$A macro call as it might appear in a program:

```
        FDBF$A  25.,,1

        FDBF$A  25.,,2,FD.RAH

        FDBF$A  ,,2,FD.WBH
```

The first example specifies that event flag 25(10) is to be used in synchronizing record I/O operations and that single buffering is to be used in processing the file.

The second example also specifies event flag 25(10) for synchronizing record I/O operations and, in addition, establishes "2" as the multiple buffer count. The buffers so specified are to be used for read-ahead operations, as indicated by the final parameter.

The last example allows event flag 32(10) to be used by default for synchronizing record I/O operations, and the two buffers specified in this case are to be used for write-behind operations.

Note in all three examples that the second parameter, i.e., the override block size parameter (ovbs), is null; thus, the standard block size in effect for the device in question will be used for all file I/O operations.

2.2.2 Run-Time FDB Initialization Macros

Although the FDB is allocated and can be initialized during program assembly, the contents of specific sections of the FDB can also be initialized or changed at run time by issuing any of the following macro calls:

FDAT$R - Initializes or alters the file attribute section of the FDB.

FDRC$R - Initializes or alters the record access section of the FDB.

FDBK$R - Initializes or alters the block access section of the FDB (see Item 4 below).

FDOP$R - Initializes or alters the file-open section of the FDB.

FDBF$R - Initializes or alters the block buffer section of the FDB.

2.2.2.1 Run-Time FDB Macro Call Exceptions - The format and the parameters of the run-time FDB initialization macros are identical to the assembly-time macros described earlier, except as noted below:

1. An R must appear as the last character in the run-time symbolic macro name, rather than an A.

2. The first parameter in all run-time macro calls must be the address of the FDB associated with the file to be processed. All other parameters in the run-time macro calls are identical to those described in sections 2.2.1.2 through 2.2.1.6 for the assembly-time macro calls, except as noted in Items 3 and 4 below.

3. The parameters in the run-time macro calls must be valid MACRO-11 source operand expressions. These parameters may be address values or literal values; they may also represent the contents of registers or memory locations. In short, any value that is a valid source operand in a MOV or MOVB instruction may be specified in a run-time macro call. In this regard, the following conventions apply:

   a. If the parameter is an address value or a literal value that is to be placed in the FDB, i.e., if the parameter itself is to be taken as an argument, it must be preceded by the number sign (#). This symbol is the immediate expression indicator for MACRO-11 programs, causing the associated argument to be taken literally in initializing the appropriate cell in the FDB. Such literal values may be specified as follows:

```
FDOP$R  #FDBADR,#1,#DSPT,#DFNB
```

b. If the parameter is the address of a location containing an argument that is to be placed in the FDB, the parameter must not be preceded by the number sign (#). Such a parameter may be specified, as follows:

```
ONE:    .WORD   1
          .
          .
          .
        FDOP$R  #FDBADR,ONE,#DSPT,#DFNB
```

where "ONE" represents the symbolic address of a location containing the desired initializing value.

c. Also, if the parameter is a register specifier (e.g., R0), the parameter must not be preceded by the number sign (#). Register specifiers are defined MACRO-11 symbols and are valid expressions in any context.

Thus, in contrast, parameters specified in assembly-time macro calls are used as arguments in generating data in .WORD or .BYTE directives, while parameters specified in run-time macro calls are used as arguments in MOV and MOVB machine instructions.

4. As noted in the description of the FDBK$A macro call in section 2.2.1.4, assembly-time initialization of the FDB with the virtual block number is meaningless, since issuing the OPEN$x macro call to prepare a file for processing automatically resets the virtual block number in the FDB to one (1). For this reason, the virtual block number can be specified only at run-time after the file has been opened. This may be accomplished through either the FDBK$R macro call or the I/O-initiating READ$/WRITE$ macro call. In all three cases, the relevant field for defining the virtual block number is the bkvb parameter. The READ$ and WRITE$ macro calls are described in detail in sections 3.15 and 3.16, respectively.

At assembly-time, the user must reserve and label a 2-word block in the program which is to be used for temporarily storing the virtual block number appropriate for intended block I/O operations. Since the user is free to manipulate the contents of these two locations at will, any virtual block number consistent with intended block I/O operations may be defined. By specifying the symbolic address (i.e., the label) of this field as the bkvb parameter in the selected run-time macro call, the virtual block number is made available to FCS.

In preparing for block I/O operations, the following general procedures must be performed:

a. At assembly-time, reserve a 2-word block in the user program through a statement that is logically equivalent to the following:

```
VBNADR: .BLKW   2
```

The label "VBNADR" names this 2-word block and defines its address. This symbol is used subsequently as the bkvb parameter in the selected run-time macro call for initializing the FDB.

b. At run-time, load this field with the desired virtual block number. This operation may be accomplished through statements logically equivalent to those shown below:

```
CLR     VBNADR
MOV     #10400,VBNADR+2
```

Note that the first word of the block is cleared. The MOV instruction then loads the second (low-order) word of the block with a numeric value. This value constitutes the 16 least significant bits of the virtual block number.

If the desired virtual block number cannot be completely expressed within 16 bits, the remaining portion of the virtual block number must be stored in the first (high-order) word of the block. This may be accomplished through statements logically equivalent to the following:

```
MOV     #1,VBNADR
MOV     #10400,VBNADR+2
```

As a result of these two instructions, 31 bits of value are defined in this 2-word block. The first word contains the 15 most significant bits of the virtual block number, and the second word contains the 16 least significant bits. Thus, the virtual block number is an unsigned value having 31 bits of magnitude. The user must ensure that the sign bit in the high-order word is not set.

c. Open the desired file for processing by issuing the appropriate version of the generalized OPEN$x macro call (see section 3.1).

d. Issue either the FDBK$R macro call or the READ$/WRITE$ macro call, as appropriate, to initialize the relevant FDB with the desired virtual block number.

If the FDBK$R macro call is elected, the following is a representative example:

```
FDBK$R  #FDBIN,,,#VBNADR
```

Regardless of the particular macro call used to supply the virtual block number, the two words at VBNADR are loaded into F.BKVB and F.BKVB+2. The first of these words (F.BKVB) is zero (0) if 16 bits is sufficient to express the desired virtual block number. The I/O-initiating READ$/WRITE$ macro call may then be issued.

Should the user, however, choose to initialize the FDB directly through either the READ$ or WRITE$ macro call, the virtual block number may be made available to FCS through a statement such as that shown below:

```
READ$   #FDBIN,#INBUF,#BUFSIZ,#VBNADR
```

where the symbol "VBNADR" represents the address of the 2-word block in the user program containing the virtual block number.

2.2.2.2 Specifying the FDB Address in Run-Time Macro Calls - In relation to Item 2 of the exceptions noted above, the address of the FDB associated with the file to be processed corresponds to the address value of the user-defined symbol appearing in the label field of the FDBDF$ macro call (see section 2.2.1.1). For example, the following statement:

```
FDBOUT: FDBDF$
```

in addition to allocating space for an FDB at assembly time, binds the label "FDBOUT" to the beginning address of the FDB associated with this file. The address value so established can then be specified as the initial parameter in a run-time macro call in any one of three ways, as follows:

1. The address of the appropriate FDB may be specified as an explicit parameter in a run-time macro call, as indicated in the following statement:

   ```
   FDAT$R  #FDBOUT,#R.VAR,#FD.CR
   ```

   The argument "FDBOUT" is taken literally by FCS as the address of an FDB; furthermore, this address value, by convention, is stored in general register zero (R0). Whenever this method of specifying the FDB address is employed, the previous contents of R0 are overwritten (and thus destroyed). Therefore, the user must exercise care in issuing subsequent run-time macro calls to ensure that the present value of R0 is suitable to current purposes.

2. The general register specifier "R0" may be used as the initial parameter in a run-time macro call, as reflected in the following statement:

```
FDAT$R  R0,#R.VAR,#FD.CR
```

   In this case, the current contents of R0 are taken by FCS as the address of the appropriate FDB. This method assumes that the address of the FDB has been previously loaded into R0 through some overt action. Note, when using this method to specify the FDB address, that the immediate expression indicator (#) must not precede the register specifier (R0).

3. A null specification may also be used as the initial parameter in a run-time macro call, as shown below:

```
FDAT$R  ,#R.VAR,#FD.CR
```

   In this instance, the current contents of R0 are taken by default as the address of the associated FDB. As in method 2 above, R0 is assumed to contain the address of the desired FDB. Although the comma in this instance constitutes a valid specification, the user is advised to employ methods 1 and 2 for consistency and clarity of purpose.

In relation to the foregoing, it should be understood that these three methods of specifying the FDB address also apply to all the FCS file-processing macro calls described in Chapter 3.

2.3 GLOBAL VERSUS LOCAL DEFINITIONS FOR FDB OFFSETS

Although the FDB offsets can be defined either locally or globally, it was fully intended in the design of FCS that the user need not necessarily be concerned with the definition of FDB offsets locally. To some extent, this design consideration was based on the manner in which MACRO-11 handles symbols,

Whenever a symbol appears in the source program, MACRO-11 automatically assumes that it is a global symbol if it is not presently defined within the current assembly. Such a symbol must be defined further on in the program; otherwise, it will be treated by MACRO-11 as a default global reference, requiring that it be resolved by the Task Builder.

Thus, the question of global versus local symbols may simply be a matter of the programmer not defining the FDB offsets and bit values locally as he codes the program. Such undefined symbols thus become global references which are reduced to absolute definitions at task-build time.

Other considerations, however, also apply to the use of global or local offsets and involve some trade-off analysis. For example, if symbols are defined locally within the source program, sufficient symbol table space may not be available at assembly-time. On the other hand, if the programmer allows the symbols to become global by default because they are not defined within the source program, the available symbol table space may then be insufficient at task-build time. (Task Builder symbol table overflow is unlikely. However, defining the offsets globally will increase link time.) If, however, sufficient symbol table space is available for both MACRO-11 and the Task Builder, the burden of symbol table space will fall where appropriate. In either case, the symbols are handled properly whether they be local or global.

The only instance in which this question takes on operational significance is when symbol table overflow problems are experienced with either MACRO-11 or the Task Builder. In this case, program size constraints dictate more careful programming. Depending on whether MACRO-11 or the Task Builder is experiencing the overflow problems, FDB offsets and bit values may be allowed to become global by defut, or they may be defined locally in the source program through the invocation of the FDOF$L and FCSBT$ macro calls (see section 2.3.2).

If the symbol table overflow problem is present at both assembly-time and task-build time, the user must reduce the size of the source modules so that they can be processed without difficulty.

It should be noted that global symbols may be used as operands and/or macro call parameters anywhere in the source program coding, as described in the following section.

2.3.1 Specifying Global Symbols in the Source Coding

Throughout the descriptions of the assembly-time macros (see sections 2.2.1.2 through 2.2.1.6), global symbols are specified as parameters in the macro calls. As noted earlier, such symbols are treated by MACRO-11 as default global references.

For example, the global symbol FD.RAN may be specified as the initial parameter in the FDRC$A macro call (see section 2.2.1.3). At task-build time, this parameter is reduced to an absolute symbol definition, causing a prescribed bit to be set in the record access byte (offset location F.RACC) of the FDB.

Global symbols may also be used as operands in user program instructions to accomplish operations associated with FDB offset locations. For example, global offsets such as F.RACC, F.RSIZ, and F.RTYP may be specified as operands in the source coding. Assume, for example, that an FDBDF$ macro call (see section 2.2.1.1) has been issued in the source program to allocate space for an FDB, as follows:

```
FDBIN:  FDBDF$
```

The coding sequence below may then appear in the source program, illustrating the use of the global offset F.RACC:

```
MOV     #FDBIN,R0
MOVB    #FD.RAN,F.RACC(R0)
```

Note that the beginning address of the FDB is first moved into general register zero (R0). However, if the desired value already exists in R0 as the result of previous action in the program, the user need issue only the second MOV instruction (which appropriately references R0). As a consequence of this instruction, the value FD.RAN initializes FDB offset location F.RACC.

An equivalent instruction is the following:

```
MOVB    #FD.RAN,FDBIN+F.RACC
```

which likewise initializes offset location F.RACC in the FDB with the value of FD.RAN. Global symbols may be used anywhere in the program in this manner to effect the dynamic storage of values within the FDB.

2.3.2 Defining FDB Offsets and Bit Values Locally

Should the user wish to declare explicitly that all FDB offsets and bit values are to be defined locally, he may do so by invoking two macro calls in the source program. The first of these, FDOF$L, causes the offsets for FDB's to be defined within the user program. Similarly, bit values for all FDB parameters may be defined locally by invoking the FCSBT$ macro call. These macro calls may be invoked anywhere in the user program.

When issued, the FDOF$L and FCSBT$ macro calls define symbols in a manner that is roughly equivalent to that shown below:

```
F.RTYP = xxxx
F.RACC = xxxx
F.RSIZ = xxxx
```

where "xxxx" represents the value assigned to the corresponding symbol.

In other words, the macros for defining FDB offsets and bit values locally do not generate any code. Their function is simply to create absolute symbol definitions within the program at assembly-time. The symbols so defined, however, appear in the MACRO-11 symbol table, rather than in the source program listing. Such local symbol definitions are thereby made available to MACRO-11 during assembly, rather than forcing them to be resolved by the Task Builder.

Whether or not the FDOF$L and FCSBT$ macro calls are invoked should not in any way affect the coding style or the manner in which the FDB offsets and bit values are used.

Note, however, if the FDOF$L macro call is issued, that the NBOF$L macro call for the local definition of the filename block need not be issued (see section 2.4.2). The FDOF$L macro call automatically defines all FDB offsets locally, including those for the filename block.

If any of the above named macro calls is to be issued in the user program, it must first be listed as an argument in an .MCALL directive (see section 2.1).

2.4 CREATING FILE SPECIFICATIONS WITHIN THE USER PROGRAM

Certain information describing the file must be present in the FDB before the file can be opened. The file is located using a file specification which contains the following:

1. A device name and unit number;
2. A directory string consisting of a group number and a member number that specifies the user file directory (UFD) to be used for the file. The term "UFD" is synonymous with the term "file directory string" appearing throughout this manual.
3. A filename;
4. A file type (RSX-11) or file extension (IAS);
5. A file version number.

The term "file specifier" is sometimes used as a synonym for "file specification."

A file specification describing the file to be processed is communicated to FCS through two user-created data structures:

1. The Dataset Descriptor. This tabular structure may be created and initialized manually through the use of .WORD directives. Section 2.4.1 describes this data structure in detail.
2. The Default Filename Block. In contrast to the manually-created dataset descriptor, the default filename block is created by issuing the NMBLK$ macro call. This macro call allocates a block of storage in the user program at assembly-time and initializes this structure with parameters supplied in the call. This structure is described in detail in section 2.4.2.

As noted in section 2.2.1.5, the FDOP$A or the FDOP$R macro call is issued to initialize the FDB with the addresses of these data structures. These address values are supplied to FCS through the "dspt" and "dfnb" parameters of the selected macro call. FCS uses these addresses to access the fields of the dataset descriptor and/or the default filename block for the file specification required in opening a specified file.

By convention, a required file specification is first sought by FCS in the dataset descriptor. Any non-null data contained therein is translated from ASCII to Radix-50 form and stored in the appropriate offsets of the filename block. This area of the FDB then serves as the execution-time repository for the information describing the file to be opened and processed. If the dataset descriptor does not contain the required information, FCS attempts to obtain the missing information from the default filename block. If neither of these structures contains the required information, an open failure occurs.

Note, however, that the device name and the unit number need not be specified in either the dataset descriptor or the default filename block, since these values are defaulted to those applicable to the system device (SY0:) if not explicitly specified.

The FCS file-processing macro calls used in opening files are described in Chapter 3, beginning with the generalized OPEN$x macro call in section 3.1.

For a detailed description of the format and content of the filename block, the reader should refer to Appendix B.

2.4.1 Dataset Descriptor

The dataset descriptor is often oriented toward the use of a fixed (built-in) filename in the user program. A given application program, for example, may require access only to a limited and non-variable number of files throughout its execution. By defining the names of these files at assembly-time through the dataset descriptor mechanism, such a program, once initiated, will execute to completion without requiring additional file specifications.

This structure, a 6-word block of storage which may be created manually within the user program through the use of .WORD directives, contains information describing a file that the user intends to open during the course of program execution. In creating this structure, any one or all of three possible string descriptors may be defined for a particular file, as follows:

1. A 2-word descriptor for an ASCII device name string;
2. A 2-word descriptor for an ASCII file directory string; and/or
3. A 2-word descriptor for an ASCII filename string.

This data structure is allocated in the user program in the following format:

DEVICE NAME STRING DESCRIPTOR -

Word 1 - Contains the length (in bytes) of the ASCII device name string.

This string consists of a 2-character alphabetic device name, followed by an optional 1- or 2-digit octal unit number. These strings may be created through statements such as those below:

```
DEVNM: .ASCII  /DK0:/

DEVNM: .ASCII  /TT10:/
```

Word 2 - Contains the address of the ASCII device name string.

DIRECTORY STRING DESCRIPTOR -

Word 3 - Contains the length (in bytes) of the ASCII file directory string.

This string consists of a group number and a member number, separated by a comma (,). The entire string is enclosed in brackets. For example, [200,200] is a directory string. A directory string can be created through statements such as those that follow:

```
DIRNM: .ASCII  /[200,200]/

DIRNM: .ASCII  /[40,100]/
```

If the user wishes to specify an explicit file directory different from the UIC under which he is currently running, the dataset descriptor mechanism permits that flexibility.

Word 4 - Contains the address of the ASCII file directory string.

FILENAME STRING DESCRIPTOR -

Word 5 - Contains the length (in bytes) of the ASCII filename string.

This string consists of a filename up to nine characters in length, an optional 3-character file type designator, and an optional file version number. The filename and file type must be separated by a dot (.), and the file version number must be preceded by a semicolon. A filename string may be created as shown below:

```
FILNM: .ASCII  /PROG1.OBJ;7/
```

Only the characters A through Z and 0 through 9 may be used in composing an ASCII filename string.

Word 6 - Contains the address of the ASCII filename string.

A length specification of zero (0) in word 1, 3, or 5 of the dataset descriptor indicates that the corresponding device name, directory, or filename string is not present in the user program. For example, the coding below creates a dataset descriptor containing only a 2-word ASCII filename string descriptor:

```
FDBOUT: FDBDF$                       ;CREATES FDB.
        FDATSA  R.VAR,FD.CR          ;INITIALIZES FILE ATTRIBUTE SECTION.
        FDRC$A  ,RECBUF,80.          ;INITIALIZES RECORD ACCESS SECTION.
        FDOP$A  OUTLUN,OFDSPT        ;INITIALIZES FILE-OPEN SECTION.
          .
          .
          .
OFDSPT: .WORD   0,0                  ;NULL DEVICE NAME DESCRIPTOR.
        .WORD   0,0                  ;NULL DIRECTORY DESCRIPTOR.
        .WORD   ONAMSZ,ONAM          ;FILENAME DESCRIPTOR.
          .
          .
          .
ONAM:   .ASCII  /OUTPUT.DAT/         ;DEFINES FILENAME STRING.
ONAMSZ=.-ONAM                        ;DEFINES LENGTH OF FILENAME STRING.
          .
          .
          .
```

Note first that an FDB labelled "FDBOUT" is created. Observe further that the FDOP$A macro call takes as its second parameter the symbol "OFDSPT". This symbol represents the address value that is stored in FDB offset location F.DSPT. This value enables the .PARSE routine (see section 4.6.1) to access the fields of the dataset descriptor in building the filename block.

The symbol "OFDSPT" also appears in the label field of the first .WORD directive, defining the address of the dataset descriptor for the .PARSE routine. The .WORD directives each allocate two words of storage for the device name descriptor, the file directory descriptor, and the filename descriptor, respectively.

In the example above, however, note that the first two descriptor fields are filled with zeros, indicating null specifications. The last .WORD directive allocates two words which contain the size and the address of the filename string, respectively. The filename string itself is explicitly defined in the .ASCII directive which follows.

Note that the statements defining the filename string need not be physically contiguous with the dataset descriptor. For each such ASCII string referenced in the dataset descriptor, however, corresponding statements must appear elsewhere in the source program to define the appropriate ASCII data string(s).

A dataset descriptor for each of several files to be accessed by the user program may be defined in this manner.

2.4.2 Default Filename Block - NMBLK$ Macro Call

As noted earlier, the user may also define a default filename block in the program as a means of providing required file information to FCS. For this purpose, the NMBLK$ macro call may be issued in connection with each FDB for which a default filename block is to be defined. When this macro call is issued, space is allocated within the user program for the default filename block, and the appropriate locations within this data structure are initialized according to the parameters supplied in the call.

Note in the parameter descriptions below that symbols of the form N.xxxx are used to represent the offset locations within the filename block. These symbols are differentiated from those that apply to the other sections of the FDB by the beginning character "N". All versions of the generalized OPEN$x macro call (see section 3.1) use these symbols to identify offsets in storing file information in the filename block.

The NMBLK$ macro call is specified in the following format:

```
label: NMBLK$  fnam,ftyp,fver,dvnm,unit
```

where:

label
: represents a user-defined symbol that names the default filename block and defines its address. This label is the symbolic value that is normally specified as the dfnb parameter when the FDOP$A or the FDOP$R macro call is issued, causing FDB offset location F.DFNB to be initialized with the address of the default filename block.

fnam
: represents the default filename. This parameter may consist of up to nine ASCII characters. The character string is stored as six bytes in Radix-50 format, starting at offset location N.FNAM of the default filename block.

ftyp
: represents the default file type. This parameter may consist of up to three ASCII characters. The character string is stored as two bytes in Radix-50 format in offset location N.FTYP of the default filename block.

fver — represents the binary default file version number. When specified, this binary value identifies a particular version of a file. This value is stored in offset location N.FVER of the default filename block.

dvnm — represents the default name of the device upon which the volume containing the desired file is mounted. This parameter consists of two ASCII characters which are stored in offset location N.DVNM of the default filename block.

unit — represents a binary value identifying which unit (among several like units) is to be used in processing the file. If specified, this numeric value is stored in offset location N.UNIT of the default filename block.

Only the characters A through Z and 0 through 9 may be used in composing the filename and file type strings above.

Although the file version number and the unit number above are binary values, these numbers are normally represented in octal form when printed, when input via a command string, or when supplied through a dataset descriptor string.

As evident from the foregoing, all the default information supplied in the NMBLK$ macro call is stored in the default filename block at offset locations which correspond to identical fields in the filename block within the FDB. This default information is moved into the corresponding offsets of the filename block when any version of the generalized OPEN$x macro call is issued under any of the following conditions:

1. All the file information required by FCS to open the file is not present in the dataset descriptor. Missing information is then sought in the default filename block by the .PARSE routine (see section 4.6.1), which is automatically invoked as a result of issuing any version of the generalized OPEN$x macro call.

2. A dataset descriptor has not been created in the user program.

3. A dataset descriptor is present in the user program, but the address of this structure has not been made available to FCS through any of the assembly-time or run-time macro calls which initialize FDB offset location F.DSPT.

The following coding illustrates the general method of specifying the NMBLK$ macro call:

```
FDBOUT: FDBDF$                          ;ALLOCATES SPACE FOR AN FDB.
        FDAT$A  R.VAR,FD.CR             ;INITIALIZES FILE ATTRIBUTE SECTION.
        FDRC$A  ,RECBUF,80.             ;INITIALIZES RECORD ACCESS SECTION.
        FDOP$A  OUTLUN,,OFNAM           ;INITIALIZES FILE OPEN SECTION.

FDBIN:  FDBDF$                          ;ALLOCATES SPACE FOR AN FDB.
        FDRC$A  ,RECBUF,80.             ;INITIALIZES RECORD ATTRIBUTE SECTION.
        FDOP$A  INLUN,,IFNAM            ;INITIALIZES FILE OPEN SECTION.

OFNAM:  NMBLK$  OUTPUT,DAT              ;ESTABLISHES FILENAME AND FILE TYPE.
IFNAM:  NMBLK$  INPUT,DAT,,DT,1         ;ESTABLISHES FILENAME. FILE TYPE,
                                        ;DEVICE NAME, AND UNIT NUMBER.
```

The first NMBLK$ macro call in the coding sequence above creates a default filename block to establish default information for the FDB named "FDBOUT". The label "OFNAM" in this macro defines the beginning address of the default filename block allocated within the user program. Note that this symbol is specified as the dfnb parameter in the FDOP$A macro call associated with this default filename block to initialize the file-open section of the corresponding FDB. The accompanying parameters in the first NMBLK$ macro call define the filename and the file type, respectively, of the file to be opened; all remaining parameter fields in this call are null.

The second NMBLK$ macro call accomplishes essentially the same operations in connection with the FDB named "FDBIN". Note in this macro call that the third parameter (the file version number) is null, as reflected by the extra comma. This null specification indicates that the latest version of the file is desired. All other parameter fields contain explicit declarations defining default information for the applicable FDB.

The offsets for a filename block can be defined locally in the user program, if desired, by issuing the following macro call:

```
        NBOF$L
```

This macro call does not generate any code. Its function is merely to define the filename block offsets locally, presumably to conserve symbol table space at task-build time. The NBOF$L macro call need not be issued if the FDOF$L macro call has been invoked, since the filename block offsets are defined locally as an automatic result of issuing the FDOF$L macro call.

If desired, the user may initialize fields in the default filename block directly with appropriate values. This may be accomplished with in-line statements in the program. For example, a specific offset in the default filename block may be initialized through coding that is logically equivalent to the following:

```
        .
        .
DFNB:   NMBLK$  RSXLIB,OBJ
        .
        .
NUTYP:  .RAD50  /DAT/
        .
        .
        MOV     NUTYP,DFNB+N.FTYP
```

where the symbol "NUTYP" in the MOV instruction above represents the address of the newly-defined Radix-50 file type "DAT" which is to be moved into destination offset N.FTYP of the default filename block labeled "DFNB". Any of the offsets within the default filename block may be manually initialized in this manner to establish desired values or to override previously-initialized values.

2.4.3 Dynamic Processing of File Specifications

For users who wish to make use of a collection of routines available from the system object library (SY:[1,1]SYSLIB.OLB) for processing command line input dynamically, Chapter 6 should be consulted. Chapter 6 describes the Get Command Line Routine (GCML) and the Command String Interpreter (CSI), both of which may be linked with the user program to provide all the logical capabilities required in processing dynamic terminal input or indirect command file input.

2.5 OPTIMIZING FILE ACCESS

When certain information is present in the filename block of an FDB, a file can be opened in a manner referred to throughout this manual as "opening a file by file ID". This type of open requires a minimum of system overhead, resulting in a significant increase in the speed of preparing a file for access by the user program. If files are frequently opened and closed during program execution, opening files by file ID accomplishes substantial savings in overall execution time.

To open a file by file ID, the minimum information that must be present in the filename block of the associated FDB consists of the following:

1. File Identification Field. This 3-word field, beginning at filename block offset location N.FID, contains a file number in the first word and a file sequence number in the second word; the third word is reserved for the implementation of multi-volume/multi-header files. The file identification field is maintained by the system and ordinarily need not be of concern to the user.

2. Device Name Field. This 1-word field at filename block offset location N.DVNM contains the 2-character ASCII name of the device on which the volume containing the desired file is mounted.

3. Unit Number Field. This 1-word field at filename block offset location N.UNIT contains a binary value identifying the particular unit (among several like units) on which the volume containing the desired file is mounted.

These three fields are written into the filename block in either of two ways:

1. As a function of issuing any version of the generalized OPEN$x macro call for a file associated with the FDB in question; or

2. As a result of initializing the filename block manually using the .PARSE routine (see section 4.6.1) and the .FIND routine (see section 4.7.1).

These two methods of setting up the filename block in anticipation of opening a file by file ID are described in detail in the following sections.

2.5.1 Initializing the Filename Block as a Function of OPEN$x

To understand how the process of opening a file by file ID is effected, it should be noted that the initial issuance of the generalized OPEN$x macro call (see section 3.1) for a given file first invokes the .PARSE routine (see section 4.6.1). The .PARSE routine is automatically linked into the user program along with the code for OPEN$x. This routine first zeros the filename block and then fills it in with information taken from the dataset descriptor and/or the default filename block.

Thus, issuing the generalized OPEN$x macro call results in the invocation of the .PARSE routine each time a file is opened. The .PARSE function, however, can be bypassed altogether in subsequent OPEN$x calls by saving and restoring the filename block before attempting to re-open that same file.

This is made possible because of the logic of the OPEN$x macro call. Specifically, after the initial OPEN$x for a file has been completed, the necessary context for re-opening that file exists within the filename block. Therefore, before closing that file, the entire filename block can be copied into user memory space and later restored to the FDB at the desired point in program flow for use in re-opening that same file.

The option to re-open files in this manner stems from the fact that FCS is sensitive to the presence of any non-zero value in the first word of the file identification field of the filename block. When the OPEN$x function is invoked, FCS first examines offset location N.FID of the filename block. If the first word of this field contains a value other than zero (0), FCS logically assumes that the remaining context necessary for opening that file is present in the filename block and, therefore, unconditionally opens that file by file ID.

To ensure that an undesired value does not remain in the first word of the N.FID field from a previous OPEN$x/CLOSE$ sequence, the first word of this field is zeroed as the file is closed.

In opening files by file ID, the user need only ensure that the manual saving and restoring of the filename block are accomplished with in-line MOV instructions that are consistent with the desired sequence of processing files. This process should, in general, proceed as outlined below:

1. Open the file in the usual manner by issuing the OPEN$x macro call.

2. Save the filename block by copying it into user memory space with appropriate MOV instructions. The filename block begins at offset location F.FNB.

   The value of the symbol S.FNB is the size of the filename block in bytes, and the value of the symbol S.FNBW is the size of the filename block in words. If desired, the NBOF$L macro call (see section 2.4.2) may be invoked in the user program to define these symbols locally. These symbolic values may be used in appropriate MOV instructions to accomplish the saving and restoring of the filename block. It is the user's responsibility to reserve sufficient space in the program for saving the filename block.

3. At the end of current file operations, close the file in the usual manner by issuing the CLOSE$ macro call.

4. When, in the normal flow of program logic, that same file is about to be re-opened, restore the filename block to the FDB by doing the reverse of Step 2.

5. Re-open the file by issuing any one of the macro calls available in FCS for opening an existing file. Since the first word of offset location N.FID of the filename block now contains a non-zero value, FCS unconditionally opens the file by file ID, regardless of the specific type of open macro call issued.

Although it is necessary to save only the file identification, device name, and unit number fields of the filename block in anticipation of re-opening a file by file ID, the user is advised to save the entire filename block. The filename, file type, file version number, and directory ID fields, etc., may also be relevant. For example, an OPEN$x, save, CLOSE$, restore, OPEN$x, and DELET$ sequence would require saving and restoring the entire filename block. When the user is logically finished with file processing and he wants to delete the file, the delete operation will not work properly unless the entire filename block has been saved and restored.

2.5.2 Initializing the Filename Block Manually

In addition to saving and restoring the filename block in anticipation of re-opening a file by file ID, the filename block can also be initialized manually. If the user chooses to do so, the .PARSE and .FIND routines (see sections 4.6.1 and 4.7.1, respectively) may be invoked at appropriate points to build the required fields of the filename block. After the .PARSE and .FIND logic is completed, all the information required for opening the file exists within the filename block. When any one of the available FCS macro calls that open existing files is then issued, FCS unconditionally opens that file by file ID.

Occasionally, instances arise which make such manual operations desirable, especially if the user program is operating in an overlaid environment. In this case, it is highly desirable that the code for opening a file be broken up into smaller segments in the interest of conserving memory space. Since the body of code for the OPEN$x and .PARSE functions is sizable, two other types of macro calls for opening files are provided for use with overlaid programs. The OFID$ and OFNB$ macro calls (see sections 3.5 and 3.6, respectively) are specifically designed for this purpose.

The structure recommended for an overlaid environment is to have either the OFID$ or the OFNB$ code on one branch of the overlay and the .PARSE and .FIND code on another branch. Then, if the user wishes to open a file by file ID, the .PARSE and .FIND routines can be invoked at will to insert required information in the filename block before opening the file.

The OFID$ macro call can be issued only in connection with an existing file. The OFNB$ macro call, on the other hand, may be used for opening either an existing file or for creating and opening a new file. In addition, the OFNB$ macro call requires only the manual invocation of the .PARSE routine to build the filename block before opening the file.

If conservation of memory is an objective, and if the user program will be opening both new and existing files, it is recommended that only the OFNB$ routine be included in one branch of the overlay, since including the OFID$ routine would needlessly consume memory space.

In all cases, however, it is important to note that all the macro calls for opening existing files are sensitive to the presence of any non-zero value in the first word (N.FID) of the filename block. If this field contains any value other than zero (0), the file is unconditionally opened by file ID. This does not imply, however, that only the file identification field (N.FID) is required to open the file in this manner. The device name field (N.DVNM) and the unit number field (N.UNIT) must also be appropriately initialized. The logic of the FCS macro calls for opening existing files assumes that these other required fields are present in the filename block if the file identification field contains a non-zero value.

Because many programs continually re-use FDB's, the CLOSE$ function (see section 3.8) zeros the file identification field (N.FID) of the filename block. This action prevents the field (which pertains to a previous operation) from being used mistakenly to open a file for a current operation. Thus, if a user later intends to open a file by file ID using information presently in the filename block, the entire filename block (not just N.FID) must be saved before closing the file. Then, at the appropriate point in program flow, the filename block may be restored to open the desired file by file ID.

2.6 INITIALIZING THE FILE STORAGE REGION

The file storage region (FSR) is an area allocated in the user program as a buffer pool to accommodate the program's block buffer requirements in performing record I/O (GET$ and PUT$) operations. Although the FSR is not applicable to block I/O (READ$ and WRITE$) operations, the FSRSZ$ macro must be issued once in every program that uses FCS, regardless of the type of I/O to be performed.

The macro calls associated with the initialization of the FSR are described below.

2.6.1 FSRSZ$ - Initialize FSR at Assembly-Time

The size of the FSR, as allocated in user memory space, is a function of two variables:

1. The number of files that may be open simultaneously for record I/O operations; and

2. The combined sizes of the respective block buffers to be used for such operations.

The MACRO-11 programmer establishes the size of the FSR at assembly-time by issuing a macro call having the following format:

```
FSRSZ$ files,bufsiz
```

where: files represents a numeric value that is interpreted by FCS according to the following conventions:

1. When a non-zero value is specified, it establishes the maximum number of files that can be open simultaneously for record I/O processing.

2. When zero (0) is specified, it constitutes an implicit declaration that no record I/O processing is to be done. Rather, it indicates that an unspecified number of files may be open simultaneously for block I/O processing.

For example, if the user intends to access three files for block I/O and two files for record I/O, the FSRSZ$ macro call is specified as follows:

```
FSRSZ$ 2
```

On the other hand, if the user intends to access three (or any number of) files for block I/O operations and no files for record I/O operations, the FSRSZ$ macro call takes zero (0) as an argument, as shown below:

```
FSRSZ$  0
```

Thus, the FSRSZ$ macro call must be issued once in every program that uses FCS, regardless of the type of I/O to be performed.

bufsiz represents a numeric value defining the total block buffer pool requirement (in bytes) when all files are open simultaneously for record I/O processing. The combined size of all the FSR block buffers is calculated as described in section 2.7.1. If this parameter is not specified, FCS assumes a default size of 512(10) bytes per block buffer required.

NOTE

An IAS or RSX-11D user must not allocate an FSR block buffer less than 512(10) bytes in length for spooled output to a record-oriented device (such as a line printer).

The FSRSZ$ macro call does not generate any executable code; it merely defines and allocates space for the $$FSR1 program section (i.e., the FSR block buffer pool).

The following statements are illustrative of FSRSZ$ macro calls as they might appear in a user program:

```
FSRSZ$  0

FSRSZ$  2,512.
```

The first statement declares that block I/O operations are to be used in processing files; nothing is implied regarding the number of such files that may be open simultaneously for processing. The last statement explicitly declares that two files may be open simultaneously for record I/O processing; additionally, a maximum of 512(10) bytes will be available in the FSR for use as buffers for these files.

2.6.2 FINIT$ - Initialize FSR at Run-Time

In addition to the FSRSZ$ macro call described in the preceding section, the FINIT$ macro call must also be issued in a MACRO-11 program to call initialization coding to set up the FSR. This macro call takes the following format:

```
label:  FINIT$
```

where: label represents an optional user-specified symbol that allows control to be transferred to this location during program execution. Other instructions in the program may reference this label, as in the case of a program that has been written so that it can be restarted. Considerations relative to the FINIT$ macro call in such a restartable program are presented below.

The FINIT$ macro call should be issued in the program's initialization code. Although the first FCS call issued for opening a file performs the FSR initialization implicitly (if it has not already been accomplished through an explicit invocation of the FINIT$ macro call), it is necessary, in the case of a program that is written so that it can be restarted, to issue the FINIT$ macro call in the program's initialization code, as shown in the second example below. This requirement derives from the fact that such a program performs all its initialization at run-time, rather than at assembly-time.

For example, a program that is not written so that it can be restarted might accomplish the initialization of the FSR implicitly through the following macro call:

```
START:  OPEN$R  #FDBIN          ;IMPLICITLY INITIALIZES THE FSR
                                ;AND OPENS THE FILE.
```

In this case, although transparent to the user, the OPEN$R macro call automatically invokes the FINIT$ operation. The label "START" is the transfer address of the program.

In contrast, a program that embodies the capability to be restarted must issue the FINIT$ macro call explicitly at program initialization in the manner shown below:

```
START:  FINIT$               ;EXPLICITLY INITIALIZES THE FSR AND
        OPEN$R  #FDBIN       ;OPENS THE FILE.
```

In this case, the FINIT$ macro call cannot be invoked arbitrarily elsewhere in the program; it must be issued at program initialization. Doing so forces the appropriate re-initialization of the FSR, whether or not it has been done in a previous execution of the program through an OPEN$x macro call.

Also important in the above context is the fact that calling any of the file-control routines described in Chapter 4, such as .PARSE, first requires the initialization of the FSR. However, the FINIT$ operation must be performed only once per program execution. Note also that FORTRAN programs issue a FINIT$ macro call at the beginning of the program execution; therefore, MACRO-11 routines used with the FORTRAN object time system must not issue a FINIT$ macro call.

2.7 INCREASING THE SIZE OF THE FILE STORAGE REGION

Procedures for increasing the size of the FSR for either MACRO-11 or FORTRAN programs are presented in the following sections.

2.7.1 FSR Extension Procedures for MACRO-11 Programs

To increase the size of the FSR for a MACRO-11 program, the user has two options:

1. Modify the parameters in the FSRSZ$ macro call appropriately to redefine the number of files that may be open simultaneously for record I/O processing and to establish the total buffer pool requirement for these files. Re-assemble the program.

2. Use the EXTSCT (extend program section) command at task-build time to define the new size of the FSR. To invoke this option, the command is specified in the following form:

   ```
   EXTSCT = $$FSR1:length
   ```

   where "$$FSR1" is the symbolic name of the program section within the FSR that is reserved for use as the block buffer pool, and "length" represents a numeric value defining the total required size of the buffer pool in bytes.

   The size of the FSR cannot be reduced at task-build time.

In calculating the total size of the block buffer pool, i.e., the value of "length" in the EXTSCT command above, either of the formulas below may be used:

```
FSR size = S.BFHD*files+bufsiz

FSR size = files*(S.BFHD+512.)
```

where:

S.BFHD — is a symbol which defines the number of bytes required for each block buffer header. If desired, this symbol may be defined locally in the user program by issuing the following macro call:

```
BDOFF$ DEF$L
```

files — represents a numeric value defining the maximum number of files that may be open simultaneously for record I/O processing.

bufsiz — represents a numeric value defining the total number of bytes required for all the FSR block buffers.

The EXTSCT command is described in further detail in the Task Builder Reference Manual of the host operating system.

2.7.2 FSR Extension Procedures for FORTRAN Programs

For a FORTRAN program, if an explicit ACTFIL statement is not issued in the optional keyword input to the Task Builder, an ACTFIL statement with a default value of four (4) is generated automatically during task-build. To extend the size of the FSR at task-build time, the user may issue the following command:

```
ACTFIL = files
```

where: files represents a decimal value defining the maximum number of files that may be open simultaneously for record I/O processing.

This command, similar to the EXTSCT command above, causes program section $$FSR1 to be extended by an amount sufficient to accommodate the number of active files anticipated for simultaneous use by the program.

The size of the FSR for a FORTRAN program can also be decreased at task-build time. As noted above, for either IAS or RSX-11, the default value for the ACTFIL command is 4. Thus, if 0, 1, 2, or 3 is specified as the "files" parameter, the size of $$FSR1 (the FSR block buffer pool) is reduced accordingly.

The ACTFIL command is described in greater detail in the Task Builder Reference Manual of the host operating system.

2.8 COORDINATING I/O OPERATIONS

In the IAS/RSX-11 environment, user programs perform all I/O operations by issuing GET$/PUT$ and READ$/WRITE$ macro calls (see Chapter 3). These calls do not access the physical devices in the system directly. Rather, when any one of these calls is issued, an I/O-related system directive called QUEUE I/O is invoked as the interface between the FCS file-processing routines at the user level and the system I/O drivers at the device level. Device drivers are included for all the standard I/O devices supported by IAS and RSX-11 systems. Although transparent to the user, the QUEUE I/O directive is used for all FCS file access operations.

When invoked, the QUEUE I/O directive instructs the system to place an I/O request for the associated physical device unit into a queue of priority-ordered requests for that unit. This request is placed according to the priority of the issuing task. As required system resources become available, the requested I/O transfer takes place.

As implied above, two separate and distinct processes are involved in accomplishing a specified I/O transfer:

1. The successful queuing of the GET$/PUT$ or READ$/WRITE$ I/O request; and

2. The successful completion of the requested data transfer operation.

These processes, both of which yield success/failure indications that may be tested by the user program, must be performed successfully in order for the specified I/O operation to have been completed. It is important to note that FCS totally synchronizes record I/O operations for the user, even in the case of multiple-buffered operations. In the case of block I/O operations, the flexibility of FCS allows the user to synchronize all block I/O activities, thus enabling him to satisfy logical processing dependencies within the program.

2.8.1 Event Flags

I/O operations proceed concurrently with other system activity. After an I/O request has been queued, the system does not force an implied wait for the issuing task until the requested operation is completed. Rather, the operation proceeds in parallel with the execution of the issuing task, and it is the task's responsibility to synchronize the execution of I/O requests. Tasks use event flags in synchronizing these activities. With respect to event flags, the system merely executes primitive operations that manipulate, test, and/or wait for these indicators of internal task activity.

The completion of an I/O transfer, for example, is recognized by the system as a significant event. If the user has specified a particular event flag to be used by the task in coordinating I/O completion processing, that event flag is set, causing the system to evaluate the eligibility of other tasks to run. Any event flag from 1 through 32(10) may be defined for local use by the task. If the user has not specified an event flag, FCS uses event flag 32(10) by default to signal the completion of I/O transfers.

Specific FDB-initialization and I/O-initiating macro calls in FCS enable the user to specify event flags, if desired, that are unique to his task and which are set and reset only as a result of that task's operation.

For record I/O operations, such an event flag may be defined through the efn parameter of the FDBF$A or the FDBF$R macro call (see section 2.2.1.6 or 2.2.2, respectively).

For block I/O operations, an event flag may be declared through the bkef parameter of the FDBK$A or the FDBK$R macro call (see section 2.2.1.4 or 2.2.2, respectively); alternatively, a block event flag may be declared through the corresponding parameter of the I/O-initiating READ$ or WRITE$ macro call (see section 3.15 or 3.16, respectively).

In both record and block I/O operations, the event flag is cleared when the I/O request is queued and set when the I/O operation is completed. In the case of record I/O operations, only FCS manipulates the event flag. Additionally, the user is unaware of the event flag's state and he has no need to know. Furthermore, the user must not issue a WAITFOR system directive predicated on the event flag used for coordinating record I/O operations. A record I/O operation, for example, may not even involve an I/O transfer; rather, it may only involve the blocking or deblocking of a record within the FSR block buffer. On the other hand, the event flag defined for synchronizing block I/O operations is totally under the user's control.

Through event-associated system directives, the user can clear event flags, set event flags, test whether a specified event flag is set, or cause a task to be suspended until a specified event flag is set. These event-associated directives are described in detail in the Executive Reference Manual of the host operating system. The setting and checking of event flags allow tasks in a real-time system to communicate with each other and thereby synchronize their execution.

Event flags and device-dependencies related thereto are described in further detail in the IAS/RSX-11D Device Handlers Reference Manual or the RSX-11M I/O Drivers Reference Manual.

Also, a code indicating the success or failure of the QUEUE I/O request resulting from the READ$/WRITE$ macro call is returned to the Directive Status Word ($DSW). If desired, symbolic location $DSW may be tested to determine the status of the I/O request. The success/failure codes for the QUEUE I/O directive are listed in the manuals referenced above.

2.8.2 I/O Status Block

Because of the comparative complexity of block I/O operations, an optional parameter is provided in the FDBK$A and the FDBK$R macro calls, as well as the READ$ and WRITE$ macro calls, which enables the system to return status information to the user task for block I/O operations. The I/O status block is not applicable to record I/O (GET$ or PUT$) operations.

This optional parameter, called the I/O status block address, is made available to FCS through any of the macro calls identified above.

When this parameter is supplied, the system returns status information to a 2-word block reserved in the user program. Although the I/O status block is used principally as a QUEUE I/O housekeeping mechanism for containing certain device-dependent information, this area also contains information of particular interest to the user.

Specifically, the second word of the I/O status block is filled in with the number of bytes transferred during a READ$ or WRITE$ operation. When performing READ$ operations, it is good practice to always use the value returned to the second word of the I/O status block as the number of bytes actually read, rather than assuming that the requested number of bytes was transferred. Employing this technique allows the program to properly read virtual blocks of varying length from a device such as a magnetic tape unit, provided that the requested byte count is at least as large as the largest virtual block. (For magnetic tape units, almost all virtual blocks are 512(10) bytes or less in length.) For WRITE$ operations, the specified number of bytes are always transferred, otherwise an error condition exists.

Also, the low-order byte of the first word of the I/O status block contains a code which reflects the final status of the READ$/WRITE$ operation. The codes returned to this byte may be tested to determine the status of any given block I/O transfer. The binary values of these status codes always have the following significance:

Code Value	Meaning
+	I/O transfer completed.
0	I/O transfer still pending.
-	I/O error condition exists.

The format of the I/O status block and the error codes returned to the low-order byte of its first word are described in detail in the IAS/RSX-11D Device Handlers Reference Manual or the RSX-11M I/O Drivers Reference Manual.

If the address of the I/O status block is not made available to FCS (and hence to the QUEUE I/O directive) through any of the macro calls noted above, no status information is returned to the I/O status block. In this case, the fact that an error condition may have occurred during a READ$ or WRITE$ operation is simply lost. Thus, supplying the address of the I/O status block to the associated FDB is highly desirable in order to facilitate normal error reporting.

An I/O status block may be defined in the user program at assembly-time through any storage directive logically equivalent to the following:

```
IOSTAT: .BLKW   2
```

where the label "IOSTAT" is a user-defined symbol naming the I/O status block and defining its address. This symbolic value is specified as the bkst parameter in the FDBK$A or the FDBK$R macro call to initialize FDB offset location F.BKST; it may also be specified as the corresponding parameter in the READ$ or the WRITE$ macro call, initializing this cell in the FDB as an integral function of issuing the desired I/O request.

2.8.3 AST Service Routine

An asynchronous system trap (AST) is a software-generated interrupt that causes the sequence of instructions currently being executed to be interrupted and control to be transferred to another instruction sequence elsewhere in the program. If desired, the user may specify the address of an AST service routine that is to be entered upon completion of a block I/O transfer. Since an AST is a trap action, it constitutes an automatic indication of block I/O completion.

The address of an AST service routine may be specified as an optional parameter (bkdn) in the FDBK$A or the FDBK$R macro call (see section 2.2.1.4 or 2.2.2, respectively); this parameter may also be specified in the READ$ or the WRITE$ macro call, initializing the FDB at the time the I/O request is issued (see section 3.15 or 3.16, respectively).

Usually, an AST address is specified to enable a running task to be interrupted in order to execute special code upon completion of a block I/O request. If the address of an AST service routine is not specified, the transfer of control does not occur, and normal task execution continues.

The main purpose of an AST service routine is to inform the user program that a block I/O operation has been completed, thus enabling the program to continue immediately with some other desired (and perhaps logically dependent) operation (e.g., another I/O transfer).

If an AST service routine is not provided by the user, some other mechanism, such as event flags or the I/O status block, must be used as a means of determining block I/O completion. In the absence of such a routine, for example, the user may test the low-order byte of the first word in the I/O status block to determine if the block I/O transfer has been completed. A WAIT$ macro call (see section 3.18) may also be issued in connection with a READ$ or WRITE$ operation to suspend task execution until a specified event flag is set to indicate the completion of block I/O.

The implementation of an AST service routine in the user program is application-dependent and must be coded specifically to meet particular user I/O processing requirements. A detailed discussion of asynchronous system traps is beyond the scope of this document. The reader is therefore referred to the Executive Reference Manual of the host operating system for discussions of various trap-associated system directives.

CHAPTER 3

FILE-PROCESSING MACRO CALLS

The user manipulates files through a set of file-processing macro calls. These macro calls are invoked and expanded at assembly-time. The resulting code is then executed at run-time to perform the operations listed below:

OPEN$	- To open and prepare a file for processing;
OPNS$	- To open and prepare a file for processing and to allow shared access to that file (depending on the mode of access);
OPNT$	- To create and open a temporary file for processing;
OFID$	- To open an existing file using file identification information in the filename block;
OFNB$	- To open a file using filename information in the filename block;
CLOSE$	- To terminate file processing in an orderly manner;
GET$	- To read logical data records from a file;
GET$R	- To read fixed-length records from a file in random mode;
GET$S	- To read records from a file in sequential mode;
PUT$	- To write logical data records to a file;
PUT$R	- To write fixed-length records to a file in random mode;
PUT$S	- To write records to a file in sequential mode;
READ$	- To read virtual data blocks from a file;
WRITE$	- To write virtual data blocks to a file;
DELET$	- To remove a named file from the associated volume directory and to deallocate the space occupied by the file; and
WAIT$	- To suspend program execution until a requested block I/O operation is completed.

Most of the parameters associated with the file-processing macro calls supply information to the FDB. Such parameters cause MOV or MOVB instructions to be generated in the object code, resulting in the initialization of specific locations within the FDB.

The final parameter in all file-processing macro calls is the symbolic address of a user-coded error-handling routine. This routine is entered upon detection of an error condition during the file-processing operation. When this optional parameter is specified, the following code is generated:

```
        Code for macro
          .
          .
          .
        BCC     nn$             ;TESTS C-BIT IN PROCESSOR STATUS WORD.
        JSR     PC,ERRLOC       ;INITIATES ERROR-HANDLING ROUTINE
                                ;AT "ERRLOC" ADDRESS.
nn$:                            ;CONTINUES NORMAL PROGRAM EXECUTION.
```

where "nn$" represents an automatically-generated local symbol. If the operation is completed successfully, the C-bit (carry condition code) in the Processor Status Word is not set, and FDB offset location F.ERR contains a positive value. The BCC instruction then results in a branch to the local symbol "nn$" and the continuation of normal program execution.

If, however, an error condition is detected during the execution of the file-processing routine, the C-bit in the Processor Status Word is set, FDB offset location F.ERR contains a negative value (indicating an error condition), and the branch to the local symbol "nn$" does not occur. Instead, the JSR instruction is executed, loading the PC with the symbolic address (ERRLOC) of the error-handling routine and initiating its execution.

If this optional parameter is not specified, the error processing routine is not called, and the user must explicitly test the C-bit in the Processor Status Word to ascertain the status of the requested operation.

Note that the execution of the FCS file-processing routines causes all user program general registers to be saved, except R0, which, by convention, is used by FCS to contain the address of the FDB associated with the file being processed.

3.1 OPEN$x - GENERALIZED OPEN MACRO CALL

Before any file can be processed by the user (or system) program, it must first be opened. The type of action that the user intends to perform on a file is indicated to FCS by an alphabetic suffix accompanying the macro name. For example, in issuing the generalized macro call,

```
        OPEN$x
```

"x" represents any one of the following alphabetic suffixes, each of which denotes a specific type of processing anticipated for the file:

R - Read an existing file;

W - Write (create) a new file;

M - Modify an existing file without changing its length;

U - Update an existing file and extend its length, if necessary; or

A - Append (add) data to the end of an existing file.

NOTE

The generalized OPEN$x macro call can be issued without an alphabetic suffix. In this case, the type of action to be performed on the file is indicated to FCS through an additional parameter in the macro call. This value, called the file-access (facc) parameter, causes offset location F.FACC in the associated FDB to be initialized. Section 3.7 describes this macro call in detail.

Depending on the alphabetic suffix supplied in the OPEN$x macro call, certain other types of operations may or may not be allowed, as noted below:

1. If R is specified (for reading an existing file), that file cannot also be written, i.e., a PUT$ or WRITE$ operation cannot be performed on that file.

2. If M or U is specified (for modifying or updating an existing file), that file can be both read and written, i.e., concurrent GET$/PUT$ or READ$/WRITE$ operations may be performed on that file.

3. If M is specified (for modifying an existing file), that file cannot be extended.

4. If W or A is specified (for creating a new file or appending data to an existing file), that file may be read, written, and/or extended.

The program that is issuing the OPEN$x macro call must have appropriate access privileges for the action specified. Table 3-1 summarizes the access privileges for the various forms of the OPEN$x macro call. This table also shows where the next record or block will be read or written in the file after it is opened.

Table 3-1
File Access Privileges Resulting from OPEN$x Macro Call

MACRO	ACCESS PRIVILEGES	POSITION OF FILE AFTER OPEN$x
OPEN$R	Read	First record of existing file.
OPEN$W	Read, write, extend	First record of new file.
OPEN$M	Read, write	First record of existing file.
OPEN$U	Read, write, extend	First record of existing file.
OPEN$A	Read, write, extend	End of existing file. (For special PUT$R considerations, see section 3.13.)

When any form of the OPEN$x macro call is issued, FCS first fills in the filename block with filename information retrieved from the dataset descriptor (see section 2.4.1). FCS gains access to this data structure through the address value stored in FDB offset location F.DSPT.

If any required data has been omitted from the dataset descriptor, FCS attempts to obtain the missing information from the default filename block. This data structure, which may also contain user-specified filename information, is created in the program by issuing the NMBLK$ macro call (see section 2.4.2). FCS gains access to this structure through the address value stored in FDB offset location F.DFNB.

The address values in offset locations F.DSPT and F.DFNB may be supplied to FCS through the FDOP$A macro call, the FDOP$R macro call, or the OPEN$x macro call. FCS requires access to the dataset descriptor and/or the default filename block in retrieving filename information used in opening files.

If a new file is to be created, the OPEN$W macro call is issued. FCS then performs the following operations:

1. Creates a new file and obtains file identification information for the file. File identification information is maintained by FCS in offset location N.FID of the filename block. The filename block in the FDB begins at offset location F.FNB.

2. Initializes the file attribute section of the file header block using information obtained from the FDB associated with the file being created. Each file on a volume has an associated file header block that describes the attributes of that file. The format and content of the file header block are presented in detail in Appendix F.

3. Places an entry for the file in the user file directory (UFD). If, however, an entry for a file having the same name, type, and version number already exists in the UFD, the old file is deleted. If a particular type of macro call is issued explicitly specifying that the file not be superseded, the old file is not deleted. This type of OPEN$ operation is described in section 3.7.

4. Associates the assigned logical unit number (LUN) with the file to be created.

5. Allocates a buffer for the file from the FSR block buffer pool if record I/O (GET$/PUT$) operations are to be used in processing the file.

If an existing file is to be opened, any one of the following macro calls may be issued: OPEN$R, OPEN$M, OPEN$U, or OPEN$A. FCS then performs the following operations:

1. If file identification information is not present in the filename block, FCS constructs the filename block from information taken from the dataset descriptor and/or the default filename block. FCS then searches the user file directory (UFD) by filename to obtain the required file identification information. When found, this information is stored in the filename block, beginning at offset location N.FID.

2. Associates the assigned logical unit number (LUN) with the file.

3. Reads the file header block and initializes the file attribute section of the FDB associated with the file being opened.

4. Allocates a buffer for the file from the FSR block buffer pool if record I/O (GET$/PUT$) operations are to be used in processing the file.

NOTE

As described in section 2.6, the user allocates buffers through the FSRSZ$ macro call. The number of buffers allocated is dependent upon the number of files that the user intends to open simultaneously for record I/O operations.

If block I/O operations are to be used, FDB offset location F.RACC must be initialized with the FD.RWM parameter via the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call. This parameter inhibits the allocation of a buffer when the file is opened.

3.1.1 Format of Generalized OPEN$x Macro Call

The generalized macro call for opening files takes the following form:

```
OPEN$x  fdb,lun,dspt,racc,urba,urbs,err
```

where: x — represents the alphabetic suffix specified as part of the macro name, indicating the desired type of operation to be performed on the file. The possible values for this parameter are: R, W, M, U, or A (see section 3.1).

fdb — represents the symbolic address of the associated FDB.

lun — represents the logical unit number (LUN) associated with the desired file. This parameter identifies the device on which the volume containing the desired file is mounted. Normally, the logical unit number associated with the file is specified through the corresponding parameter of the FDOP$A or the FDOP$R macro call. If so specified, the lun parameter need not be present in the OPEN$x macro call. Each FDB must have a unique LUN.

dspt — represents the symbolic address of the dataset descriptor. Normally, this address value is specified through the corresponding parameter of the FDOP$A or the FDOP$R macro call. If so specified, this parameter need not be present in the OPEN$x macro call.

This parameter specifies the address of the manually-created dataset descriptor (see section 2.4.1). If the Command String Interpreter (CSI) is being used to interpret command lines dynamically, this parameter is used to specify the address of the dataset descriptor within the CSI control block (see offset location C.DSDS in section 6.2.2).

racc — represents the record access byte. One or more symbolic values may be specified in this field to initialize the record access byte (F.RACC) in the assocated FDB. Any combination of the following parameters may be specified:

FD.RWM - Indicates that block I/O (READ$/WRITE$) operations are to be used in processing the file. If this parameter is not specified, FCS assumes by default that record I/O (GET$/PUT$) operations are to be used in processing the file.

FD.RAN - Indicates that random access to the file is to be used for record I/O (GET$/PUT$) operations. If this parameter is not specified, FCS uses sequential access by default.

FD.PLC - Indicates that locate mode (see section 1.6.2) is to be used for record I/O (GET$/PUT$) operations. If this parameter is not specified, FCS uses move mode (see section 1.6.1) by default.

FD.INS - Indicates that a PUT$ operation in sequential mode in the body of a file shall not truncate the file. Effectively, this parameter prevents the logical end of the file from being reset to a point just beyond the inserted record. If this parameter is not specified, a PUT$ operation in sequential mode truncates the file to a point just beyond the inserted record, but no deallocation of file blocks occurs.

The specification of this parameter allows a data record in the body of the file to be overwritten. Care must be exercised, however, to ensure that the record being written is the same length as the record being replaced.

If the FD.RAN parameter above is specified, the file is accessed in random mode. In this case, a PUT$ operation in the file, without exception, does not truncate the file.

If the record access byte in the FDB has already been initialized through the corresponding parameters of the FDRC$A or the FDRC$R macro call, the racc parameters need not be present in the OPEN$x macro call.

urba — represents the symbolic address of the user record buffer. This parameter initializes FDB offset location F.URBD+2.

If the user record buffer address has already been supplied to the FDB through the corresponding parameter of the FDRC$A or the FDRC$R macro call, this parameter need not be present in the OPEN$x macro call.

urbs — represents a numeric value defining the size of the user record buffer (in bytes). This parameter initializes FDB offset location F.URBD.

If the size of the user record buffer has already been supplied to the FDB through the corresponding parameter of the FDRC$A or the FDRC$R macro call, this parameter need not be present in the OPEN$x macro call.

err — represents the symbolic address of an optional user-coded error-handling routine.

Specific FDB requirements for record I/O operations (GET$ and PUT$ macro calls) are detailed in sections 3.9.2 and 3.12.2.

The following examples depict representative uses of the OPEN$x macro call.

A macro call to open and modify an existing file, for example, might take the following form:

```
OPEN$M  R0,#INLUN,,#FD.RAN!FD.PLC
```

Note in this macro call that the FDB address is assumed to be present in R0. The third parameter, i.e., the dataset descriptor pointer, is not specified; this null specification (indicated by the extra comma) assumes that FDB offset location F.DSPT (if required) has already been initialized. The last parameter, consisting of two values separated by an exclamation point, establishes random access and locate modes for GET$/PUT$ operations.

The following macro call might be issued to update an existing file:

```
OPEN$U  R0,#INLUN,,,#RECBUF,#80.
```

This macro call also assumes that the FDB address is in R0. Note also that the dspt and racc parameter fields are null, based on the premise that the dataset descriptor pointer (F.DSPT) has been provided previously to the FDB and that the record access byte (F.RACC) has also been previously initialized. Finally, the last two parameters establish the address and the size of the user record buffer, respectively.

This last example shows a macro call that might be issued to allow data to be appended to the end of a file:

```
OPEN$A  #OUTFDB
```

This macro call specifies the address of an FDB as the only parameter. In this case, it is assumed that all other parameters required by FCS in opening and operating on the file have been previously supplied to the FDB through the appropriate assembly-time or run-time macro calls.

Note in all three examples above that the error parameter is not specified, requiring that the user explicitly test the C-bit in the Processor Status Word to ascertain the success of the specified operation.

3.1.2 FDB Requirements for Generalized OPEN$x Macro Call

The information required for opening a file may be supplied to the FDB through the following macro calls:

1. The assembly-time macro calls described in section 2.2.1.
2. The NMBLK$ macro call described in section 2.4.2.
3. The run-time macro calls described in section 2.2.2.
4. The various macro calls described in this chapter for opening files.

The particular combination of macro calls used to define and initialize the FDB is a matter of choice, as indicated above. Of far greater significance is the fact that certain information must be present in the FDB before the associated file can be opened. In this regard, the following rules apply for creating and opening new files, for opening existing files, and for specifying desired file options:

1. To Create a New File. If a new file is to be created through the OPEN$W macro call, the following information must first be supplied to the FDB. This information may be specified through the FDAT$A macro call (see section 2.2.1.2) or the FDAT$R macro call (see section 2.2.2):

   a. The record type must be established for record I/O operations. To accomplish this, byte offset location F.RTYP must be initialized with either of the following symbolic values:

R.FIX - Indicates that fixed-length records are to be written into the file.

R.VAR - Indicates that variable-length records are to be written into the file.

b. The desired record attributes must be specified for record I/O operations. The record attributes are defined by initializing byte offset location F.RATT with the appropriate value(s), as follows:

FD.FTN - Indicates that the first byte of each record is to contain a FORTRAN carriage-control character.

FD.CR - Indicates that a line-feed (<LF>) character is to precede each record and that a carriage-return (<CR>) character is to follow the record when that record is output to a device requiring carriage-control information (e.g., to a terminal). The <LF> and <CR> characters are not actually embedded within the record. Their presence is merely implied through the file attribute FD.CR.

FD.BLK - Indicates that records are not allowed to cross block boundaries.

c. If fixed-length records are to be written to the file, the record size (in bytes) must be specified for record I/O operations to appropriately initialize FDB offset location F.RSIZ.

Items a. through c. above cannot be supplied to the FDB through any of the various macros used to create and/or open files (e.g., OPEN$W, OPEN$R, etc.). Furthermore, none of the above information is required when opening an existing file, since FCS obtains such information from the first 14 bytes of the user file attribute section of the file's header block (see Appendix F).

2. To Open Either a New File or an Existing File. Regardless of whether the file being opened is yet to be created or already exists, the following information must be present in the FDB before that file can be opened:

a. The record access byte must be initialized for record/block I/O operations. The symbolic values below may be specified in the FDRC$A macro call (see section 2.2.1.3), the FDRC$R macro call (see section 2.2.2), or the generalized OPEN$x macro call to initialize FDB offset location F.RACC:

FD.RWM - Indicates that READ$/WRITE$ (block I/O) operations are to be used in processing the file. If this parameter is not specified, GET$/PUT$ (record I/O) operations result by default.

FD.RAN - Indicates that random access mode (GET$/PUT$ record I/O) is to be used in processing the file. If this parameter is not specified, sequential access mode results by default.

FD.PLC - Indicates that locate mode (GET$/PUT$ record I/O) is to be used in processing the file. If this parameter is not specified, move mode results by default.

FD.INS - Indicates that a PUT$ operation in sequential mode in the body of a file shall not truncate the file. If this parameter is not specified, such an operation truncates the file. In this case, the logical end of the file is reset to a point just beyond the inserted record, but no deallocation of file blocks occurs.

b. The user record buffer descriptors, i.e., the urba and urbs parameters, must be specified for record I/O operations. To accomplish this, the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call may be used. The selected macro call defines the address and the size of the area reserved in the program for use as a buffer during record I/O operations. The urba and urbs parameters initialize FDB offset locations F.URBD+2 and F.URBD, respectively.

   FDB requirements specific to GET$ and PUT$ operations in move and locate mode are presented in detail in sections 3.9.2 and 3.12.2, respectively.

c. The logical unit number must be specified to initialize FDB offset location F.LUN. The initialization of this cell can be accomplished through the lun parameter of the FDOP$A, the FDOP$R, or the generalized OPEN$x macro call. Each FDB must have a unique logical unit number.

d. If file identification information is not already present in the FDB, either the dataset descriptor pointer (F.DSPT) or the default filename block address (F.DFNB) must be specified to enable FCS to obtain required filename information for use in opening the file. These address values may be specified in either the FDOP$A macro call (see section 2.1.1.5) or the FDOP$R macro call (see section 2.2.2). The generalized OPEN$x macro call (see section 3.1) may also be used to specify the dataset descriptor pointer.

e. If desired, an event flag number for synchronizing record I/O operations must be specified to initialize FDB offset location F.EFN. This optional parameter may be specified in either the FDBF$A macro call (see section 2.2.1.6) or the FDBF$R macro call (see section 2.2.2). If not specified, FCS uses event flag number 32(10) by default in synchronizing all record I/O activity.

3. Specifying Desired File Options. If certain options are desired for a given file, they must be specified before that file is opened. Since this information is needed only in opening the file, it is zeroed when the file is closed, thus ensuring that the FDB is properly re-initialized for subsequent use. The options that may be specified for a given file are described below:

   a. The override block size (ovbs parameter) must be specified in either the FDBF$A or the FDBF$R macro call to initialize FDB offset location F.OVBS. This parameter need be specified only if the standard default block size in effect for the associated device is to be overridden. The override block size is specified only in connection with record-oriented devices (such as line printers) and sequential devices (such as magnetic tape units).

b. The multiple buffer count (mbct parameter) must be specified in either the FDBF$A or the FDBF$R macro call to initialize FDB offset location F.MBCT. If multiple-buffered record I/O operations are to be used, this parameter must be greater than one (1), and it must agree with the desired number of buffers to be used. This parameter is not overlaid, nor is it zeroed when the file is closed.

   If the multiple buffer count is not established as described above, multiple buffered operations can still be invoked by changing the default buffer count in the FSR. A default buffer count of one (1) is stored in symbolic location .MBFCT of $$FSR2. This default value can be altered to reflect the number of buffers intended for use during record I/O operations. The procedure for modifying this cell in $$FSR2 is described at the end of section 2.2.1.6.

   Also, if multiple buffering is to be employed, the appropriate control flag must be specified as the mbfg parameter in either the FDBF$A or the FDBF$R macro call to appropriately initialize FDB offset location F.MBFG. Either of two symbolic values may be specified for this purpose, as follows:

   FD.RAH - Indicates that read-ahead operations are to be used in processing the file.

   FD.WBH - Indicates that write-behind operations are to be used in processing the file.

   Offset location F.MBFG need be initialized only if the standard default buffering assumptions are inappropriate. When a file is opened for reading (OPEN$R), read-ahead operations are assumed by default; for all other forms of OPEN$x, write-behind operations are assumed. It may be useful, for example, to override the write-behind default assumption for a file opened through the OPEN$M or the OPEN$U macro call when that file is being used basically for sequential read operations, but scattered updating is also being performed.

c. To allocate required file space at the time a file is created, the cntg parameter must be specified in either the FDAT$A or the FDAT$R macro call. This parameter initializes FDB offset location F.CNTG. A positive value so specified results in the allocation of a contiguous file having the specified number of blocks; a negative value, on the other hand, results in the allocation of a noncontiguous file having the specified number of blocks.

d. The address of the 5-word statistics block in the user program must be moved manually into FDB offset location F.STBK. This address value specifies an area in the user program to which FCS returns certain statistical information about a file when it is opened. If this parameter is not specified, no return of such information occurs.

The format and content of the statistics block are presented in Appendix H. If the user elects to define such an area in his program, he may do so with coding logically equivalent to that shown below:

```
STBLK:  .BLKW   5
```

Offset location F.STBK may then be manually initialized, as follows:

```
        MOV     #STBLK,FDBADR+F.STBK
```

where "STBLK" is the user-defined symbolic address of the statistics block, and the destination operand of this instruction defines the appropriate offset location within the desired FDB.

3.2 OPNS$x - OPEN FILE FOR SHARED ACCESS

The OPNS$x macro call is issued to open a file for shared access. This macro call has the same format, i.e., takes the same alphabetic suffixes and run-time parameters, as the generalized OPEN$x macro call. The shared access conditions which result from the use of this macro call are summarized in section 1.8.

3.3 OPNT$W - CREATE AND OPEN TEMPORARY FILE

The OPNT$W macro call is issued to create and open a temporary file for some special purpose of limited duration. If a temporary file is to be used only once, it is best created through the OPNT$D macro call described in the following section.

The OPNT$W macro call creates a file but does not enter a filename for that file into any associate user directory file. This macro call simply enters appropriate file identification information into the volume's index file and, in addition, maintains the file identification field (offset location N.FID) in the associated filename block. The index file is a file which consists of file header blocks for user files (see Appendix E).

In using the OPNT$W macro call, the user bears the responsibility for marking the temporary file for deletion, as described in the procedure below. Then, after all operations associated with that file are completed, closing the file also results in its deallocation. All space formerly occupied by the file is then returned to the pool of available storage on the volume for reallocation.

Although the OPNT$W macro call takes the same parameters as the generalized OPEN$x macro call, the former executes faster because no directory entries are made for a temporary file.

Creating a temporary file is usually done when a program requires a file only for the duration of its execution (e.g., for use as a work file). The general sequence of operations in such instances proceeds as follows:

1. Open a temporary file by issuing the OPNT$W macro call. Perform any desired operations on that file. If the file is to be used only for a single OPNT$W/CLOSE$ sequence, go to Step 6; otherwise, continue with Step 2.

2. Before closing the file for processing, save the filename block in the associated FDB. The general procedure for saving (and restoring) the filename block is discussed in section 2.5.1.

3. Close the file by issuing the CLOSE$ macro call (see section 3.8). Continue other processing in the program, as desired.

4. In anticipation of re-opening the temporary file, restore the filename block to the FDB by accomplishing the reverse of Step 2 above.

5. Re-open the file by issuing any of the FCS macro calls which open existing files. Resume operations on the file; repeat the save, CLOSE$, restore, open sequence any desired number of times.

6. Before closing the file the last time, call the .MRKDL routine, as shown below, to mark the file for deletion:

   ```
   CALL    .MRKDL
   ```

   The .MRKDL routine is described in section 4.13.1.

7. Close the file by issuing the CLOSE$ macro call.

If the filename block is not saved, the file identification field therein is destroyed, since this field is reset to zero (0) when the file is closed.

Thus, not saving the filename block before closing a temporary file results in a "lost" file, since no directory entry is made for a temporary file. The usual procedure of listing the volume's directory is therefore inapplicable. The only way such a file can be recovered is to use the file structure verification utility program (VFY) to search the volume's index file. The VFY program has the capability to compare the files listed in all the directories on the volume with those listed in the index file. If a file appears in the index file, but not in a directory, VFY identifies that file for the user. This program is described in detail in the IAS System Management Guide, RSX-11D Utility Programs Procedures Manual, or RSX-11M Utilities Procedures Manual.

3.4 OPNT$D - CREATE AND OPEN TEMPORARY FILE AND MARK FOR DELETION

The OPNT$D macro call is issued to create and open a temporary file and, in addition, to mark the file for deletion. File identification information for such a file is entered into the volume's index file and the filename block in the associated FDB (but not in any associated volume directory). A file marked for deletion cannot be opened by another program. Furthermore, when the file is closed, it is automatically deleted from the volume, returning its space to the pool of available storage on the volume for reallocation.

The presumption in issuing the OPNT$D macro call is that the file thus created is to be used only once. This is a particularly desirable way to open a temporary file, since the file will be deleted, even if the program terminates abnormally without closing the file.

The OPNT$D macro call takes the same format and parameters as the generalized OPEN$x macro call.

3.5 OFID$ - OPEN FILE BY FILE ID

The OFID$ macro call is issued to open an existing file using information stored in the file identification field (offset location N.FID) of the filename block. Thus, issuing this macro call invokes an FCS routine which opens a file only by file ID (see section 2.5). The OFID$ call, which has the same format and takes the same parameters as the generalized OPEN$x macro call (see section 3.1), is designed for use with overlaid programs.

In describing the functions of the OFID$ macro call, either one of two assumptions may apply, as follows:

1. That the necessary context for opening the file has been saved from a previous OPEN$x operation and restored to the filename block in anticipation of opening that file by file ID. The saving and restoring of the filename block are discussed in detail in section 2.5.1.

2. That the desired file is to be opened for the first time. In this case, the necessary context for opening the file must first be stored in the filename block before the OFID$ macro call can be issued.

In most cases, the latter assumption applies, requiring that the following procedures be performed:

1. Call the .PARSE routine (see section 4.6.1). This routine takes information from a specified dataset descriptor and/or default filename block and initializes and fills in the specified filename block.

2. Call the .FIND routine (see section 4.7.1). This routine locates an appropriate directory entry for the file (by filename) and stores the file identification information therefrom in the 6-byte file identification field of the filename block, starting at offset location N.FID. As a result of Steps 1 and 2, the necessary context then exists in the associated filename block for opening the file by file ID.

3. Issue the OFID$ macro call.

The advantage in using the .PARSE and .FIND routines in conjunction with the OFID$ macro call is that the user can overlay his program, placing .PARSE and .FIND on one branch, and the code for OFID$ on another branch. This overlay structure reduces the program's overall memory requirements.

Unlike the other FCS macro calls for opening files, the OFID$ macro call requires a non-zero value in the first word of the file identification field (N.FID) in order to work properly. When this field contains a non-zero value, FCS assumes that the remaining context necessary for opening that file is present and, accordingly, opens the file by file ID.

3.6 OFNB$ - OPEN FILE BY FILENAME BLOCK

The OFNB$ macro call is issued to open either an existing file or to create and open a new file using filename information in the filename block. Similar to the OFID$ macro call above, the OFNB$ call is designed for use with overlaid programs. However, the OFNB$ macro call differs in two important respects: it can be issued to create a new file, and it can be issued to open a file by filename block.

In describing the functions of the OFNB$ macro call, the same assumptions outlined above for OFID$ apply, viz., that the filename block has been saved and restored in anticipation of issuing the OFNB$ macro call, or that the file is being opened for the first time. Since the procedures for saving and restoring the filename block are detailed in section 2.5.1, the following discussion assumes that the desired file is being opened for the first time. In this case, the filename block in the FDB must be initialized, as described below.

To open a file by filename block, the following information must be present in the filename block of the associated FDB:

A. The filename (offset location N.FNAM);

B. The file type or extension (offset location N.FTYP);

C. The file version number (offset location N.FVER);

D. The directory ID (offset location N.DID);

E. The device name (offset location N.DVNM); and

F. The unit number (offset location N.UNIT).

In providing the information above to the filename block, either of two general procedures may be used, as described in the following sections.

3.6.1 Dataset Descriptor and/or Default Filename Block

If the dataset descriptor contains all the required information listed above, perform the following procedures:

1. Call the .PARSE routine (see section 4.6.1). This routine takes information from a specified dataset descriptor and/or default filename block and fills in the appropriate offsets of a specified filename block.

2. Issue the OFNB$ macro call.

3.6.2 Default Filename Block Only

If a default filename block is to be used in providing the required information to FCS, perform the following procedures:

1. Issue the NMBLK$ macro call (see section 2.4.2) to create and initialize a default filename block. With the exception of the directory ID, this structure provides all the requisite information to FCS.

2. To provide the directory ID, call either of the following routines:

   a. Call the .GTDIR routine (see section 4.8.1) to retrieve the directory ID from the specified dataset descriptor and to store the directory ID in the default filename block; or

   b. Call the .GTDID routine (see section 4.8.2) to retrieve the default UIC from $$FSR2 and to store the directory ID in the default filename block.

   c. Move the entire default filename block manually into the filename block associated with the file being opened.

3. Issue the OFNB$ macro call.

Note that the coding for OFNB$ operations normally resides in an overlay apart from that containing the other FCS routines identified above.

The issuance of the OFNB$ macro call is usually done under the premise that the filename block contains the requisite information, as described above. However, if the file identification field (offset location N.FID) in the filename block contains a non-zero value when the call to OFNB$ is issued, the file is unconditionally opened by file ID.

The OFNB$ macro call has the same format and takes the same parameters as the generalized OPEN$x macro call (see section 3.1).

If the user expects to open both new and existing files, and memory conservation is an objective, the OFNB$ macro call is most suitable for opening such files. The OFID$ coding should not be included in the same overlay with OFNB$, since OFID$ overlaps the function of OFNB$ and, therefore, needlessly consumes memory space.

3.7 OPEN$ - GENERALIZED OPEN FOR SPECIFYING FILE ACCESS

Usually, when the user wishes to create a file, the filename and the file type are specified, and FCS is allowed to assign the next higher file version number. However, if the OPEN$W macro call is issued for a file having an explicit filename, file type, and file version number, and a file of that description already exists in the specified user file directory (UFD), the old file is superseded.

By issuing the OPEN$ macro call without an alphabetic suffix, and by specifying two additional parameters, the user can inhibit the automatic supersession of a file when a duplicate file specification is encountered in the UFD. Rather than deleting the old version of the file, an error indication (IE.DUP) is returned to offset location F.ERR of the applicable FDB.

All parameters of this macro call are identical to those specified for the generalized OPEN$x macro call (see section 3.1), with the exception of the facc parameter and the dfnb parameter. These additional parameters are described below.

To open a file without superseding an existing file having an identical file specification, a macro call of the following form is used:

```
OPEN$   fdb,facc,lun,dspt,dfnb,racc,urba,urbs,err
```

where:

facc
: represents any one or an appropriate combination of the following symbolic values indicating how the specified file is to be accessed:

  FO.RD - Indicates that an existing file is to be opened for reading only.

  FO.WRT - Indicates that a new file is to be created and opened for writing.

  FO.APD - Indicates that an existing file is to be opened and appended.

  FO.MFY - Indicates that an existing file is to be opened and modified.

  FO.UPD - Indicates that an existing file is to be opened, updated, and, if necessary, extended.

  FA.NSP - Indicates, in combination with FO.WRT above, that the old file having the same file specification is not to be superseded by the new file.

  FA.TMP - Indicates, in combination with FO.WRT above, that the file is to be a temporary file.

  FA.SHR - Indicates that the file is to be opened for shared access.

dfnb
: represents the symbolic address of the default filename block. This parameter is the same as that described in connection with the FDOP$A/FDOP$R macro call.

The above parameters initialize FDB offset locations F.FACC and F.DFNB with appropriate values.

Any logically consistent combination of the above file access symbols is permissible. The particular combination required to create and write a new file without superseding an existing file is shown below:

```
OPEN$   #OUTFDB,#FO.WRT!FA.NSP
```

The following macro call creates a temporary file for shared access:

```
OPEN$   #OUTFDB,#FO.WRT!FA.TMP!FA.SHR
```

3.8 CLOSE$ - CLOSE SPECIFIED FILE

When the processing of a file is completed, it must be closed by issuing the CLOSE$ macro call. The CLOSE$ operation performs the following housekeeping functions:

1. Waits for all I/O operations in progress for the file to be completed (multiple-buffered record I/O only).
2. Ensures that the FSR block buffer containing data for an output file is completely written if it is partially filled (record I/O only).
3. De-accesses the file.
4. Releases the FSR block buffer(s) allocated for the file (record I/O only).
5. Prepares the FDB for subsequent use by clearing appropriate FDB offset locations.
6. Calls an optional user-coded error-handling routine if an error condition is detected during the CLOSE$ operation.

3.8.1 Format of CLOSE$ Macro Call

The CLOSE$ macro call takes the following format:

```
CLOSE$  fdb,err
```

where:

fdb — represents the symbolic address of the associated FDB.

err — represents the symbolic address of an optional user-coded error-handling routine.

The following examples illustrate the use of the CLOSE$ macro call:

```
CLOSE$  #FDBIN,CLSERR

CLOSE$  ,CLSERR

CLOSE$  R0
```

The first example shows an explicit declaration for the relevant FDB and the symbolic address of an error-handling routine to be entered if the CLOSE$ operation is not completed successfully. The last two examples assume that R0 currently contains the address of the appropriate FDB.

3.9 GET$ - READ LOGICAL RECORD

The GET$ macro call is used to read logical records from a file. After a GET$ operation, the next record buffer descriptors in the FDB always identify the record just read, i.e., offset location F.NRBD+2 contains the address of the record just read, and offset location F.NRBD contains the size of that record (in bytes). This is true of GET$ operations in both move and locate mode.

In move mode, a GET$ operation moves a record to the user record buffer (as defined by the current contents of F.URBD+2 and F.URBD), and the address and size of that record are then returned to the next record buffer descriptors in the FDB (F.NRBD+2 and F.NRBD).

In locate mode, if the entire record resides within the FSR block buffer, then the address and the size of the record just read are returned to the next record buffer descriptors (F.NRBD+2 and F.NRBD). If, on the other hand, the entire record does not reside within the FSR block buffer, then that record is moved piecemeal into the user record buffer, and the address of the user record buffer and the size of the record are returned to offset locations F.NRBD+2 and F.NRBD, respectively.

After returning from a GET$ operation in locate mode, whether or not moving the record was necessary, F.NRBD+2 always contains the address of the record just read, and F.NRBD always contains the size of that record.

GET$ operations are fully synchronous, i.e., record I/O operations are completed before control is returned to the user program.

Specific FDB requirements for GET$ operations are presented in section 3.9.2 below.

3.9.1 Format of GET$ Macro Call

To read a logical record, the GET$ macro call is specified in the following format:

```
        GET$    fdb,urba,urbs,err
```

where:

- **fdb** represents the symbolic address of the associated FDB.

- **urba** represents the symbolic address of a user record buffer to be used for record I/O operations in move or locate mode. When specified, this parameter initializes FDB offset location F.URBD+2.

- **urbs** represents a numeric value defining the size (in bytes) of the user record buffer. This parameter determines the largest record that can be placed in the user record buffer in move or locate mode. When specified, this parameter initializes offset location F.URBD in the associated FDB.

- **err** represents the symbolic address of an optional user-coded error-handling routine.

If neither the urba nor the urbs parameter is specified in the GET$ macro call, FCS assumes that these requisite values have been supplied previously through the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call. Any non-zero values in offset locations F.URBD+2 and F.URBD resulting therefrom are used as the address and the length, respectively, of the user record buffer.

If either of the following conditions occurs during record I/O operations, FCS returns an error indication (IE.RBG) to offset location F.ERR of the FDB, indicating an illegal record size:

1. In move mode, the record size exceeds the limit specified in offset location F.URBD; or

2. In locate mode, the record size exceeds the limit specified in offset location F.URBD, and the record must be moved because it crosses block boundaries.

The following statements are representative of the GET$ macro call:

```
GET$    R0,,,ERROR

GET$    ,#RECBUF,#25.,ERROR

GET$    #INFDB
```

In the first example, the address of the desired FDB is assumed to be present in R0. Note that the next two parameters, i.e., the user record buffer address (urba) and the user record buffer size (urbs), are null. In this case, FCS assumes that the appropriate values for FDB offset locations F.URBD+2 and F.URBD, respectively, have been specified previously in the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call. The final parameter in the string is the symbolic address of a user-coded error-handling routine.

The second example also assumes that R0 contains the address of the desired FDB. Explicit parameters then define the address and the size, respectively, of the user record buffer.

The last example shows a GET$ macro call in which only the address of the FDB is specified.

3.9.2 FDB Mechanics Relevant to GET$ Operations

The following sections summarize the essential aspects of GET$ operations in move and locate mode with respect to the associated FDB.

The discussions below focus mainly on whether or not a user record buffer is required under certain conditions. In this regard, the reader should recall that the user record buffer descriptors, i.e., the urba and the urbs parameters, may be specified in the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call, as well as the I/O initiating GET$ macro call. These parameters need be present in the GET$ macro call (to appropriately initialize the FDB) only if not previously supplied through some other available means.

If operating in random mode, then the number of the record to be read is maintained by FCS in offset locations F.RCNM and F.RCNM+2 of the associated FDB. This value is incremented after each GET$ operation to point to the next record in the FSR block buffer. Thus, unless a different record number is explicitly specified before each issuance of the GET$ macro call, the next record in sequence is read. The specified user record buffer size (i.e., the urbs parameter) always determines the largest record that can be read during a GET$ operation.

3.9.2.1 GET$ Operations in Move Mode

With respect to GET$ operations in move mode, the following generalizations apply:

1. If records are always moved to the same user record buffer, the urba and urbs parameters need be specified only in the initial GET$ macro call. Alternatively, these values may be specified beforehand through any available means identified above for initializing the user record buffer descriptor cells in the FDB. In any case, offset locations F.URBD+2 and F.URBD remain appropriately initialized for all subsequent GET$ operations in move mode which involve the same user record buffer.

3.9.2.2 GET$ Operations in Locate Mode

In performing GET$ Operations in locate mode, the user should take into account the following:

1. If fixed-length records are to be processed, and if they fit evenly within the FSR block buffer, the user record buffer descriptors need not be present in the associated FDB.

2. If fixed-length records which do not fit evenly within the FSR block buffer are to be processed, or if variable-length records are to be processed, the user record buffer descriptors need not be present in the FDB, provided that the file being processed exhibits the attribute of records not being allowed to cross block boundaries (FD.BLK).

   The property of records not crossing block boundaries is established as the file is created. Specifically, if offset location F.RATT in the FDB is initialized with FD.BLK prior to file create-time, then the records in the resulting file will not be allowed to cross block boundaries.

   For an existing file, the user file attribute section of the file header block is read when the file is opened; thus, all attributes of that file are made known to FCS, including whether or not records within that file are allowed to cross block boundaries.

   The design of FCS requires the utilization of a user record buffer only in the event that records (either fixed or variable in length) cross block boundaries.

3. If a GET$ operation is performed in locate mode, and the record is contained entirely within the FSR block buffer, the address of the record within the FSR block buffer and the

size of that record are returned to offset locations F.NRBD+2 and F.NRBD, respectively, in the associated FDB. However, if that record crosses block boundaries, it is moved to the user record buffer. In this case, the address of the user record buffer and the size of the record are returned to offset locations F.NRBD+2 and F.NRBD, respectively.

In summary, if the potential exists for crossing block boundaries during GET$ operations in locate mode, then the user record buffer descriptors must be supplied through any available means to appropriately initialize offset locations F.URBD+2 and F.URBD in the associated FDB.

3.10 GET$R - READ LOGICAL RECORD IN RANDOM MODE

The GET$R macro call is used to read fixed-length records from a file in random mode. Thus, by definition, issuing this macro call requires that the user be intimately familiar with the structure of the file to be read and, furthermore, that he be able to specify precisely the number of the record to be read.

The GET$ and GET$R macro calls are identical, except that GET$R allows the specification of the desired record number. If the desired record number is already present in the FDB (at offset locations F.RCNM and F.RCNM+2), then GET$ and GET$R may be used interchangeably. If, however, the record access byte in the FDB (offset location F.RACC) has not been initialized for random-access operations with FD.RAN in the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call, then neither GET$ nor GET$R will read the desired record.

The GET$R macro call takes two more parameters in addition to those specified in the GET$ macro call, as shown below:

```
GET$R   fdb,urba,urbs,lrcnm,hrcnm,err
```

where:

lrcnm — represents a numeric value specifying the low-order 16 bits of the number of the record to be read. This value, which must be specified, is stored in offset location F.RCNM+2 in the FDB. The GET$R macro call seldom requires more than 16 bits to express the record number. A logical record number up to 65,536(10) may be specified through this parameter. If this parameter is not sufficient to completely express the magnitude of the record number, the following parameter must also be specified.

hrcnm — represents a numeric value specifying the high-order 15 bits of the number of the record to be read. This value is stored in FDB offset location F.RCNM. If specified, the combination of this parameter and the lrcnm parameter above determines the number of the desired record. Thus, an unsigned value having a total of 31 bits of magnitude may be used in defining the record number.

If this parameter is not specified, offset location F.RCNM retains its initialized value of zero (0).

If F.RCNM is used to express a desired record number for any given GET$R operation, this cell must be cleared before issuing a subsequent GET$R macro call that requires 16 bits or less to express the desired record number; otherwise, any residual value in F.RCNM will yield an incorrect record number.

If the lrcnm and hrcnm parameters are not specified in a subsequent GET$R macro call, the next sequential record is read, since the record number in offset locations F.RCNM+2 and F.RCNM is automatically incremented with each GET$ operation. In the case of the first GET$R after opening the file, record number one is read, because the record number has been initialized to zero by the OPEN. If other than the next sequential record is to be read, the user must explicitly specify the number of the desired record.

The following statements are representative of the use of the GET$R macro call:

```
        GET$R   #INFDB,#RECBUF,#160.,#1040.,,ERROR

        GET$R   #FDBADR,#RECBUF,#160.,R3
```

Note in the first example that the number of the desired record to be read, i.e., 1040(10), is expressed through the first of two available fields for this purpose; the second field is not required and is therefore reflected as a null specification.

The second example reflects the use of general register 3 in specifying the logical record number. This register, or any other location so used, must be preset with the desired record number before issuing the GET$R macro call.

3.11 GET$S - READ LOGICAL RECORD IN SEQUENTIAL MODE

The GET$S macro call is used to read logical records from a file in sequential mode. Although the routine invoked by the GET$S macro call requires less memory than that invoked by GET$ (see section 3.9), GET$S has the same format and takes the same parameters. The GET$S macro call is designed specifically for use in an overlaid environment where the amount of memory available to the program is limited and files are to be read in strictly sequential mode.

Note, if both GET$S and PUT$S are to be used by the program, that the savings in memory utilization over GET$ and PUT$ will be realized only if GET$S and PUT$S are placed on different branches of the overlay structure.

3.12 PUT$ - WRITE LOGICAL RECORD

The PUT$ macro call is used to write logical records to a file. For PUT$ operations, offset locations F.NRBD+2 and F.NRBD in the associated FDB must contain the address and the size, respectively, of the record to be written. The distinction between move mode and locate mode for PUT$ operations relates to the building or the assembling of the data into a record. Specifically, in move mode, the record is built in a buffer of the user's choice. This buffer is not necessarily the user record buffer previously described in the context of record I/O operations. In other words, the user may build records in an area of his program apart from that normally defined by the user record buffer descriptors in the FDB (F.URBD+2 and F.URBD). In this case, the address of the record buffer so used and the size of the record are specified in the PUT$ macro call, and the record thus built is then moved into the FSR block buffer.

In locate mode, however, the record is built at the address specified by the contents of offset location F.NRBD+2, and only the record size need be specified in the PUT$ macro call. Then, if the record so built is not already in the FSR block buffer, it is moved therein as the PUT$ operation is performed.

PUT$ operations are fully synchronous, i.e., record I/O operations are completed before control is returned to the user program.

A random PUT$ operation in locate mode requires the use of the .POSRC routine. This operation is described in detail in section 4.9.2.

Specific FDB requirements for PUT$ operations are presented in section 3.12.2 below.

3.12.1 Format of PUT$ Macro Call

The PUT$ macro call takes the following format:

```
        PUT$    fdb,nrba,nrbs,err
```

where:

- fdb — represents the symbolic address of the associated FDB.
- nrba — represents the symbolic address of the next record buffer, i.e., the address of the record to be PUT$. This parameter initializes FDB offset location F.NRBD+2.
- nrbs — represents a numeric value specifying the size of the next record buffer, i.e., the length of the record to be PUT$. This parameter initializes FDB offset location F.NRBD.
- err — represents the symbolic address of an optional user-coded error-handling routine.

The following examples are representative of the uses of the PUT$ macro call:

```
        PUT$    #FDBADR,,,ERRRT

        PUT$    ,,#160.,ERRRT

        PUT$    R0
```

In the first example, note that the next record buffer address (nrba parameter) and the next record buffer size (nrbs parameter) are null.

These null specifications imply that the current values in offset locations F.NRBD+2 and F.NRBD of the associated FDB are suitable to the current operation. Note also that fixed-length records could also be written in locate mode by issuing this macro call.

The second example contains null specifications in the first two parameter fields, assuming that R0 currently contains the address of the associated FDB and that variable-length records are to be written to the file.

Finally, the last example specifies only the address of the FDB; all other parameter fields are null.

3.12.2 FDB Mechanics Relevant to PUT$ Operations

The discussions below highlight those aspects of PUT$ operations in move and locate mode which have a bearing on the associated FDB.

The conditions under which a user record buffer is or is not used are summarized. As is the case for GET$ operations, if a user record buffer is required for PUT$ operations, the buffer descriptors (i.e., the urba and urbs parameters) may be supplied to the associated FDB through the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call. In any case, offset locations F.URBD+2 and F.URBD must be appropriately initialized if PUT$ operations require the utilization of a user record buffer. Note, however, that PUT$ operations in move mode never require a user record buffer.

If the user record buffer is required, the specified size of that buffer (i.e., the urbs parameter) always determines the size of the largest record that can be written to the specified file.

Whether in move or locate mode, a PUT$ operation uses the information in offset locations F.NRBD+2 and F.NRBD, i.e., the next record buffer descriptors, to determine whether the record must be moved into the FSR block buffer. In the event that the record does have to be moved, and the size of that record is such that it will not fit in the space remaining therein, one of two possible operations is performed:

1. If records are allowed to cross block boundaries, then the first part of the record is moved into the FSR block buffer, thereby completing a virtual block. That block buffer is then written out to the volume, and the remaining portion of the record is moved into the beginning of the next FSR block buffer.

2. If records are not allowed to cross block boundaries (because of the file attribute FD.BLK specified in the associated FDB), then the FSR block buffer is written out to the volume as is, and the entire record is moved into the beginning of the next FSR block buffer.

3.12.2.1 PUT$ Operations in Move Mode

A PUT$ operation in move mode is basically driven by specifying in each PUT$ macro call the address and the size of the record to be written. Then, as the PUT$ operation is performed, FCS moves the record into the appropriate area of the FSR block buffer.

In summary, the following generalizations apply for PUT$ operations in move mode:

1. The user record buffer descriptors need not be present in the FDB because the programmer is dynamically specifying the address and the length of the record to be written at each issuance of a PUT$ macro call. The values so specified dynamically update offset locations F.NRBD+2 and F.NRBD in the associated FDB.

2. If the file consists of the fixed-length records, then the generalized OPEN$x macro call (see section 3.1) will initialize offset location F.NRBD with the appropriate record size, as defined by the contents of offset location F.RSIZ. Thus, the size of the record need not be specified as the urbs parameter in any PUT$ macro call involving this file.

3. If variable-length records are being PUT$, the size of each record must be specified as the urbs parameter in each PUT$ macro call involving this file, thus setting offset location F.NRBD to the appropriate record size.

3.12.2.2 PUT$ Operations in Locate Mode

Basically, a user record buffer is required for PUT$ operations in locate mode only when the potential exists for records to cross block boundaries. In other words, if there is insufficient space in the FSR block buffer to accommodate the building of the next record, the user must provide a buffer in his own memory space in order to build that record.

When a file is initially opened for PUT$ operations in locate mode, FCS sets up offset location F.NRBD+2 to point to the area in the FSR block buffer where the next record is to be built. Then, each PUT$ operation thereafter in locate mode updates the address value in this cell to point to the area in the FSR block buffer where the next record is to be built. Thus, after each PUT$ operation in locate mode, F.NRBD+2 points to the area where the next record is to be built. This logic dictates whether the user record buffer is required in locate mode.

In this regard, the following generalizations apply:

1. If fixed-length records are being PUT$ and they fit evenly within the FSR block buffer, a user record buffer is not required.

2. If a fixed-length record crosses block boundaries, the user record buffer descriptors must be present in offset locations F.URBD+2 and F.URBD of the associated FDB. In this case, after determining that the record will not fit in the FSR block buffer, FCS sets offset location F.NRBD+2 to point to the user record buffer. Then, when the record is PUT$, it is moved from the user record buffer to the FSR block buffer.

3. If a variable-length record is being PUT$, the potential exists for crossing block boundaries. In this case, the user record buffer descriptors must be present in offset locations F.URBD+2 and F.URBD of the associated FDB. Moreover, the

size of each variable-length record must be specified as the nrbs parameter in each PUT$ macro call.

The determination as to whether FCS will point offset location F.NRBD+2 to the FSR block buffer for the PUT$ operation or to the user record buffer is based on whether there is potentially enough room in the FSR block buffer to accommodate the record.

Because the records are variable in length, it must be assumed that the largest possible record will be PUT$, as defined by the size of the user record buffer (F.URBD). Thus, if a record of this defined size will not fit in the space remaining in the FSR block buffer, FCS sets offset location F.NRBD+2 to point to the user record buffer.

Each PUT$ operation in locate mode sets up the FDB for the next PUT$. In other words, the specified record size is used by FCS as the worst case condition in determining whether sufficient space exists in the FSR to build the next record.

If variable-length records are being processed that are shorter than the largest defined record size, FCS may move records unnecessarily from the user record buffer to the FSR block buffer. For example, assume that the user has allocated a 132-byte record buffer. Assume further that the available remaining space in the FSR block buffer is less than 132 bytes. In this case, FCS will continue to point the user to his own record buffer for PUT$ operations, even if he continues to PUT$ very short (10- or 20-byte) records. Thus, some unavoidable movement of records takes place in locate mode.

If the largest record that the user intends to PUT$ is 80 bytes, for example, then the largest defined record size should not be specified as 132 bytes (or any length larger than that intended to be PUT$). Aside from having to allocate a smaller user record buffer, PUT$ operations in locate mode will be more efficient if this precaution is observed. Exercising care in this regard reduces the tendency to move records from the user record buffer to the FSR block buffer when they might otherwise be built directly in the FSR block buffer.

3.13 PUT$R - WRITE LOGICAL RECORD IN RANDOM MODE

The PUT$R macro call is used to write fixed-length records to a file in random mode. As noted in section 3.10 in connection with the GET$R macro call, operations on random access files require the user to be intimately familiar with the contents of such files. The PUT$R macro call likewise relies entirely on the user for the specification of the number of the record before a specified PUT$ operation can be performed. Since the usual purpose of a PUT$R operation is to update known records in a file, it is assumed that the user also knows the number of such records within the file.

The PUT$ and PUT$R macro calls are identical, except that PUT$R allows the specification of the desired record number. If the desired record number is already present in the FDB (at offset locations F.RCNM and F.RCNM+2), then PUT$ and PUT$R may be used interchangeably. However, if the record access byte in the FDB (offset location F.RACC) has not been initialized for random-access operations with FD.RAN in the FDRC$A, the FDRC$R, or the generalized OPEN$x macro call, then neither PUT$ not PUT$R will write the desired record.

The PUT$R macro call takes two more parameters in addition to those specified in the PUT$ macro call, as shown below:

```
PUT$R   fdb,nrba,nrbs,lrcnm,hrcnm,err
```

where:

lrcnm — represents a numeric value specifying the low-order 16 bits of the number of the record to be processed. This parameter serves the same purpose as the corresponding parameter in the GET$R macro call (see section 3.10), except that it identifies the record to be written.

hrcnm — represents a numeric value specifying the high-order 15 bits of the number of the record to be processed. This parameter serves the same purpose as the corresponding parameter in the GET$R macro call, except that it identifies the record to be written.

If this parameter is not specified, offset location F.RCNM retains its initialized value of zero (0).

If F.RCNM is used in expressing a desired record number for any given PUT$R operation, the user must clear this cell before issuing a subsequent PUT$R macro call that requires 16 bits or less in expressing the desired record number; otherwise, any residual value in F.RCNM results in an incorrect record number.

The lrcnm and hrcnm parameters initialize offset locations F.RCNM+2 and F.RCNM, respectively, in the associated FDB. If these values are not specified in a subsequent PUT$R macro call, the next sequential record is written, since FCS automatically increments the record number in these cells with each PUT$ operation. In the case of the first PUT$R after opening the file, record number one is written, because the record number has been initialized to zero by the OPEN. Note that this is true even if the file has been opened for an append (OPEN$A). If other than the next sequential record is to be written, the user must explicitly specify the number of the desired record.

A representative example of the use of the PUT$R macro call follows:

```
PUT$R   #OUTFDB,#RECBUF,,#12040.,,ERRLOC

PUT$R   #FDBADR,#RECBUF,,R4

PUT$R   #FDBADR,#RECBUF,,LRN
```

In the first example, the presence of "RECBUF" as the next record buffer address (nrba) parameter merely indicates that the user is specifying the address of the record. Although specifying this address repeatedly is unnecessary, it is not invalid. Normally, a buffer address is specified dynamically, since other PUT$ macro calls may be referencing different areas in memory; thus, the address of the record must be explicitly specified in each PUT$ macro call. Note also that the next record buffer size (nrbs) parameter is null, since this parameter is required only in the case of writing variable-length records. Also, the second of the two available parameters for defining the record number is null.

Note in the second and third examples that R4 and a memory location (LRN) are used to specify the logical record number. Such a specification assumes that the user has preset the desired record number in the referenced location.

A random PUT$ operation in locate mode requires the use of the .POSRC routine. This operation is described in detail in section 4.9.2.

3.14 PUT$S - WRITE LOGICAL RECORD IN SEQUENTIAL MODE

The PUT$S macro call is used to write logical records to a file in sequential mode. Although the routine invoked by the PUT$S macro call requires less memory than that invoked by PUT$ (see section 3.12), PUT$S has the same format and takes the same parameters. The PUT$S macro call is designed specifically for use in an overlaid environment Where the amount of memory available to the program is limited and files are to be written in strictly sequential mode.

Note, if both GET$S and PUT$S are to be used by the program, that the savings in memory utilization over GET$ and PUT$ will be realized only if GET$S and PUT$S are placed on different branches of the overlay structure.

3.15 READ$ - READ VIRTUAL BLOCK

The READ$ macro call is issued to read a virtual block of data from a device (e.g., a disk or DECtape). In addition, if certain optional parameters are specified in the macro call, status information is returned to the I/O status block (see section 2.8.2), and/or the program traps to a user-coded AST service routine at the completion of block I/O operations (see section 2.8.3).

In issuing the READ$ (or WRITE$) macro call, the user is responsible for synchronizing all block I/O operations. For this reason, the WAIT$ macro call is provided (see section 3.18), allowing the user to suspend program execution until a specified READ$/WRITE$ operation has been completed. When the WAIT$ macro call is issued in conjunction with a READ$ (or WRITE$) macro call, the user must ensure that the event flag number and the I/O status block address specified in both macro calls are the same.

3.15.1 Format of READ$ Macro Call

From the format below, note that the parameters of the READ$ macro call are identical to those of the FDBK$A or the FDBK$R macro call, with the exception of the fdb and err parameters. Certain FDB parameters may be set at assembly-time (FDBK$A), initialized at run-time (FDBK$R), or set dynamically by the READ$ macro call. In any case, certain information must be present in the FDB before the specified READ$ (or WRITE$) operation can be performed. These requirements are noted in section 3.15.2 below.

The READ$ macro call takes the following format:

```
        READ$   fdb,bkda,bkds,bkvb,bkef,bkst,bkdn,err
```

where: fdb represents the symbolic address of the associated FDB.

bkda — represents the symbolic address of the block I/O buffer in the user program. This parameter need not be specified if offset location F.BKDS+2 has been previously initialized through either the FDBK$A or the FDBK$R macro call.

bkds — represents a numeric value specifying the size (in bytes) of the virtual block to be read. This parameter need not be specified if offset location F.BKDS has been previously initialized through either the FDBK$A or the FDBK$R macro call. In any case, the maximum block size that may be specified for file-structured devices is 512(10) bytes, i.e., the size of one virtual block.

bkvb — represents the symbolic address of a 2-word block in the user program containing the number of the virtual block to be read. This parameter causes offset locations F.BKVB and F.BKVB+2 to be initialized with the virtual block number; F.BKVB+2 contains the low-order 16 bits of the virtual block number, and F.BKVB contains the high-order 15 bits.

As noted in connection with the FDBK$A macro call described in section 2.2.1.4, assembly-time initialization of the virtual block number in the FDB is ineffective, since the generalized OPEN$x macro call sets the virtual block number in the FDB to one (1). The virtual block number can be made available to FCS only through the FDBK$R macro call or the I/O-initiating READ$ (or WRITE$) macro call after the file has been opened. The virtual block number is created as described in Item 4 of section 2.2.2.1.

The READ$ function checks the specified virtual block number to ensure that it does not reference a non-existent block, i.e., a block beyond the end of the file. If the virtual block number references non-existent data, an end-of-file (IE.EOF) error indication is returned to the I/O status block (see bkst parameter below) and to offset location F.ERR of the associated FDB; otherwise, the READ$ operation proceeds normally.

If the virtual block number is not specified through any of the available means identified above, automatic sequential operation results by default, beginning with virtual block number 1. The virtual block number is incremented by one (1) automatically after each READ$ operation is performed.

bkef — represents a numeric value specifying the event flag number to be used for synchronizing block I/O operations. This event flag number is used by FCS to signal the completion of the specified block I/O operation. The event flag number, which may also be specified in either the FDBK$A or the FDBK$R macro call, initializes FDB offset location F.BKEF; if so specified, this parameter need not be included in the READ$ (or WRITE$) macro call.

If this optional parameter is not specified through any available means, event flag 32(10) is used by default.

The function of an event flag is discussed in further detail in section 2.8.1.

bkst represents the symbolic address of the I/O status block in the user program (see section 2.8.2). This parameter, which initializes offset location F.BKST, is optional. The I/O status block is filled in by the system when the requested block I/O transfer is completed, indicating the success/failure of the requested operation.

The address of the I/O status block may also be specified in either the FDBK$A or the FDBK$R macro call. If the address of this 2-word structure is not supplied to FCS through any of the available means, status information is not returned to the I/O status block. However, the event flag specified through the bkef parameter above is set to indicate block I/O completion, but the user program must assume that the operation was successful. An error indication cannot be returned to the user program without an I/O status block address.

bkdn represents the symbolic entry-point address of an AST service routine (see section 2.8.3). If this parameter is specified, a trap occurs upon completion of the specified READ$ (or WRITE$) operation. This parameter, which is optional, initializes offset location F.BKDN. This address value may also be made available to FCS through either the FDBK$A or the FDBK$R macro call, and, if so specified, need not be present in the READ$ (or WRITE$) macro call.

If the address of an AST service routine is not specified through any available means, no AST trap occurs at the completion of block I/O operations.

err represents the symbolic address of an optional user-coded error-handling routine.

The following examples are representative of READ$ macro calls that may be issued to accomplish a variety of operations:

```
READ$   R0

READ$   #INFDB,,,,,,,ERRLOC

READ$   R0,#INBUF,#BUFSIZ,,#22.,#IOSADR,#ASTADR,ERRLOC

READ$   #INFDB,#INBUF,#BUFSIZ,#VBNADR
```

The first example assumes that R0 contains the address of the associated FDB. Also, all other required FDB initialization has been accomplished through either the FDBK$A or the FDBK$R macro call.

The second example shows an explicit declaration of the associated FDB and includes the symbolic address of a user-coded error-handling routine.

In the third example, R0 again contains the address of the associated FDB. The block buffer address and the size of the block are specified next in symbolic form. The address of the 2-word block in the user program containing the virtual block number is not specified, as indicated by the additional comma in the parameter string. The event flag number, the address of the I/O status block, and the address of the AST service routine then follow in order. Finally, the symbolic address of an optional error routine is specified.

The fourth example reflects, as the last parameter in the string, the symbolic address of the 2-word block in the user program containing the virtual block number.

3.15.2 FDB Requirements for READ$ Macro Call

The READ$ macro call requires that the associated FDB be initialized with certain values before it can be issued. These values may be specified through either the FDBK$A or the FDBK$R macro call, or they may be made available to the FDB through the various parameters of the READ$ macro call. In any case, the following values must be present in the FDB to enable READ$ operations to be performed:

1. The block buffer address (in offset location F.BKDS+2);
2. The block byte count (in offset location F.BKDS); and
3. The virtual block number (in offset locations F.BKVB+2 and F.BKVB).

3.16 WRITE$ - WRITE VIRTUAL BLOCK

The WRITE$ macro call is issued to write a virtual block of data to a block-oriented device (e.g., a disk or DECtape). Like the READ$ macro call, if certain optional parameters are specified in the WRITE$ macro call, status information is returned to the I/O status block (see section 2.8.2), and, at the completion of the I/O transfer, the program traps to an AST service routine that is supplied to coordinate asynchronous block I/O operations (see section 2.8.3).

Whether or not the address of an AST service routine and/or an event flag number is supplied, the user is responsible for synchronizing all block I/O processing. Again, as with READ$ operations, the WAIT$ macro call can be issued in conjunction with the WRITE$ macro call to suspend program execution until a program-dependent I/O transfer has been completed. When the WAIT$ macro call is used for this purpose, the event flag number and the I/O status block address in both macro calls must be the same.

3.16.1 Format of WRITE$ Macro Call

The WRITE$ macro call takes the same parameters as the READ$ macro call, as shown below. However, the bkvb parameter, in this case, represents the number of the virtual block to be written. The virtual block number is incremented by one (1) automatically after each WRITE$ operation is performed.

The WRITE$ macro call has the following format:

```
WRITE$  fdb,bkda,bkds,bkvb,bkef,bkst,bkdn,err
```

When this macro call is issued, the virtual block number (i.e., the bkvb parameter) is checked to ensure that it references a block within the file's allocated space; if it does, the block is written. If the specified block is not within the file's allocated space, FCS attempts to extend the file. If this attempt is successful, the block is written; if not, an error code indicating the reason for the failure of the extend operation is returned to the I/O status block and to offset location F.ERR of the associated FDB.

If FCS determines that the file must be extended, the actual extend operation is performed synchronously. After the extend operation has been successfully completed, the WRITE$ operation is queued, and only then is control returned to the instruction immediately following the WRITE$ macro call.

The following examples illustrate representative WRITE$ macro calls:

```
WRITE$  R0

WRITE$  #OUTFDB,#OUTBUF,#BUFSIZ,#VBNADR,#22.

WRITE$  R0,,,,#22.,#IOSADR,#ASTADR,ERRLOC
```

The first example specifies only the FDB address and assumes that all other required values are present in the FDB. The second example reflects explicit declarations for the FDB, the block buffer address, the block buffer size, the virtual block number address, and the event flag number for signaling block I/O completion. The third example shows null specifications for three parameter fields, then continues with the event flag number, the address of the I/O status block, and the address of the AST service routine. Finally, the address of a user-coded error-handling routine is specified.

3.16.2 FDB Requirements for WRITE$ Macro Call

WRITE$ operations require the presence of the same information in the FDB as READ$ operations (see section 3.15.2 above).

3.17 DELET$ - DELETE SPECIFIED FILE

The DELET$ macro call causes the directory information for the file associated with the specified FDB to be deleted from the appropriate user file directory (UFD). The space occupied by the file is then deallocated and returned to the pool of available storage on the volume for reallocation.

This macro call can be issued for a file that is either open or closed. If issued for an open file, that file is then closed and deleted; if issued for a closed file, that file is deleted only if the filename string specified in the associated dataset descriptor or default filename block contains an explicit file version number.

Thus, if the file is not open, and the file version number is 0 (indicating the latest version), or if the file version number is -1 (indicating the oldest version), then the DELET$ operation will fail.

3.17.1 Format of DELET$ Macro Call

The DELET$ macro call takes the following format:

```
DELET$  fdb,err
```

where:

- fdb — represents the symbolic address of the associated FDB.
- err — represents the symbolic address of an optional user-coded error-handling routine.

The following statements are illustrative of DELET$ macro calls:

```
DELET$  R0

DELET$  #OUTFDB,ERRLOC

DELET$  R0,ERRLOC
```

3.18 WAIT$ - WAIT FOR BLOCK I/O COMPLETION

The WAIT$ macro call, which is issued only in connection with READ$ and WRITE$ operations, causes program execution to be suspended until the requested block I/O transfer is completed. This macro call may be used to synchronize a block I/O operation which depends on the successful completion of a previous block I/O transfer.

As noted in section 3.15 in connection with the READ$ macro call, the user may specify an event flag number through the bkef parameter. This event flag number is used during READ$ operations to indicate the completion of the requested transfer. If desired, the user may issue a WAIT$ macro call (specifying the same event flag number and I/O status block address) following the READ$ (or WRITE$) macro call.
In this case, the READ$ operation is initiated in the usual manner, but the Executive of the host operating system suspends program execution until the specified event flag is set, indicating that the I/O transfer has been completed. The system then returns information to the I/O status block, indicating the success/failure of the operation. FCS then moves the I/O status block success/failure indicator into offset location F.ERR of the associated FDB, and returns with the C-bit in the Processor Status Word cleared if the operation is successful, or set if the operation is not successful. Task execution then continues with the instruction immediately following the WAIT$ macro call.

The system returns the final status of the I/O operation to the I/O status block (see section 2.8.2) upon completion of the requested operation. A positive value (+) indicates successful completion, and a negative value (-) indicates unsuccessful completion.

Event flags are discussed in further detail in section 2.8.1.

3.18.1 Format of WAIT$ Macro Call

The WAIT$ macro call is specified in the following format:

```
WAIT$   fdb,bkef,bkst,err
```

where:

fdb — represents the symbolic address of the associated FDB.

bkef — represents a numeric value specifying the event flag number to be used for synchronizing block I/O operations. The WAIT$ macro causes task execution to be suspended by invoking the WAITFOR system directive. This parameter must agree with the corresponding (bkef) parameter in the associated READ$/WRITE$ macro call.

If this parameter is not specified, either in the WAIT$ macro call or the associated READ$/WRITE macro call, FDB offset location F.BKEF is assumed to contain the desired event flag number, as previously initialized through the bkef parameter of the FDBK$A or the FDBK$R macro call.

bkst — represents the symbolic address of the I/O status block in the user program (see section 2.8.2). Although this parameter is optional, if specified, it must agree with the corresponding (bkst) parameter in the associated READ$/WRITE$ macro call.

If this parameter is not specified, either in the WAIT$ macro call or the associated READ$/WRITE$ macro call, FDB offset location F.BKST is assumed to contain the address of the I/O status block, as previously initialized through the bkst parameter of the FDBK$A or the FDBK$R macro call. If F.BKST has not been initialized, no return of information to the I/O status block occurs.

err — represents the symbolic address of an optional user-coded error-handling routine.

The following statements are representative of WAIT$ macro calls:

```
WAIT$   R0

WAIT$   #INFDB,#25.

WAIT$   R0,#25.,#IOSTAT

WAIT$   R0,,#IOSTAT,ERRLOC
```

The first example assumes that R0 contains the address of the associated FDB; furthermore, since the event flag number (bkef parameter) is not specified, offset location F.BKEF is assumed to contain the desired event flag number. If this cell in the FDB contains zero (0), event flag number 32(10) is used by default.

The second example shows an explicit specification of the FDB address and also specifies 25(10) as the event flag number. Again, in this example, the FDB is assumed to contain the address of the I/O status block. In contrast, the third example shows an explicit specification for the address of the I/O status block.

Finally, the fourth example contains a null specification for the event flag number, and, in addition, specifies the address of a user-coded error-handling routine.

It should be noted that the WAIT$ macro call associated with a given READ$ or WRITE$ operation need not be issued immediately following the macro call to which it applies. For example, the following sequence is typical:

1. Issue the desired READ$ or WRITE$ macro call.

2. Perform other processing that is not dependent on the completion of the requested block I/O transfer.

3. Issue the WAIT$ macro call.

4. Perform the processing that is dependent on the completion of the requested block I/O transfer.

When performing multiple asynchronous transfers in the same general sequence as above, a separate buffer, I/O status block, and event flag must be maintained for each operation. If the user intends to wait for the completion of a given transfer, the appropriate event flag number and I/O status block address must be specified in the associated WAIT$ macro call.

CHAPTER 4

FILE CONTROL ROUTINES

File control routines can be invoked in MACRO-11 programs to perform the following functions:

- Read or write default directory string descriptors in $$FSR2;
- Read or write the default file protection word in $$FSR2;
- Read or write the file owner word in $$FSR2;
- Convert a directory string from ASCII to binary, or vice versa;
- Find, insert, or delete a directory entry;
- Set a pointer to a byte within a virtual block or to a record within a file;
- Mark a place in a file for a subsequent OPEN$x operation;
- Issue an I/O command and wait for its completion;
- Rename a file;
- Extend a file;
- Mark a temporary file for deletion;
- Delete a file by filename block;
- Place directory information in a default filename block or a filename block;
- Perform device-specific control functions.(1)

---

(1) Does not apply to RSX-11M

4.1 CALLING FILE CONTROL ROUTINES

The CALL macro is used to invoke file control routines. These routines are included from the system object library (SY:[1,1]SYSLIB.OLB) at task-build time and incorporated into the user task. The file control routines are called as shown below:

```
CALL    .RDFDR

CALL    .EXTND
```

Before the CALL macro is issued, certain file control routines require that specific registers be preset with requisite information. These requirements are identified in the respective descriptions of the routines. Upon return, all registers are preserved, except those explicitly specified as changed.

As a general rule, if an error is detected by a file control routine, the C-bit (carry condition code) in the Processor Status Word is set, and an error indication is returned to FDB offset location F.ERR. However, certain file control routines do not return error indications because of the specific nature of their functions. The following file control routines are listed according to whether or not they return error indications.

Normal Error Return (C-bit and F.ERR)	No Error Return
.ASCPP	.RDFDR
.PARSE	.WDFDR
.PRSDV	.RDFFP
.ASLUN	.WDFFP
.FIND	.RFOWN
.ENTER	.WFOWN
.REMOV	.PPASC
.GTDIR	.MARK
.GTDID	
.POINT	
.POSRC	
.POSIT	
.XQIO	
.RENAM	
.EXTND	
.MRKDL	
.DLFNB	
.CTRL(1)	

Appendix I lists the error indicators that are placed in FDB offset location F.ERR by the routines identified above.

---

(1) Does not apply to RSX-11M

4.2 DEFAULT DIRECTORY STRING ROUTINES

The following routines are used to read and write directory string descriptors.

4.2.1 .RDFDR - Read $$FSR2 Default Directory String Descriptor

The user calls the .RDFDR routine to read the default directory string descriptor words from program section $$FSR2 of the FSR. These descriptor words define the address and the length of an ASCII string which contains the default directory string. This directory string constitutes the default directory that is to be used by FCS when one is not explicitly specified in a dataset descriptor.

Unless the user explicitly changes the default directory string descriptor words in $$FSR2 through the .WDFDR routine below, the default directory for a task will always correspond to the UIC under which the task is running.

When called, the .RDFDR routine returns the default directory string descriptor words to the following registers:

R1 Contains the size (in bytes) of the default directory string in $$FSR2.

R2 Contains the address of the default directory string in $$FSR2.

4.2.2 .WDFDR - Write New $$FSR2 Default Directory String Descriptor

The .WDFDR routine is called to create new default directory string descriptor words in $$FSR2. For example, if a user program is to operate on files in the directory [220,220], regardless of the UIC the program runs under, then the user may change the default directory string descriptor cells in $$FSR2 to point to the alternate directory string [220,220] created elsewhere in the program. To do this, the desired directory string is first created through an .ASCII directive. Then, by calling the .WDFDR routine, the default directory string descriptor cells in $$FSR2 are modified to point to the new directory string.

Assume that the task is currently running under default UIC [200,200]. By issuing a MACRO-11 directive similar to the following:

```
NEWDDS: .ASCII  /[220,220]/
```

a new directory string is defined. Then, by calling the .WDFDR routine, the user can modify the string descriptor cells in $$FSR2 to point to the new directory string. Thus, the default directory string in $$FSR2 remains intact; only the string descriptors within $$FSR2 are changed.

The following registers must be preset before calling the .WDFDR routine:

R1 Must contain the size (in bytes) of the new directory string.

R2 Must contain the address of the new directory string.

NOTE

Changing the default directory string descriptor words in $$FSR2 does not change the default UIC in $$FSR2 or the task's privileges.

4.3 DEFAULT FILE PROTECTION WORD ROUTINES

The routines described below are used to read and write the default file protection word in a location in program section $$FSR2 of the file storage region (FSR). This word is used only at file-creation time (e.g., by the OPEN$W macro call) to establish the default file protection values for the new file. Unless altered, this value constitutes the default file protection word for that file. If the value is minus one (-1), it indicates that the volume default file protection value, as established through the INITIALIZE, INITVOLUME, or MOUNT command, is to be used for the new file. The IAS User's Guide, RSX-11D User's Guide, and RSX-11M Operator's Procedures Manual, respectively, describe these initialization commands in detail.

The default file protection word has the following format:

Bits 15 12	11 8	7 4	3 0
WORLD	GROUP	OWNER	SYSTEM

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

Bit 3	2	1	0
DELETE	EXTEND	WRITE	READ

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.

4.3.1 .RDFFP - Read $$FSR2 Default File Protection Word

The user calls the .RDFFP routine to read the default file protection word in program section $$FSR2 of the FSR. No registers need be set before calling this routine.

When called, the .RDFFP routine returns the following information:

R1 Contains the default file protection word from $$FSR2.

4.3.2 .WDFFP - Write New $$FSR2 Default File Protection Word

The .WDFFP routine is used to write a new default file protection word into $$FSR2.

The following register must be preset before calling this routine:

R1 Must contain the new default file protection word to be written into $$FSR2. If this register is set to minus one (-1), the default file protection values established through the INITIALIZE, INITVOLUME, or MOUNT command will be used in creating all subsequent new files.

4.4 FILE OWNER WORD ROUTINES

The file owner word, like the default file protection word above, is a location in program section $$FSR2 of the FSR. The file owner word is used only at file-creation time (e.g., by the OPEN$W macro call) to establish the owner of the new file.

Normally, the file owner word contains the default UIC under which the task is running. However, through the .WFOWN routine (see section 4.4.2 below), the file owner word can be changed, if desired, so that any new files then created by the user program will have the desired UIC.

The format of the file owner word is shown below:

15 8	7 0
GROUP	MEMBER

The routines for reading and writing the file owner word are described below.

NOTE

The UIC and the file protection word for the file (see section 4.3) must not be set such that the UIC under which the task is running does not have access to the file. If this condition prevails, a privilege violation will result.

4.4.1 .RFOWN - Read $$FSR2 File Owner Word

The .RFOWN routine is used to read the file owner word from a location in $$FSR2. No registers need be preset before calling this routine.

When called, the .RFOWN routine returns the following information:

R1 Contains the file owner word (UIC).

4.4.2 .WFOWN - Write New $$FSR2 File Owner Word

The .WFOWN routine is used to write a new file owner word into $$FSR2.

The following register must be preset before calling this routine:

R1 Must contain the new file owner word to be written into $$FSR2.

4.5 ASCII/BINARY UIC CONVERSION ROUTINES

The following routines are called to convert a directory string from ASCII to binary, or vice versa.

4.5.1 .ASCPP - Convert ASCII Directory String to Equivalent Binary UIC.

The .ASCPP routine is called to convert an ASCII directory string to its corresponding binary UIC.

The following registers must be preset before calling this routine:

R2 Must contain the address of the directory string descriptor in the user program (see section 2.4.1) for the string to be converted.

R3 Must contain the address of a word location in the user program to which the binary UIC is to be returned. The member number is stored in the low-order byte of the word, and the group number is stored in the high-order byte.

4.5.2 **.PPSAC - Convert UIC to ASCII directory string.** The .PPASC routine is called to convert a binary UIC to its corresponding ASCII directory string.

The following registers must be preset before calling this routine:

R2 Must contain the address of a storage area within the user program into which the ASCII string is to be placed. The resultant string can be up to 9 bytes in length, e.g., [200,200].

R3 Must contain the binary UIC value to be converted. The low-order byte of the register contains the member number, and the high-order byte of the register contains the group number.

R4 Must contain a control code. Bits 0 and 1 of this register indicate the following:

Bit 0 is set to 0 to suppress leading zeros (e.g., 001 is returned as 1). Bit 0 is set to 1 to indicate that leading zeros are not to be suppressed.

Bit 1 is set to 0 to place separators in the directory string (e.g., [10,20]. Bit 1 is set to 1 to suppress separators (e.g., 1020).

The .PPASC routine increments the contents of R2 to point to the byte immediately following the last byte in the converted directory string.

NOTE

IAS and RSX-11D only: For a discussion of UIC's and UFD's, see the IAS or RSX-11D User's Guide.

4.6 FILENAME BLOCK ROUTINES

Two routines are available for performing functions related to a specified filename block. These routines are described in the following sections.

4.6.1 .PARSE - Fill In All Filename Information

When called, the .PARSE routine first zeros the filename block pointed to by R1 and then stores the following information in the filename block:

1. The ASCII device name (N.DVNM);

2. The binary unit number (N.UNIT);

3. The directory ID (N.DID);

4. The Radix-50 filename (N.FNAM);

5. The Radix-50 file type or extension (N.FTYP); and

6. The binary file version number (N.FVER).

The format of a filename block is shown in detail in Appendix B.

Before the .PARSE routine can be called, the FINIT$ macro call (see section 2.6) must be invoked explicitly in the user program, or it must be invoked implicitly through a prior OPEN$x macro call. Note, however, that the FINIT$ call must be issued only once in the initialization section of the program, i.e., the FINIT$ operation must be performed only once per task execution. Furthermore, FORTRAN programs issue a FINIT$ call at the beginning of task execution; therefore, MACRO-11 routines used with the FORTRAN object time system must not issue a FINIT$ macro call.

The following registers must be preset before calling the .PARSE routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of the filename block to be filled in. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0 (i.e., R0 + F.FNB).

R2 If .PARSE is to access a dataset descriptor in building the specified filename block, this register must contain the address of the desired dataset descriptor. This structure is usually, but not necessarily, the same as that associated with the FDB specified in R0, i.e., the dataset descriptor pointed to by the address value in F.DSPT.

If R2 contains zero (0), this value implies that a dataset descriptor has not been defined; therefore, the dataset descriptor logic of .PARSE is bypassed.

R3 If .PARSE is to access a default filename block in building the specified filename block, this register must contain the address of the desired default filename block. This structure is usually, but not necessarily, the same as that associated with the FDB specified in R0, i.e., the default filename block pointed to by the address value in F.DFNB.

As above, if R3 contains zero (0), this value implies that a default filename block has not been defined; therefore, the default filename block logic of .PARSE is bypassed.

Thus, R0 and R1 each must contain the address of the appropriate data structure, while either R2 or R3 must contain the address of the desired filename information. Both R2 and R3, however, may contain address values if the referenced structures both contain information required in building the specified filename block.

The .PARSE routine fills in the specified filename block in the order described in the following sections.

4.6.1.1 Device and Unit Information

The .PARSE routine first attempts to fill in the filename block with device (N.DVNM) and unit (N.UNIT) information. The following operations are performed in sequence until the required information is obtained from the specified data structures:

1. If the address of a dataset descriptor is specified in R2 and this structure contains a device string, the device and unit information therein is moved into the specified filename block.

2. If Step 1 fails, and if the address of a default filename block is specified in R3, and this structure contains a non-zero value in the device name field, the device and unit information therein is moved into the specified filename block.

3. If Step 2 fails, .PARSE uses the device and unit currently assigned to the logical unit number in offset location F.LUN of the specified FDB in building the filename block.

This feature allows a program to use pre-assigned logical units which are assigned through either the device assignment (ASG) option of the Task Builder or one of the following commands: the ASSIGN (under IAS) or the REASSIGN (under RSX). In this case, the user simply avoids specifying the device string in the dataset descriptor and the device name in the default filename block.

4. If the logical unit number in F.LUN is currently unassigned, .PARSE assigns this number to the system device (SY0:).

Once the device and unit are determined and the logical unit number is assigned, .PARSE invokes the GLUN$ directive to obtain necessary device information. Requisite information is returned to the following offsets in the filename block pointed to by R1:

N.DVNM - Device Name Field. Contains the redirected device name.

N.UNIT - Unit Number Field. Contains the redirected unit number.

In addition, requisite information is returned to the following offsets in the FDB pointed to by R0:

F.RCTL - Device Characteristics Byte. This cell contains device-dependent information from the first byte of the third word returned by the GLUN$ directive. The bit definitions pertaining to the device characteristics byte are described in detail in Table A-1. If desired, the user can examine this cell in the FDB to determine the characteristics of the device associated with the assigned LUN.

F.VBSZ - Device Buffer Size Word. This location contains the information from the sixth word returned by the GLUN$ directive. The value in this cell defines the device buffer size (in bytes) pertaining to the device associated with the assigned LUN.

The GLUN$ directive is described in detail in the Executive Reference Manual of the host operating system.

4.6.1.2 Directory Identification Information

Following the operations described in the preceding section, .PARSE attempts to fill in the filename block with directory identification information (N.DID), as follows:

1. If the address of a dataset descriptor is specified in R2 and this structure contains a directory string, that directory string is used to find the associated UFD in the MFD, and the resulting file ID is then moved into the directory ID field of the specified filename block.

2. If Step 1 fails, and if the address of a default filename block is specified in R3, and this structure contains a non-zero directory ID, it is moved into the specified filename block.

Since none of the parameters of the NMBLK$ macro call (see section 2.4.2) initialize the three words starting at offset location N.DID in the default filename block, these cells must be initialized manually, or they must be initialized by issuing a call to either the .GTDIR routine (see section 4.8.1) or the .GTDID routine (see section 4.8.2). Note that these routines can also be used to initialize a specified filename block directly with required directory information.

3. If neither Step 1 nor Step 2 yields the required directory string, .PARSE uses the default directory string in $$FSR2 to obtain the directory ID in the same manner as described in Step 1 above. The default directory string is set initially to correspond to the UIC under which the task is running.

4.6.1.3 Filename, File Type or Extension, and File Version Information

Following the operations described in the preceding section, .PARSE attempts to obtain filename information (N.FNAM, N.FTYP, and N.FVER), as follows:

1. If the address of a dataset descriptor is specified in R2 and this structure contains a filename string, the filename information therein is moved into the specified filename block.

2. If the address of a default filename block is specified in R3, and one or more of the filename, file type or extension, and file version number fields of the dataset descriptor specified in R2 are null, then the corresponding fields of the default filename block are used to fill in the specified filename block.

3. If neither Step 1 nor Step 2 yields the requisite filename information, any specific field(s) not available from either source remain(s) null.

NOTE

If a dot (.) appears in the filename string without an accompanying file type designation (e.g., TEST. or TEST.;3), the file type is interpreted as being explicitly null. In this case, the default file type is not used. Similarly, if a semicolon (;) appears in the filename string without an accompanying file version number (e.g., TEST.DAT;), the file version number is likewise interpreted as being explicitly null; again, the default file version number is not used in this case.

4.6.1.4 Other Filename Block Information

Finally, after performing all the operations above, the .PARSE routine also fills in the filename block status word (offset location N.STAT) of the filename block specified in R1. The bit definitions for this word are presented in Table B-2. Note in this table that an "explicit" directory, device, filename, file type, or file version number specification pertains to ASCII data supplied through the dataset descriptor pointed to by R2.

In addition, .PARSE explicitly zeros offset location N.NEXT in the filename block pointed to by R1. This action has implications for wildcard operations, as described in section 4.7.1 below.

4.6.2 .PRSDV - Fill in Device and Unit Information Only

The .PRSDV routine is identical to the .PARSE routine above, except that it performs only those operations associated with requisite device and unit information (see section 4.6.1.1). This routine zeros the filename block pointed to by R1, performs a .PARSE operation on the device and unit fields in the specified dataset descriptor and/or default filename block, and assigns the logical unit number contained in offset location F.LUN of the specified FDB.

4.6.3 .ASLUN - Assign Logical Unit Number

The .ASLUN routine is called to assign a logical unit number to a specified device and unit and to return the device information to a specified FDB and filename block.

The following registers must be preset before calling this routine:

- R0 Must contain the address of the desired FDB.
- R1 Must contain the address of the filename block containing the desired device and unit. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

If the device name field (offset location N.DVNM) of the filename block pointed to by R1 contains a non-zero value, the specified device and unit are assigned to the logical unit number contained in offset location F.LUN in the FDB pointed to by R0.

If N.DVNM in the filename block contains zero (0), then the device and unit currently assigned to the specified logical unit number are returned to the appropriate fields of the filename block.

Finally, if the specified logical unit number is not assigned to a device, the .ASLUN routine assigns it to the system device (SY0:) by default.

The information returned to the specified filename block and to the specified FDB is identical to that returned by the device and unit logic of the .PARSE routine (see section 4.6.1.1).

4.7 DIRECTORY ENTRY ROUTINES

The following routines are used to find, insert, and delete directory entries. The term "directory entry" encompasses entries in both the master file directory (MFD) and the user file directory (UFD).

4.7.1 .FIND - Locate Directory Entry

The .FIND routine is called to locate a directory entry by filename and to fill in the file identification field (N.FID) of a specified filename block.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

When invoked, the .FIND routine searches the directory file specified by the directory ID field of the filename block. This file is searched for an entry that matches the specified filename, file type, and file version number. In this regard, two special file versions are defined:

Version 0 is matched by the latest (largest) version number encountered in the directory file.

Version -1 is matched by the oldest (smallest) version number encountered in the directory file.

If either of these special versions is specified in the filename block, the matching version number is returned to the filename block. In this way, the actual version number is made available to the program.

Certain wildcard operations require the use of the .FIND routine. Three bits in the filename block status word (see N.STAT in Table B-2) indicate whether a wildcard (*) was specified for a filename, a file type, or a file version number field. If the wildcard bit in N.STAT is set for a given field, any directory entry will match in that corresponding field. Thus, if the filename and file version number fields contain wildcard specifications (*), and the file type field is specified as .OBJ (i.e., *.OBJ;*), the first directory entry encountered that contains .OBJ in the file type field will match, irrespective of the values present in the other two fields.

When a wildcard match is found, the complete filename, file type, and file version number fields of the matching entry are returned to the filename block, along with the file ID field (N.DID). Thus, the program can determine the actual name of the file just found. Offset location N.NEXT in the filename block is also set to indicate where that directory entry was found in the directory file. This information is used in subsequent .FIND operations to locate the next matching entry in the directory file.

For example, the .FIND routine is often used to open a series of files when wildcard specifications are used. The following operations are typical:

1. Call the .PARSE routine. This routine zeros offset location N.NEXT in the filename block in preparation for the iterative .FIND operations described in Step 3 below.

2. Check for wildcard bits set by the .PARSE routine in the filename block status word (see N.STAT in Table B-2). An instruction sequence such as that shown below may be used to test for the setting of wildcard bits in N.STAT:

```
        BIT     #NB.SVR!NB.STP!NB.SNM,N.STAT(R1)

        BEQ     NOWILD          ;BRANCH IF NOT SET.
```

3. If wildcard specifications are present in the filename block status word, repeat the following sequence until all the desired wildcard files have been processed:

```
        CALL    .FIND

        BCS     DONE            ;ERROR CODE IE.NSF INDICATES
                                ;NORMAL TERMINATION.

        OPEN$
```

   Wildcard .FIND operations update offset location N.NEXT in the filename block. In essence, the contents of this cell provide the necessary information for continuing the search of the directory file for a matching entry.

4. Perform the desired operations on the file.

NOTE

The above procedure applies only for the following types of wildcard file specifications:

```
TEST.DAT;*
TEST.*;*
*.DAT;*
```

The procedure does not work for the following types of wildcard file specifications:

```
*.DAT
TEST.*
```

In summary, if a wildcard file specification is present in either the filename field or the file type field, the file version number field must also contain either an explicit wildcard specification (*) or a specific file version number (e.g., 2, 3, etc.). In the latter case, however, the version number cannot be zero (0), for the latest version of the file, or minus one (-1), for the oldest version of the file.

4.7.2 .ENTER - Insert Directory Entry

The .ENTER routine is used to insert an entry by filename into a directory.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

If the file version number field of the filename block contains zero (0), indicating a default version number, the .ENTER routine scans the entire directory file to determine the current highest version number for the file. If a version number for the file is found, this entry is incremented to the next higher version number; otherwise, it is set to one (1). The resulting version number is returned to the filename block, making this number known to the program.

NOTE

Wildcard specifications cannot be used in connection with .ENTER operations.

4.7.3 .REMOV - Delete Directory Entry

The .REMOV routine is called to delete an entry from a directory by filename. This routine only deletes a specified directory entry; it does not delete the associated file.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

Wildcard specifications operate in the same manner as for the .FIND routine described in section 4.7.1 above, except that the special file version numbers zero (0) and minus one (-1) are illegal. The file version number for .REMOV operations must be explicit or wildcard. Each .REMOV operation deletes the next directory entry having the specified filename, file type, and file version number.

4.8 FILENAME BLOCK ROUTINES

The following routines are used to insert directory information in a specified filename block.

4.8.1 .GTDIR - Insert Directory Information in Filename Block

The .GTDIR routine is called to insert directory information taken from a directory string descriptor into a specified filename block.

Before calling this routine, the following registers must be preset:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block in which the directory information is to be placed. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

R2 Must contain the address of the 2-word directory string descriptor in the user program. This string descriptor defines the size and the address of the desired directory string.

This routine performs a .FIND operation for the specified user file directory (UFD) in the master file directory (MFD) and returns the resulting directory ID to the three words of the specified filename block, starting at offset location N.DID. The .GTDIR routine preserves the information in offset locations N.FNAM, N.FYTP, N.FVER, N.DVNM, and N.UNIT of the filename block, but zeros (clears) the rest of the filename block.

The .GTDIR routine can also be used in conjunction with the NMBLK$ macro call (see section 2.4.2) to insert directory information into a specified default filename block.

4.8.2 .GTDID - Insert Default Directory Information in Filename Block

The .GTDID routine provides an alternate means for inserting directory information into a specified filename block. Instead of allowing the specification of the directory string, as in the .GTDIR routine above, this routine uses the UIC in the default file owner word in $$FSR2 as the desired user file directory (UFD).

Before calling this routine, the following registers must be preset:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block in which the directory information is to be placed. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

When called, the .GTDID routine takes the UIC from the default file owner word in $$FSR2 and performs a .FIND operation for the associated user file directory (UFD) in the master file directory (MFD). The resulting directory ID is returned to the three words of the specified filename block, starting at offset location N.DID. As with the .GTDIR routine, .GTDID preserves offset locations N.FNAM, N.FTYP, N.FVER, N.DVNM, and N.UNIT in the filename block, but zeros the rest of the filename block.

The .GTDID routine embodies considerably less code than the .GTDIR routine, since it does not invoke the .PARSE logic; furthermore, .GTDID is intended specifically for use in programs which open files via the OFNB$ macro call (see section 3.6). Such a program does not invoke the .PARSE logic because all required filename information is provided to the program in filename block format.

As is true of the .GTDIR routine described in section 4.8.1 above, .GTDID may also be used in conjunction with the NMBLK$ macro call (see section 2.4.2) to insert directory information (N.DID) into a specified default filename block. The user also has the option to initialize offset location N.DID manually with required directory information.

4.9 FILE POINTER ROUTINES

The following routines are used to point to a byte or a record within a specified file.

4.9.1 .POINT - Position File to Specified Byte

The .POINT routine is called to position a file to a specified byte in a specified virtual block. If locate mode is in effect for record I/O operations, the .POINT routine also updates the value in offset location F.NRBD+2 in the associated FDB in preparation for a PUT$ operation in locate mode.

The following registers must be preset before calling this routine:

- R0 Must contain the address of the desired FDB.
- R1 Must contain the high-order bits of the virtual block number.
- R2 Must contain the low-order bits of the virtual block number.
- R3 Must contain the desired byte number within the specified virtual block.

For a description of virtual block numbers and how these 2-word values are formed, refer to Item 4 in section 2.2.2.1.

The .POINT routine is often used in conjunction with the .MARK routine to achieve a limited degree of random access with variable-length records. The .MARK routine saves the positional context of a file in anticipation of temporarily closing that file and then re-opening it later at the same position. For such purposes, the following procedure applies:

1. Call the .MARK routine (see section 4.9.3 below) to save the current positional context of the file.
2. Close the file.

3. When desired, re-open the file.

4. Load the information returned by the .MARK routine into R1, R2, and R3, as required above, before calling the .POINT routine.

5. Call the .POINT routine.

6. Resume processing of the file.

4.9.2 .POSRC - Position File to Specified Record

The .POSRC routine is called to position a file to a specified fixed-length record within a file. If locate mode is in effect for record I/O operations, the .POSRC routine also updates the value in offset location F.NRBD+2 in the associated FDB in preparation for a PUT$ operation in locate mode.

Before calling this routine, the user must set offset locations F.RCNM+2 and F.RCNM in the FDB to the desired record number and ensure that the correct record size is reflected in offset location F.RSIZ of the FDB.

Also, the register below must be preset before calling the .POSRC routine:

R0 Must contain the address of the associated FDB.

The .POSRC routine is used when performing random access PUT$ operations in locate mode. Normally, PUT$ operations in locate mode are sequential; however, when random access mode is used, the following procedure must be performed to ensure that the record is built at the desired location:

1. Set offset locations F.RCNM+2 and F.RCNM in the associated FDB to the desired record number.

2. Call the .POSRC routine.

3. Build the new record at the address returned (by the .POSRC call) in offset location F.NRBD+2 of the associated FDB.

4. Perform the PUT$ operation.

4.9.3 .MARK - Save Positional Context of File

The .MARK routine allows the user to record the current positional context of a file for later use. For example, the user may mark the current position of the file, close that file, and later re-open the file and return to the same position within the file. The .MARK routine is also useful in altering records within a file. After determining the record to be altered, the user may .MARK the file and retrieve information elsewhere in the file for use in updating the desired record. Then, by returning to the saved position of the file, the desired record may be altered. This iterative sequence may be repeated any number of times to update desired records in the file.

The register below must be preset before calling this routine:

R0 Must contain the address of the associated FDB.

When called, the .MARK routine returns information to the following registers:

R1 Contains the high-order bits of the virtual block number.

R2 Contains the low-order bits of the virtual block number.

R3 Contains the number of the next byte within the virtual block.

R3 points to the next byte in the block. For example, if four GET$ operations are performed, followed by a call to the .MARK routine, R3 points to the first byte in the fifth record in the file.

4.9.4 .POSIT - Return Positional Information for Specified Record

The .POSIT routine calculates the virtual block number and the byte number pertaining to the beginning of a specified record.

The following register must be preset before calling this routine:

R0 Must contain the address of the associated FDB.

In addition, offset locations F.RCNM and F.RCNM+2 in the associated FDB must contain the desired record number.

Unlike the .POSRC routine above, which positions the file to the specified record, .POSIT simply calculates the positional information for a specified record so that a .POINT operation can be later performed to position to the desired record.

The register values returned by the .POSIT routine are identical to those described above for the .MARK routine.

4.10 QUEUE I/O FUNCTION ROUTINE (.XQIO)

The .XQIO routine is called to execute a specified QUEUE I/O function and to wait for its completion.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the desired QUEUE I/O function code. Refer to the IAS/RSX-11D Device Handlers Reference Manual or the RSX-11M I/O Drivers Reference Manual for the desired QUEUE I/O directive function codes.

R2 Must contain the number of optional parameters to be included in the QUEUE I/O directive, if any.

R3 Must contain the beginning address of the list of optional QUEUE I/O directive parameters, if R2 contains a non-zero value.

4.11 RENAME FILE ROUTINE (.RENAM)

The .RENAM routine is called to change the name of a file in its associated directory. To rename a file, the user must specify the address of an FDB containing filename information, a LUN, and an event flag number to be used in connection with renaming the file. If the file to be renamed is open when the call to .RENAM is issued, that file is closed under its new name, provided that the renaming operation is successful.

The following registers must be preset before calling this routine:

R0 Must contain the address of the FDB associated with the originally-named file.

R1 Must contain the address of the FDB containing the desired filename information, LUN assignment, and event flag to be associated with renaming the file.

If the renaming operation is successful, a new directory entry is created, and the original entry is deleted. If the operation is not successful, the file is closed under its original name, and the associated directory is not affected.

NOTE

The renaming process is merely a directory operation which replaces an old entry with a new entry. The filename stored in the file's header block is not altered.

4.12 FILE EXTENSION ROUTINE (.EXTND)

The .EXTND routine is called to extend either contiguous or noncontiguous files. The file to be extended can be either open or closed.

The following registers must be preset before calling this routine:

R0 Must contain the address of the associated FDB.

R1 Must contain a numeric value specifying the number of blocks to be added to the file.

R2 Must contain the extension control bits, as appropriate. The possible bit configurations for controlling file extend operations are detailed in Table 4-1. This table defines the bits in the low-order byte of R2. The high-order 8 bits of R2 (bits 8 through 15) are used in conjunction with the 16 bits of R1 to define the number of blocks to be added to the file (see Note below).

NOTE

The contents of R1 and the high-order byte of R2 (bits 8 through 15) are used by FCS in accomplishing the specified .EXTND operation. Thus, 24 bits of magnitude are available for specifying the number of blocks by which the file is to be extended.

Table 4-1
R2 Control Bits for .EXTND Routine

BIT SETTINGS - Low-Order Byte of R2								BIT DEFINITIONS AND MEANING
7	6	5	4	3	2	1	0	
								EX.ENA - Bit 7 = 0
0	x	x	x	x	x	x	0	EX.AC1 - BIT 0 = 0; indicates that extend is to be noncontiguous.
0	x	x	x	x	x	x	1	EX.AC1 - BIT 0 = 1; indicates that extend is to be contiguous and that file is to be contiguous.
								EX.ENA - Bit 7 = 1
1	x	x	x	x	x	x	0	EX.AC1 - Bit 0 = 0; indicates that noncontiguous area is to be added to the file.
1	x	x	x	x	x	x	1	EX.AC1 - Bit 0 = 1; indicates that contiguous area is to be added to the file.
1	x	x	x	x	x	1	x	EX.AC2 - Bit 1 = 1; indicates that the largest available contiguous area is to be added to the file if desired extend space is not available. This bit is set only if bit 0 in EX.AC1 is set to one (1).
1	x	x	x	x	0	x	x	EX.FCO - Bit 2 = 0; indicates that the file is to be noncontiguous.
1	x	x	x	x	1	x	x	EX.FCO - Bit 2 = 1; indicates that the file is to be contiguous.
1	x	x	x	0	x	x	x	EX.ADF - Bit 3 = 0; indicates that the user intends to allocate the number of blocks specified by R1 and the high-order bits of R2 (see Note above).
1	x	x	x	1	x	x	x	EX.ADF - Bit 3 = 1; indicates that file extension is to occur according to the volume default extend value, as established by the INITIALIZE, INITVOLUME, or MOUNT command.

4.13 FILE DELETION ROUTINES

The following routines are provided for deleting files.

4.13.1 .MRKDL - Mark Temporary File for Deletion

The .MRKDL routine is used in conjunction with a temporary file, i.e., a file created through the OPNT$W macro call (see section 3.3). Such a file has no associated directory entry.

A call to the .MRKDL routine is issued prior to closing a temporary file. The file so marked is then deleted automatically when the file is closed.

Before calling the .MRKDL routine, the following register must be preset:

R0 Must contain the address of the associated FDB. This FDB is assumed to contain the file identification, device name, and unit information pertaining to the file to be deleted.

If the .MRKDL routine is invoked while the temporary file is open, as is normally done, then the file is deleted unconditionally when it is closed, even if the calling task terminates abnormally without closing the file.

4.13.2 .DLFNB - Delete File by Filename Block

This routine is used to delete a file by filename block. The .DLFNB routine assumes that the filename block is completely filled in, and, when called, it closes the file, if necessary, and then deletes the file.

Before calling this routine, the following register must be preset:

R0 Must contain the address of the associated FDB.

The .DLFNB routine operates in the same manner as the routine invoked by the DELET$ macro call (see section 3.17), but .DLFNB does not require any of the .PARSE logic and is thus considerably smaller (in terms of memory requirements) than the normal DELET$ function.

Like the DELET$ operation, however, if the file to be deleted is not currently open, and if an explicit file version number is not present in offset location N.FVER of the associated filename block, then the .DLFNB operation will fail.

4.14 DEVICE CONTROL ROUTINE (.CTRL)*

The .CTRL routine is called to perform device-specific control functions. The following are examples of .CTRL device-specific functions:

1. Rewind a magnetic tape volume set,
2. Position to the logical end of a magnetic tape volume set,
3. Close the current magnetic tape volume and continue file operations on the next volume.

The following registers must be preset before calling this routine.

R0 Must contain the address of the associated FDB.

R1 Must contain one of the following function codes.

FF.RWD to rewind a magnetic tape volume set

FF.POE to position to the logical end of a magnetic tape volume set

FF.NV to close the current volume and continue file operations on the next volume of a magnetic tape volume set

R2 and R3 must contain zeros.

See Chapter 5 for an explanation of the use of .CTRL to accomplish magnetic tape device-specific functions.

*This routine does not apply to RSX-11M.

CHAPTER 5

FILE STRUCTURES

IAS, RSX-11D, and RSX-11M support an identical file structure on disk and DECtape. IAS and RSX-11D support also, a file structure on magnetic tape.

RSX-11M supports a magnetic tape file structure only in conjunction with the File Exchange Utility (FLX). This program is described in detail in the RSX-11M Utilities Procedures Manual.

The disk and DECtape file structure is called FILES-11; the IAS/RSX-11D magnetic tape file structure is ANSI standard.

5.1 DISK AND DECTAPE FILE STRUCTURE (FILES-11)

Volumes contain both user files and system files. Disks and DECtapes initialized through the INITIALIZE (IAS) or INITVOLUME (RSX) command have the standard FILES-11 structure built for them automatically. The standard system files created through these commands include the following:

1. Index file;
2. Storage allocation file;
3. Bad block file;
4. Master file directory (MFD); and
5. Checkpoint file (not used by RSX-11M).

Each FILES-11 volume has a file of each type. A volume may have more than one directory file; such files, created by the CREATE/DIRECTORY command in IAS, and the UFD command in RSX-11 systems, are used by the system to locate user files on the volume.

The INITVOLUME command is described in detail in the RSX-11D User's Guide or the RSX-11M Operator's Procedures Manual; the INITIALIZE command description can be found in the IAS User's Guide.

5.1.1 User File Structure

User data files on disk and DECtape consist of ordered sets of virtual blocks that constitute the virtual structure of the file as it appears to the user. Virtual blocks can be read and written directly by issuing READ$ and WRITE$ macro calls (see sections 3.15 and 3.16, respectively). Virtual blocks are numbered in ascending sequence relative to the first block in the file (which is virtual block 1).

The virtual blocks of a file are stored on the volume as logical blocks. The logical block size of all volumes is 256 words; thus, each virtual block is also 256 words. When access to a virtual block is requested, the virtual block number is mapped into a logical block number. The logical block number is then mapped to the physical address on the associated volume.

5.1.2 Directory Files

A directory file contains directory entries. Each entry consists of a filename and its associated file number and file sequence number. The number of directory files required depends on the number of users of the volume. For single-user volumes, only a master file directory (MFD) is needed; for multiple-user volumes, a master file directory (MFD) is required, and one user file directory (UFD) is required for each user of the volume.

The master file directory contains a list of all the user file directories on the volume, and each user file directory contains a list of all that user's files. User file directories (UFD's) are identified by user identification codes (UIC's). A user file directory is created by the UFD command in RSX-11 systems, and the CREATE/DIRECTORY command in IAS. These commands are described in detail in the RSX-11D User's Guide, the RSX-11M Operator's Procedures Manual, or the IAS System Management Guide.

Figures 5-1 and 5-2 illustrate the directory structure for single-user and multiple-user volumes, respectively.

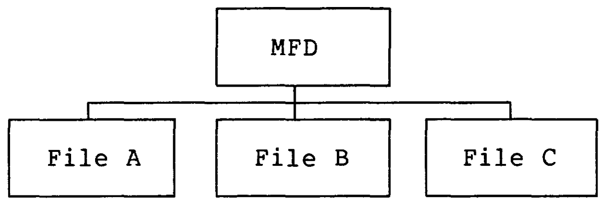

Figure 5-1
Directory Structure for Single-User Volumes

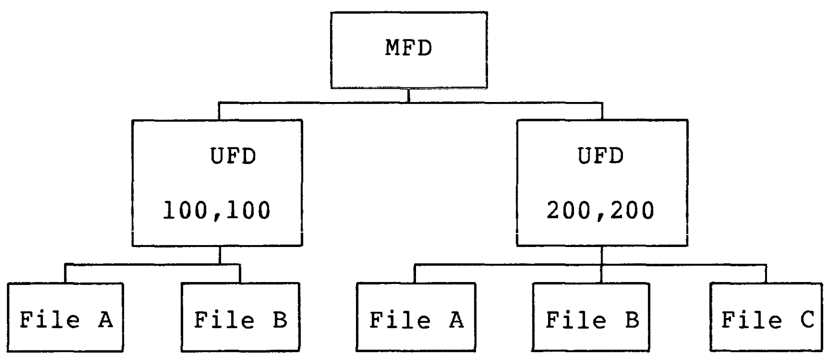

Figure 5-2
Directory Structure for Multiple-User Volumes

5.1.3 Index File

The index file contains volume information and user file header blocks, both of which are used by the file control primitives (FCP). Because the file header blocks (see below) are stored in the index file, they can be located very quickly. Furthermore, since a file header block is 256 words in length, it can be read into memory with a single access.

The index file is created when a volume is initialized for use by the host operating system. During initialization, the information required by the system is placed in the index file. Appendix E contains a detailed description of the format and content of an index file.

5.1.4 File Header Block

Associated with each file is a file header block that contains information describing the file. File header blocks are stored in the index file. Each file header block is 256 words in length and contains three areas: the header area, the identification area, and the map area.

The header area identifies the block as a file header block. Each file is uniquely identified by a file ID consisting of two words. The first word of the file ID, i.e., the file number, is used to calculate the virtual block number (VBN) of that file's header block in the index file. (This calculation is done, as follows: VBN = the file number + 3 + the number of index file bit map blocks.) The second word, i.e., the file sequence number, is used to verify that the header block is in fact the header for the desired file.

When a request to access a file is issued, both the file number and the file sequence number are specified. The access request will be denied if the file sequence number does not match the corresponding field in the file header block associated with the specified file number.

When a file is deleted, its file header block is made available for the subsequent creation of a new file, and when the new file is created, a different file sequence number is stored in the file header block. If a user attmpts to access a file that has been deleted (e.g., by referencing an obsolete directory entry), this updated file sequence number ensures the failure of the access request, even if the same file header block is re-used for a different file.

The identification area specifies the creation name of the file and identifies the file owner's UIC. This area also specifies the creation date and time, the revision number, the date and time of the last revision (i.e., the time and date on which the last modification to the file occurred), and the expiration date.

The map area provides the information needed by the system to map virtual block numbers to logical block numbers.

A checksum value is computed each time the file header block is read from or written to the volume, thus ensuring that the file header block was transferred correctly. Appendix F contains a detailed description of the format and content of the file header block.

5.2 MAGNETIC TAPE FILE PROCESSING (IAS AND RSX-11D ONLY)

IAS and RSX-11D 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 X.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 Appendix G.

5.2.1 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 G.1.1.1. Volumes produced by nonDIGITAL systems are restricted to read-only access unless explicitly overridden at MOUNT time.

5.2.2 Rewinding Volume Sets

A magnetic tape volume set can be rewound either by using the FDOP$R macro call before an OPEN$ or CLOSE$ or by using the .CTRL file control subroutine. Regardless of the method used to rewind the volume set, the following procedures are performed by the file control 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 an OPEN$ macro call or by a .CTRL call, a request to mount the first volume is printed on the operator's console,
4. If the rewind was requested on a CLOSE$ macro call, no mount message is issued until the next volume is needed.

5.2.3 Positioning to the Next File Position

The FDOP$R macro call can be used to indicate that the file just opened 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 option 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, FA.POS must be set in FDB location F.ACTL. The default value for this FDB position is 0 (not FA.POS). The default indicates that 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 FA.POS.

The next file position option is ignored by directory device file processors. However, programs written mainly for directory devices cannot specify the next file position option in open commands for output and, therefore, cause the position to end process to be used automatically.

5.2.4 Single File Operations

Single file operations are performed by specifying the rewind option before the open and before the close. Using this approach, scratch tape operations can be performed as follows:

1. Open 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 (rewind is optional),
4. Read and process the data,
5. Close the file with rewind,
6. Open the second file with rewind specified,
7. Write the data records,
8. Close the file with rewind and perform any additional processing.

5.2.5 Multiple File Operations

A multiple file volume is created by opening, writing, and then closing a series of files without specifying a rewind. The sequential processing of files on the volume can be accomplished by closing without rewind and then opening the next file without rewind.

Opening a file for extend (OPEN$) 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. Open a file for output with rewind,

2. Write data records and close the file,

3. Open 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.

5.2.6 Using .CTRL

The .CTRL file control routine can be called to override normal FCS defaults for magnetic tape. Examples of its uses are:

1. Continue processing a file on the next volume of a volume set before the end of the current volume is reached,

2. Position to the logical end of the volume set,

3. Rewind a volume at other than file open or close.

When .CTRL is used to continue processing a file on the next volume, the first file section on the next volume is opened. 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 .CTRL processing occurs.

1. If the current volume is the last volume in the set, i.e., there is no next volume, end of file is reported to the user.

2. 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.

3. The header label (HDR1) of the first file section is read and checked.

4. If all required fields check, the operation continues.

5. If any check fails, the operator is requested to mount the correct volume.

For output files, the following processing occurs.

1. The current file section is closed with EOV1 and EOV2 labels and the volume is rewound.

2. The next volume is mounted.

3. 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.

NOTE

I/O buffers that are currently in memory are written on the next file section.

When .CTRL is used to position to the logical end of the volume set, the file system positions between the two tape marks at the logical end of last volume in the set.

5.2.7 Examples of Magnetic Tape Processing

The following pages contain examples of FCS statements used to process magnetic tape. Macro parameters not related to magnetic tape handling have been omitted from the statements so that the user need consider only those matters directly related to magnetic tape.

5.2.7.1 Examples of OPEN$W to Create a New File - All routines expect R0 to contain the FDB address.

```
OPRWDO:
;
; OPEN WITH REWIND
;
        FDOP$R  R0,,,,#FA.ENB!FA.RWD     ;SET REWIND AND ENABLE USE
        BR      OPNOUT                   ;OF F.ACTL
OPNXTO:
;
; OPEN FOR NEXT FILE POSITION
;
        FDOP$R  R0,,,,#FA.ENB!FA.POS     ;SET POSITION TO NEXT
        BR      OPNOUT                   ;AND ENABLE USE OF F.ACTL
OPROYK:
;
; OPEN FILE AT END OF VOLUME KEEPING CURRENT USER
; ACCESS CONTROL BITS
;
        BIC     #FA.ENB,F.ACTL(R0)       ;DISABLE USE OF F.ACTL
        BR      OPNOUT
OPROVO:
;
; OPEN FILE AT END OF VOLUME - SELECT SYSTEM DEFAULT FOR
; USER ACCESS CONTROL BITS
        FDOP$R  R0,,,,#0                 ;DISABLE USE OF AND RESET
        BR      OPNOUT                   ;F.ACTL TO ZERO
;
; OPEN FILE WITH CURRENT USER ACCESS CONTROL
;
OPOURO:
        BIS     #FA.ENB,F.ACTL(R0)       ;ENABLE USE OF F.ACTL
OPNOUT: OPEN$W  R0                       ;OPEN FILE
        RETURN
```

5.2.7.2 Examples of OPEN$ to Read a File - All routines expect R0 to contain the FDB address.

```
OPRWDI:
;
; OPEN WITH REWIND
;
        FDOP$R  R0,,,,#FA.ENB!FA.RWD
        BR      OPNIN
OPCURI:
;
; OPEN STARTING SEARCH AT CURRENT TAPE POSITION KEEPING USER
; ACCESS CONTROL BITS
;
```

```
        BIC     #FA.ENB,F.ACTL(R0)       ;DISABLE USE OF F.ACTL
        BR      OPNIN
OPNIN:
;
; OPEN USING USER ACCESS CONTROL
;
OPDFLI: BIS     #FA.ENB,F.ACTL(R0)       ;ENABLE USE OF F.ACTL
        OPEN$R  R0
        RETURN
```

5.2.7.3 Examples of CLOSE$ - All routines expect R0 to contain the FDB address.

```
CLSCUR:
;
; CLOSE LEAVING TAPE AT CURRENT POSITION AND KEEPING
; USER ACCESS CONTROL BITS
;
        BIC     #FA.ENB,F.ACTL(R0)       ;DISABLE USE OF F.ACTL
        BR      CLOSE                    ;DEFAULT IS LEAVING AT CURRENT
                                         ;POSITION
CLSRWD:
;
; CLOSE REWINDING THE VOLUME
;
        FDOP$R  R0,,,,#FA.ENB!FA.RWD     ;SET REWIND AND ENABLE USE OF
        BR      CLOSE                    ;F.ACTL
;
; CLOSE WITH USER ACCESS CONTROL BITS
;
CLSDFL: BIS     #FA.ENB,F.ACTL(R0)       ;ENABLE USE OF F.ACTL
CLOSE:  CLOSE$  R0
        RETURN
```

5.2.7.4 Combined Examples of OPEN$ and CLOSE$ for Magnetic Tape - The following examples call routines in previous examples. By combining various magnetic tape operations the user can process tape volumes in the following ways.

```
;
; SCRATCH TAPE OPERATIONS--SINGLE FILE VOLUME--
;
SCROUT: MOV     #FDBOUT,R0              ;SELECT FDB AND OPEN
        CALL    OPRWDO                  ;OUTPUT FILE WITH REWIND
        RETURN
SCRIN:  MOV     #FDBIN,R0               ;SELECT FDB AND OPEN FOR
        CALL    OPRWDI                  ;INPUT WITH REWIND
        RETURN
CLSCRO: MOV     #FDBOUT,R0              ;CLOSE SCRATCH FILE
        BR      CLSVOL                  ;REWINDING VOLUME
CLSCRI: MOV     FDBIN,R0
CLSVOL: CALL    CLSRWD
        RETURN
;
; MULTI-FILE VOLUME OPERATIONS
;
OPNXTI:
;
; OPEN FILE FOR READING WHEN FILE IS NEXT OR FURTHER UP THE VOLUME
;
        MOV     #FDBIN,R0               ;SELECT FDB
        CALL    OPCURI                  ;OPEN FILE
        RETURN
OPENIN:
;
; OPEN FILE FOR READING WHEN POSITIONED PAST IT
;
        MOV     #FDBIN,R0               ;SELECT FDB
        CALL    OPRWDI
        RETURN
;
; MULTI-FILE OUTPUT OPERATIONS
;
OPNINT:
;
; START NEW VOLUME DESTROYING ALL PAST FILES ON IT
;
        MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        CALL    OPRWDO                  ;OPEN WITH REWIND
        RETURN

OPNEXT:
;
; OPEN OUTPUT FILE AT NEXT FILE POSITION DESTROYING ANY FILE
; THAT MAY BE AT OR PAST THAT POSITION
;
        MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        CALL    OPNXTO
        RETURN

OPENDT:
;
; OPEN OUTPUT FILE AT CURRENT END OF VOLUME SET KEEPING USER
; ACCESS CONTROL BITS
;
```

```
        MOV     #FDBOUT,R0                  ;SELECT OUTPUT FDB
        CALL    OPROVK
        RETURN

OPNEOV:
;
; OPEN OUTPUT FILE AT CURRENT END OF VOLUME AND MAKE THAT THE USER
; ACCESS CONTROL
;
        MOV     #FDBOUT,R0                  ;SELECT OUTPUT FDB
        CALL    OPROVO
        RETURN
;
; NOT LAST FILE IN FILE SET CLOSE ROUTINE
;
CLSFLO: MOV     #FDBOUT,R0                  ;SELECT OUTPUT FDB
        BR      CLSXX
CLSFLI: MOV     #FDBIN,R0                   ;SELECT INPUT FDB
CLSXX:  CALL    CLSCUR
        RETURN
;
; TO APPEND TO LAST FILE
;
        OPEN$A  #FDBOUT
```

CHAPTER 6

COMMAND LINE PROCESSING

As noted in section 2.4.3, a collection of routines available from the system object library (SY:[1,1]SYSLIB.OLB) may be linked with the user program to provide all the logical capabilites required to process command lines dynamically. These system facilities include the following routines:

1. Get Command Line (GCML). This routine accomplishes all the logical functions associated with the entry of command lines from a terminal, an indirect command file, or an on-line storage medium. Using GCML relieves the user of the burden of manually coding command line input operations.

2. Command String Interpreter (CSI). Normally, this routine takes command lines from the GCML command line input buffer and parses them into the appropriate dataset descriptors required by FCS for opening files.

This body of routines is linked with the user program at task-build time. GCML and CSI are often jointly incorporated in system or application programs as a standardized interface for obtaining and interpreting dynamic command line input. The flow of data during command line processing is shown in Figure 6-1.

Although these routines are presented in the context of being used together for processing command line input, each may be used independently of the other. Doing so, however, means that the user must manually code the functions otherwise performed by the missing component. The joint use of these routines is assumed throughout this chapter to be the "normal" situation.

The invocation of GCML and CSI functions requires that certain initialization operations be accomplished at assembly time. This initialization sets up the GCML command line input buffer, defines and initializes control blocks for both GCML and CSI, and establishes the necessary working storage and communication areas for these routines. Also, the appropriate macro calls which invoke GCML and CSI execution-time functions must be included in the source coding at desired logical points to effect the dynamic processing of command lines.

GCML and CSI macro calls observe the same register conventions applicable to FCS. All registers, except R0, are preserved exactly as in all FCS macro calls. R0 is used to contain the address of the GCML control block or the CSI control block, as appropriate.

As with all FCS macro calls, the GCML and CSI macro calls must also be listed as an argument in an .MCALL directive (see section 2.1) before being issued in the user program.

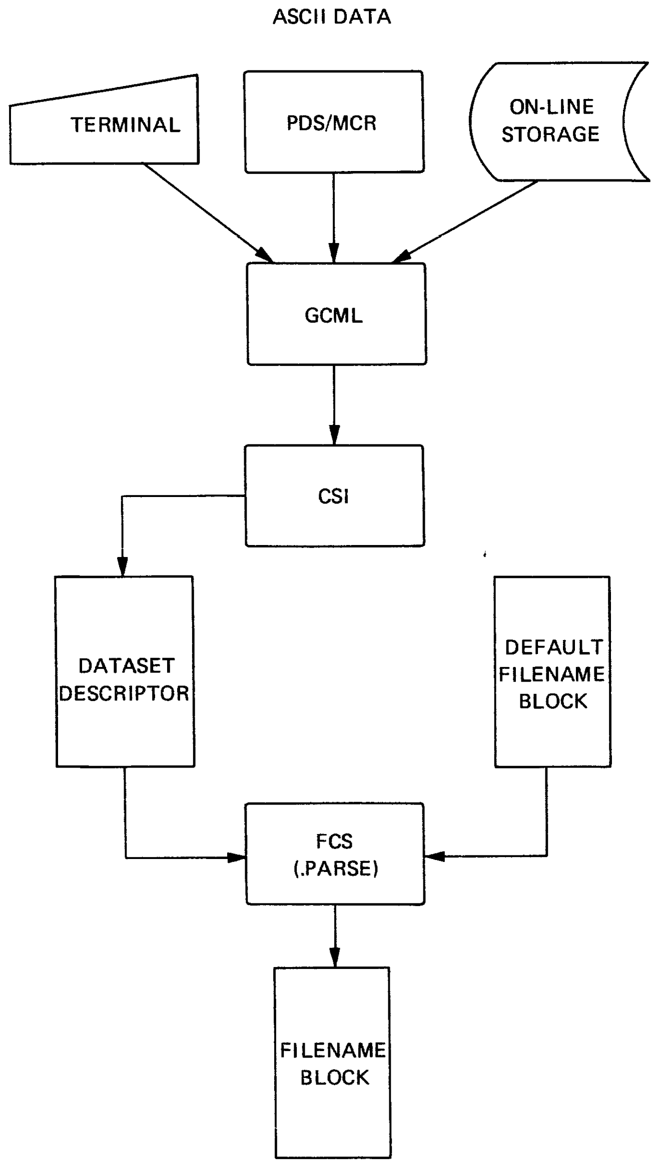

Figure 6-1
Data Flow During Command Line Processing

6.1 GET COMMAND LINE (GCML)

The Get Command Line routine (GCML) embodies all the logical capabilities required to enter 80-byte command lines dynamically during program execution. GCML accepts input from a terminal or an indirect command file which contains pre-defined command lines. Both these functions require the creation and initialization of a GCML control block. The macro call which accomplishes this function is described in detail in the following section. The GCML run-time macro calls that may be issued dynamically are described in section 6.1.3.

6.1.1 GCMLB$ - Allocate and Initialize GCML Control Block

Issuing the GCMLB$ macro call accomplishes the following assembly-time functions:

1. Reserves storage for and initializes a GCML control block within the user program.

2. Creates and initializes an FDB in the forepart of the GCML control block. This FDB is used to open a command file. Such a file, which may employ a terminal or a file-structured device such as a disk, is opened and read by the user program in the same manner as any other file. The initialization and maintenance of this FDB, however, is under GCML and FCS control and need not be of concern to the user.

3. Creates and initializes a default filename block within the GCML control block. This default filename block pertains to an indirect command file. If an explicit filename string is not specified by the user for an indirect command file, the values "SY:" for the device name and ".CMD" for the file type are assumed by default. There is no default designation for the filename.

4. Defines the symbolic offsets for the GCML control block and initializes certain offsets to required values. These offsets are described in detail in section 6.1.2.

The GCMLB$ macro call is specified in the following format:

```
label:  GCMLB$  maxd,prmpt,ubuf,lun,pdl
```

where:

label
: represents a symbol that names the GCML control block and defines its address. This label permits the GCML control block to be referenced directly by all the GCML run-time routines which require access to this structure (see section 6.1.3).

maxd
: represents a numeric value that specifies the maximum nesting depth permitted for indirect command files. This parameter determines the number of nested indirect command files that GCML will be allowed to access in obtaining command line input.

  An indirect command file, which often resides on disk, contains well-defined, non-varying command sequences which may be read directly by GCML to control operations which are highly repetitive (such as Task Builder activities). Significant advantages in terms of convenience and faster execution result from the use of an indirect command file.

If this parameter is not specified, a nesting level depth of zero (0) is defined by default, effectively eliminating an indirect command file as a source of command line input.

prmpt represents a user-specified, 3-character ASCII prompting sequence. This parameter constitutes a default prompt string that is typed out by GCML to the user terminal to solicit command line input.

The ASCII prompting sequence is formulated into the following 6-byte string:

A. A carriage return (<CR>) and a line-feed (<LF>);

B. The three ASCII characters specified by the user; and

C. A right angle bracket (>).

The above string initializes GCML control block offset location G.DPRM (see section 6.1.2).

If this parameter is not specified, the right angle bracket (>), preceded by three blanks, is defined by default for use by GCML as the default prompting sequence.

ubuf represents the address of a 41-word record buffer that is to be used by GCML for the temporary storage of command line input. If this parameter is not specified, a 41-word buffer is reserved by default in the GCML control block for command line input.

lun represents a logical unit number. The device assigned to this logical unit number is used by GCML as the command input device. If this parameter is not specified, a logical unit number of one (1) is used by default.

pdl represents the address of an area reserved in the user program for use as a push-down list. This area is reserved as working storage for use in connection with indirect command files.

Normally, the pdl parameter is not specified; in this case, sufficient storage for the push-down list is added to the control block by default in accordance with the algorithm described below.

The push-down list is created through statements logically equivalent to the following:

```
        .EVEN
label:  .BLKB   G.LPDL
```

The user-specified "label" names the push-down list and defines its address; G.LPDL, which is defined by the GCMLB$ macro, is the length (in bytes) of the push-down list.

The length of the push-down list is a function of the maximum number of nested indirect command files that may be accessed by GCML in obtaining command line input. The value of G.LPDL is calculated according to the following algorithm:

1. Add one (1) to the maximum nesting level depth declared through the maxd parameter (see above).

2. Multiply the sum of Step 1 by 16(10). The appropriate number of bytes that must be reserved for the push-down list.

For example, if the maxd parameter is specified as "4", the length of the push-down list is derhted as follows:

(4+1)*16. = 80. bytes

From the above, note that 16(10) bytes of storage are required for each indirect command file, plus a total of 16(10) bytes for use as general overhead.

The following examples are representative of a GCMLB$ macro call as it might appear in a user program:

```
GCLBLK: GCMLB$  4.,GCM,BUFADR,1.
GCLBLK: GCMLB$  ,,BUFADR
GCLBLK: GCMLB$  DEPTH,GCM,BUFADR,CMILUN,PDLIST
```

6.1.2 GCMLD$ - Define GCML Control Block Offsets and Bit Values

The GCMLD$ macro, which is invoked automatically by the GCMLB$ macro call, locally defines the GCML control block offsets and bit values within the current module. These offsets and associated bit values are listed and described below.

OFFSET NAME	FUNCTIONAL SIGNIFICANCE
G.ERR	Error Return Code Byte. This field initially contains zero (0). If any one of the error conditions recognized by GCML occurs during the processing of a command line, an appropriate error code is returned to offset location G.ERR in the control block. These error codes are described below: GE.IOR - Indicates that an I/O error has occurred during the input of a command line. GE.OPR - Indicates that GCML is unable to open the specified command file. GE.BIF - Indicates that a syntax error has been detected in the name of the indirect command file. GE.MDE - Indicates that an attempt has been made to exceed the maximum permissible nesting level depth for an indirect command file (see the "maxd" parameter in the GCMLB$ macro call above). GE.EOF - Indicates that the end-of-file (EOF) on the first (unnested) command file has been detected. This bit is set in connection with command file input. When the first call is issued for input, GCML attempts to retrieve an MCR/PDS command line. The first line obtained, whether it be an MCR/PDS command or a terminal command, is accomplished at command level 0. If the name of an indirect command file is then entered, the command input level is incremented to one (1). Each indirect filename entry thereafter increments the command input level. When the end-of-file (EOF) is encountered on any given indirect file, the command input level is decremented by one (1), restoring the count to the previous level and re-opening the associated command file. The next command line from that file is then read. If an MCR/PDS command has already been read at level 0, entering another MCR/PDS command when level 0 is again reached causes the error code GE.EOF to be returned to offset location F.ERR of the GCML control block. Hence, only one MCR/PDS command line can be read at level 0. If input thus fails at MCR/PDS level 0, then GCML continues to prompt for input until CTRL/Z is typed by the user to indicate terminal end-of-file (EOF).

In summary, the first line of input is always read at level 0. This initial input may be an MCR/PDS command; if the MCR/PDS command fails or is null, the command input file (normally a terminal) is then opened at level 0. Multiple inputs at level 0 are permissible only in the latter case, i.e., from the command input file.

G.MODE

Status and Mode Control Byte. This field is initialized at assembly-time with the following bit definitions to specify certain default actions for GCML during the retrieval of a command line:

GE.COM - Indicates that a command line having a leading semicolon (;) is to be treated as a comment. Such lines are not returned to the calling program. If, for any reason, the user resets this bit to zero (0), a command line containing a leading semicolon (;) will be returned to the calling program.

GE.IND - Indicates that a command line containing a leading at sign (@) is to be treated as an explicit indirect command file specifier. If, for any reason, the user resets this bit to zero (0), a command line containing a leading at sign (@) will be returned to the calling program.

GE.CLO - Indicates that the command file currently being read is to be closed after each issuance of the GCML$ macro call. If the user resets this bit to zero (0) for any reason, GCML keeps the current command file open between calls for input. In this case, the FSR (see section 2.6.1) must include one additional 512(10)-byte buffer for command line input. This requirement is additive to the total FSR block buffer space normally reserved for the maximum number of files that may be open simultaneously for record I/O processing.

Clearing the GE.CLO bit in the status and mode control byte effectively renders 512(10) bytes of FSR block buffer space unavailable for other purposes, since the command file remains open between calls for command line input.

As noted above, the user may reset any of the status and mode control bits, if desired, by issuing a Bit Clear Byte (BICB) instruction which takes as the source operand the symbolic name of the bit to be cleared. Bits other than those defined above are used internally by GCML and must not be manipulated by the user.

G.PSDS

Prompt String Descriptor. This 2-word field is initialized to zero (0) at assembly-time through the GCMLB$ macro call (see section 6.1.1).

When the GCML$ macro call is issued to request command line input (see section 6.1.3.1), the address and the length of a prompting sequence is usually not specified. In this case, the prompt string descriptor words in the GCML control block are cleared, causing GCML to type out the default prompt string contained in offset location G.DPRM (see below) to solicit command line input.

If, for any reason, the user wishes to define an alternate prompt string elsewhere in his program, he may do so through the .ASCII directive. The address and length of this alternate prompt string may then be specified as the "adpr" and "lnpr" parameters in subsequent GCML$ macro calls. These parameters cause offset locations G.PSDS+2 and G.PSDS to be initialized with the address and the length, respectively, of the alternate prompt string. The alternate prompt string is then typed out by GCML to solicit command line input, thereby overriding the default prompt string previously established through the GCMLB$ macro call (see G.DPRM below).

If the "adpr" and "lnpr" parameters are not specified in a subsequent GCML$ macro call, offset location G.PSDS in the control block is automatically reset to zero (0), causing GCML to revert to the use of the default prompt string contained in offset location G.DPRM.

G.CMLD

Command Line Descriptor. This 2-word field is initialized by GCML after retrieving a command line. The address of the line just obtained is returned to offset location G.CMLD+2, and the length (in bytes) of the command line is returned to offset location G.CMLD.

The contents of these word locations in the GCML control block may be passed to CSI as the "buff" and "len" parameters in the CSI$1 macro call (see section 6.2.3.1). The combination of these parameters constitutes the command line descriptors which enable CSI to retrieve file specifiers from the GCML command line input buffer.

G.ISIZ — Impure Area Size Indicator. This symbol is defined at assembly-time, indicating the size of an impure area within the GCML control block to be used as working storage for pointers, flags, counters, etc., in connection with input from an indirect command file. In usage terms, this symbol need not be of concern to the user.

The space between the FDB and the default prompt string (see G.DPRM below) constitutes the impure area of the GCML control block. The size of the FDB is defined by the value of the symbol S.FDB. Thus, the size of the impure area is equal to G.DPRM-S.FDB.

G.DPRM — Default Prompt String. This 6-byte field is initialized at assembly-time with the default prompt string created through the "prmpt" parameter of the GCMLB$ macro call (see section 6.1.1). In the absence of the "adpr" and "lnpr" parameters in the GCML$ macro call (see section 6.1.3.1), this default prompt string is typed out by GCML to solicit terminal input.

If the user wants to reference the GCML control block offsets and bit vaues in another module, the appropriate symbolic definitions may be established within that module through one of the following statements, as desired:

```
GCMLD$                  ;DEFAULT LOCAL DEFINITION.

GCMLD$  DEF$L           ;LOCAL DEFINITION.

GCMLD$  DEF$G           ;GLOBAL DEFINITION.
```

6.1.3 GCML Run-Time Macro Calls

Three run-time macro calls are provided in GCML to perform specific functions, as described below:

GCML$ - To retrieve a command line.

RCML$ - To reset the indirect command file scan to the first (unnested) level.

CCML$ - To close the current command file.

These routines are described separately in the following sections.

6.1.3.1 GCML$ - Get Command Line

The GCML$ macro call serves as the user program interface for retrieving command lines from a terminal or an indirect command file. This macro call can be issued at any logical point in the program to solicit command line input.

This macro call takes the following format:

```
GCML$   gclblk,adpr,inpr
```

where:

gclblk — represents the address of the GCML control block. This symbol must be the same as that specified at assembly-time in the label field of the GCMLB$ macro call (see section 6.1.1). If this parameter is not specified, R0 is assumed to contain the address of the GCML control block.

adpr — represents the address of the user program location containing an alternate prompt string. When this optional parameter and the inpr parameter below are present in the GCML$ macro call, the alternate prompt string is typed out on the user terminal to solicit command line input. The normal default prompt string, as contained in offset location G.DPRM of the GCML control block (see section 6.1.2), is thereby overridden.

lnpr — represents the length (in bytes) of the alternate prompt string. This parameter is also optional; if not specified, offset location G.PSDS in the GCML control block (see section 6.1.2) is cleared.

If this parameter is specified, but the "adpr" parameter above is not, an .ERROR directive is generated during assembly which causes the error message "PROMPT STRING MISSING" to be printed in the assembly listing. This message is a diagnostic announcement of an incomplete prompt string descriptor in the GCML$ macro call.

If the "adpr" and "lnpr" parameters are not specified in a subsequent GCML$ macro call, offset location G.PSDS in the GCML control block is automatically reset to zero (0), causing GCML to revert to the use of the default prompt string contained in offset location G.DPRM (see section 6.1.2 above).

When the GCML$ macro call is issued, the following actions occur:

1. R0 is loaded with the address of the GCML control block. If the "gclblk" parameter is not specified, as described above, R0 is assumed to contain the address of the GCML control block. If it does not, R0 must first be initialized manually with the address of the control block before the GCML$ macro call is issued.

2. The address and the length of the alternate prompt string, if specified, are stored in control block offset locations G.PSDS+2 and G.PSDS, respectively. These two words constitute the alternate prompt string descriptor.

3. Code is generated which calls GCML to transfer an 80-byte command line to the command line input buffer.

At the initial issuance of the GCML$ macro call, an attempt is made to retrieve an MCR/PDS command line. If this attempt fails, or if the MCR/PDS command line is null, the FDB within the GCML control block is used to open a file for command line input. If the command input device is a terminal, a prompt string is typed out to solicit input. Any desired command input may then be entered.

If appropriate, the user may enter an at sign (@) as the first character in the command line, followed by the name of an indirect command file. This filename identifies an explicit indirect command file from which input is to be read. GCML then opens this file and retrieves the first command line therein. This file is read until one of the following occurs:

1. The end-of-file (EOF) is detected on the current indirect file. In this case, the current indirect file is closed, the command input level count is decremented by one (1), and the previous command file is re-opened. If the command input level count is already zero (0) when EOF is detected, the error code GE.EOF is returned to offset location G.ERR of the GCML control block (see section 6.1.2).

2. An indirect file specifier is encountered in a command line. In this case, the current indirect command file is closed (if not already closed), and the new indirect file is opened. The first command line therein is then read.

3. An RCML$ macro call is issued in the program (see section 6.1.3.2 below). In this case, the current indirect command file is closed, and the command input count reverts to level zero (0), i.e., the top level command file is again used for input.

The user may also enter a semicolon (;) as the first character in the command line. Such a line is treated as a comment and is not returned to the calling program.

Whether a command line is entered manually or retrieved from an indirect command file, the address and the length of the command line thus obtained are returned to GCML control block offset locations G.CMLD+2 and G.CMLD, respectively. Together, these two words constitute the command line descriptors. These descriptors may be specified as the "buff" and "len" parameters in the CSI$1 macro call (see section 6.2.3.1).

Successful retrieval of a command line causes the C-bit in the Processor Status Word to be cleared. Any error condition that occurs during the retrieval of a command line, however, causes the C-bit to be set. In addition, a negative error code is returned to offset location G.ERR of the GCML control block. These error codes are described in detail in section 6.1.2 above.

Representative examples of the GCML$ macro call follow:

```
GCML$   #GCLBLK

GCML$

GCML$   #GCLBLK,#ADPR,#LNPR
```

The first example specifies the symbolic address of the GCML control block. The second example assumes that R0 contains the address of the GCML control block. Both these forms of the GCML$ macro call will employ the default prompt string contained in offset location G.DPRM of the control block to solicit command line input. The last example specifies the address and the length of an alternate prompt string that the user has defined within the program. This alternate prompt string is used by GCML to prompt for terminal input, rather than using the default prompt string contained in the GCML control block.

6.1.3.2 RCML$ - Reset Indirect Command File Scan

If, for any reason, the user finds that it is necessary or desirable to close the current indirect command file and to return to the top level file, i.e., to the first (unnested) file, he may do so by issuing the RCML$ macro call.

The RCML$ macro call is specified in the following format:

```
        RCML$   gclblk
```

where: gclblk represents the address of the GCML control block. If this parameter is not specified, R0 is assumed to contain the address of the GCML control block.

When this macro call is issued, the current indirect command file is closed, returning control to the top level (unnested) file. A subsequent GCML$ macro call then retrieves the next command line from the zero (0) level command file. Note, however, that a second MCR/PDS command at level 0 cannot be read (see GE.EOF error code in offset location G.ERR of GCML control block, section 6.1.2).

Examples of the RCML$ macro call follow:

```
        RCML$   #GCLBLK

        RCML$   R0
```

This macro call requires only the address of the GCML control block.

6.1.3.3 CCML$ - Close Current Command File

It is often desirable to close the current command file between calls for input in order to free FSR block buffer space for some other use. The command file is closed automatically after the retrieval of a command line, provided that the GE.CLO bit in the status and mode control byte remains appropriately initialized (see section 6.1.2). This bit is set to one (1) at assembly-time. If the user resets this bit to zero (0), the current command file remains open between calls for input.

For a program which frequently reads command files, this may be a desirable operational mode, since keeping the file open between calls for input reduces total file access time. However, should it be desirable to close such a file to free FSR block buffer space, the user may do so by issuing the CCML$ macro call.

The CCML$ macro call takes the following format:

```
        CCML$   gclblk
```

where: gclblk represents the address of the GCML control block. If this parameter is not specified, R0 is assumed to contain the address of the GCML control block.

Issuing this statement closes the current command file, effectively releasing 512(10) bytes of FSR block buffer space for some other use between calls for input. If the command file is already closed when the CCML$ macro call is issued, control is merely returned to the calling program. A subsequent GCML$ macro call then causes the command file to be re-opened and the next command line in the file to be returned to the calling program.

Representative forms of this macro call are shown below:

```
        CCML$   #GCLBLK

        CCML$   R0
```

As in the RCML$ macro call above, this macro call takes a single parameter, viz., the address of the GCML control block.

6.1.4 GCML Usage Considerations

As noted in section 6.1.1, the GCMLB$ macro call creates an FDB in the forepart of the GCML control block. Although this FDB ordinarily need not be manipulated by the user (since it is under GCML and FCS control), the following operations may be performed on this FDB:

1. In an irrecoverable error situation, the user may issue a CLOSE$ macro call (see section 3.8) in connection with this FDB before issuing the system EXIT$ macro call.

2. The user may test the FD.TTY bit in the device characteristics byte (offset location F.RCTL) of the FDB to determine if the command line just obtained was retrieved from a terminal.

3. In the event that error code GE.IOR is returned to control block offset location G.ERR (indicating that an I/O error has occurred during the retrieval of a command line), the user may test offset location F.ERR of the associated FDB for more complete error analysis. This cell in the FDB also contains an error code which may be helpful in determining the nature of the error condition.

   Note, if the automatic file closure feature is in effect for a command file, i.e., if the GE.CLO bit in the status and mode control byte in the GCML control block is set (see G.MODE offset in section 6.1.2), then F.ERR will very likely contain a positive value (normally +1), indicating successful completion of the close operation. A failure in closing the command file is extremely unlikely.

At task-build time, the Task Builder device assignment (ASG) directive should be issued to assign the appropriate physical device unit to the desired logical unit number. For example, to assign the logical unit number (lun parameter) in the GCMLB$ macro call (see section 6.1.1) to a terminal, the following Task Builder directive should be issued:

```
ASG = TI:1
```

The designation TI: is a pseudo device name that is redirected to the command input device. Note that the numeric value following the colon (:) must agree with the numeric value specified as the lun parameter in the GCMLB$ macro call.

The ASG directive is described in further detail in the Task Builder Reference Manual of the host operating system.

As discussed in section 2.6.1 on FSRSZ$, at any given time, there must be an FSR block buffer available for each file currently open for record I/O operations. The block buffer requirements of the command file must be considered when issuing the FSRSZ$ macro.

6.2 COMMAND STRING INTERPRETER (CSI)

The Command String Interpreter (CSI) analyzes command lines and parses them into their component device name, directory, and filename strings. The user should be aware that CSI processes command lines in the following formats only:

1. dev:[g,m]output filename.type;version/switch

   More than one such file specification can be specified by separating them with commas.

2. dev:[g,m]output filename.type;version/switch,...= dev:[g,m] input filename.type;version/switch,...

In addition, CSI maintains a dataset descriptor within the CSI control block (see next section) which may be used by FCS in opening files. The run-time routines which analyze and parse command lines for a calling user program are described in section 6.2.3.

The use of CSI requires that the CSI control block offsets and bit values be defined and that a control block be allocated within the program. The macro described in the following section accomplishes these requisite actions.

6.2.1 CSI$ - Define CSI Control Block Offsets and Bit Values

The only initialization coding required for CSI at assembly-time is that shown below:

```
        CSI$                    ;DEFINES CSI CONTROL BLOCK OFFSETS
                                ;AND BIT VALUES LOCALLY.
        .EVEN                   ;WORD ALIGNS CSI CONTROL BLOCK.
CSIBLK: .BLKB   C.SIZE          ;NAMES CSI CONTROL BLOCK AND
          .                     ;ALLOCATES REQUIRED STORAGE.
          .
          .
```

The CSI$ macro is strictly definitional in nature and does not generate any executable code. The CSI control block resulting from the .BLKB directive serves as a means of communication between CSI and the calling program. The length of the control block is specified by the symbol "C.SIZE," which is defined during the expansion of the CSI$ macro. Also, the expansion of this macro results in the local definition of the symbolic offsets and bit values within the CSI control block.

If desired, the user may cause the control block offsets to be defined globally within the current module. This is done by specifying "DEF$G" as an argument in the CSI$ initialization macro call, as shown below:

```
        CSI$    DEF$G
```

6.2.2 CSI Control Block Offset and Bit Value Definitions

The CSI$ macro call causes the following symbolic offsets and bit values within the CSI control block to be defined locally:

OFFSET NAME	FUNCTIONAL SIGNIFICANCE
C.TYPR	Command String Request Type. This byte field indicates the type of file specifier being requested. Depending on whether an input or output file specifier is being requested (see the "io" parameter in the CSI$2 macro call, section 6.2.3.2), the corresponding bit in this byte is set. The bit definitions for this byte are as follows: CS.INP - Indicates that an input file specifier is being requested. CS.OUT - Indicates that an output file specifier is being requested.
C.STAT	Command String Request Status. This byte field reflects the status of the current command line request. The bits in this field are initialized in accordance with the bit definitions listed below. The first bit is maintained by the routine invoked through the CSI$1 macro call. All the other bits in this field are maintained by the routine invoked through the CSI$2 macro call. CS.EQU - Indicates that an equal sign (=) has been detected in the current command line, signifying that the command line contains both output and input file specifiers. CS.NMF - Indicates that the current file specifier contains a filename string. Accordingly, control block offset locations C.FILD+2 and C.FILD (see below) are initialized with the address and the length (in bytes), respectively, of the command line segment containing the filename string. If no filename string is present, this bit is not set, and the filename string descriptors in the control block are cleared. CS.DIF - Indicates that the current file specifier contains a directory string. Thus, control block offset locations C.DIRD+2 and C.DIRD (see below) are initialized with the address and the length (in bytes), respectively, of the command line segment containing the directory string. If no directory string is present, this bit is not set. In this case, any residual non-zero values in the directory string descriptor cells which pertain to a previous command string request of like type (see C.TYPR above) are used by default. Thus, the last directory string encountered in a file specifier is used.

CS.DVF - Indicates that the current file specifier contains a device name string. Similarly, control block offset locations C.DEVD+2 and C.DEVD (see below) are initialized with the address and the length (in bytes), respectively, of the device name string. If no device name string is present, this bit is not set. Again, similar to CS.DIF above, any residual non-zero values in the device name descriptor cells which pertain to a previous command string request of like type are used by default. Thus, the last device name string encountered in a file specifier is used.

CS.WLD - Indicates that the current file specifier contains an asterisk (*), signalling the presence of a wildcard specification.

CS.MOR - Indicates that the current file specifier is terminated by a comma (,). The comma indicates that more file specifiers are to follow. If this bit is not set, it signifies that the end of the input or output file specifiers has been reached.

C.CMLD — Command Line Descriptor. This 2-word field is initialized with the address and the length (in bytes), respectively, of the compressed command line. In other words, the values returned to these cells constitute the output of CSI after scanning a file specifier and removing all non-significant characters from the string (i.e., nulls, blanks, tabs, and RUBOUTS).

The values contained in these cells are used by CSI as the descriptors of the compressed command line to be parsed (see CSI$2 macro call in section 6.2.3.2).

C.DSDS — Dataset Descriptor Pointer. This offset defines the address of the 6-word dataset descriptor in the CSI control block. This structure is functionally identical to the manually-created dataset descriptor detailed in section 2.4.1.

This symbol may be used to initialize offset location F.DSPT in the FDB associated with the file to be processed. Thus, FCS is able to retrieve requisite ASCII information from this structure for use in opening files.

Assembly-time initialization of F.DSPT in the associated FDB may be accomplished as follows:

```
FDOP$A  1,CSIBLK+C.DSDS
```

where "CSIBLK" is the address of the CSI control block, and "C.DSDS" represents the beginning address of the descriptor strings in the CSI control block (see C.DEVD, C.DIRD, and C.FILD below) identifying the requisite ASCII filename information.

Run-time initialization of F.DSPT in the associated FDB may also be accomplished through the dspt parameter of the FDOP$R macro call (see section 2.2.2) or the generalized OPEN$x macro call (see section 3.1).

C.DEVD — Device Name String Descriptor. This 2-word field contains the address (C.DEVD+2) and the length in bytes (C.DEVD) of the most recent device name string (of like request type) encountered in a file specifier.

C.DIRD — Directory String Descriptor. This 2-word field contains the address (C.DIRD+2) and the length in bytes (C.DIRD) of the most recent directory string (of like request type) encountered in a file specifier.

C.FILD — Filename String Descriptor. This 2-word field contains the address (C.FILD+2) and the length in bytes (C.FILD) of the filename string in the current file specifier.

If an error condition is detected by the command syntax analyzer during the syntactical analysis of a command line (see section 6.2.3.1 below), a segment descriptor is returned to this field, defining the address and the length of the command line segment in error.

C.SWAD — Current Switch Table Address. This word location contains the address of the switch descriptor table specified in the current CSI$2 macro call (see section 6.2.3.2).

C.MKW1 — CSI Mask Word 1. This word indicates the particular switch(es) present in the current file specifier after each invocation of the CSI$2 macro call. The switch mask for each of the defined switches encountered in a file specifier between delimiting commas is OR'ed into this location.

The mask for a switch is specified in the CSI$SW macro call (see section 6.2.4.1). When a switch is encountered in a file specifier for which a defined mask exists, the corresponding bits in C.MKW1 are set. By testing C.MKW1, the user can determine the particular combination of defined switches present in the current file specifier.

C.MKW2 — CSI Mask Word 2. This word provides a switch polarity indication for the user.

When a switch is present in a file specifer and that switch is not negated, the defined mask for that switch is OR'ed into C.MKW2 in the same manner as described above for C.MKW1. Conversely, when a switch is present in a file specifer and that switch is negated, the corresponding bits in C.MKW2 are cleared. Thus, for each switch indicated as being present by C.MKW1, the user can check the polarity of that switch by examining the corresponding bits in C.MKW2.

C.SIZE — Control Block Size Indicator. This symbol, which is defined during the expansion of the CSI$ macro, represents the size in bytes of the CSI control block.

6.2.3 CSI Run-Time Macro Calls

Two run-time macro calls are provided in CSI to invoke routines which perform the following functions:

CSI$1 - Initializes the CSI control block, analyzes the command line (normally contained in the GCML command line input buffer), removes non-significant characters from the line, and checks it for syntactic validity. This macro call also results in the initialization of certain cells in the CSI control block with the address and the length, respectively, of the validated and compressed command line.

CSI$2 - Parses a file specifier in the validated and compressed command line into its component device name, directory, and filename strings, and processes any associated switches and accompanying switch values. Also, certain cells in the CSI control block are initialized with the appropriate string descriptors for subsequent use by FCS in opening the specified file.

6.2.3.1 CSI$1 - Command Syntax Analyzer

The CSI$1 macro call results in the invocation of a routine called the command syntax analyzer. This routine analyzes a command line (which is normally read into the GCML command line input buffer) and checks it for syntactic validity. In addition, it compresses the file specifiers in the command line by removing all non-significant characters (i.e., nulls, tabs, blanks, and RUBOUTs). Finally, the command syntax analyzer initializes offset locations C.CMLD+2 and C.CMLD in the CSI control block (see section 6.2.2) with the address and the length (in bytes), respectively, of the validated and compressed command line. Each file specifier in the command line is then parsed into its component device name, directory, and filename strings during successive issuances of the CSI$2 macro call (see next section).

The CSI$1 macro call is issued in the following format:

```
CSI$1   csiblk,buff,len
```

where:

csiblk — represents the address of the CSI control block. If this parameter is not specified, R0 is assumed to contain the address of the CSI control block.

buff — represents the address of a command line input buffer. This parameter initializes CSI control block offset location C.CMLD+2, enabling CSI to retrieve the current command line from a command line input buffer.

If this parameter is not specified, the user must manually initialize CSI control block offset location C.CMLD+2 with the address of a command line input buffer before issuing the CSI$1 macro call. This may be accomplished through a statement similar to the following:

```
MOV     GCLBLK+G.CMLD+2,CSIBLK+C.CMLD+2
```

len — represents the length of the command line input buffer. Similarly, this parameter initializes CSI control block offset location C.CMLD, thus completing the 2-word descriptor which enables CSI to retrieve the current command line from the input buffer.

As with the "buff" parameter above, if this parameter is not specified, the user must manually initialize CSI control block offset location C.CMLD with the length of the command line input buffer before issuing the CSI$1 macro call. This may be accomplished as follows:

```
MOV     GCLBLK+G.CMLD,CSIBLK+C.CMLD
```

The combination of the buff and len parameters above enables CSI to analyze the current command line. Following the analysis of the command line, CSI updates offset location C.CMLD with the length of the validated and compressed command line.

If a syntactic error is detected during the validation of the command line, the C-bit in the Processor Status Word is set, and offset locations C.FILD+2 and C.FILD in the CSI control block (see section 6.2.2) are set to values which define the address and the length, respectively, of the command line segment in error.

Representative examples of the CSI$1 macro call follow:

```
CSI$1   #CSIBLK,#BUFF,#LEN

CSI$1   R0,GCLBLK+G.CMLD+2,GCLBLK+G.CMLD

CSI$1   #CSIBLK
```

The first example shows symbols which represent the address and the length of a command line to be analyzed (not necessarily the line contained in the GCML command line input buffer).

The second example assumes that R0 has been preset with the address of the CSI control block; the next two parameters are direct references to the command line descriptor words in the GCML control block.

Finally, the third example assumes that the required descriptor values are already present in offset locations C.CMLD+2 and C.CMLD of the control block (CSIBLK) as the result of prior action.

6.2.3.2 CSI$2 - Command Semantic Parser

The CSI$2 macro call results in the invocation of the command semantic parser. This routine uses the values in CSI control block offset locations C.CMLD+2 and C.CMLD as the address and the length, respectively, of the command line to be parsed. The referenced line is then parsed into its component device name, directory, and filename strings. In addition, 2-word descriptors for these strings are stored in a 6-word dataset descriptor in the CSI control block, beginning at offset location C.DSDS (see section 6.2.2). This field is functionally equivalent to the dataset descriptor created manually in the user program (see section 2.4.1).

The command semantic parser also decodes any switches and associated switch values present in a file specifier. If the user expects to encounter switches in the current file specifier, the command semantic parser decodes them, provided that the address of the appropriate switch descriptor table has been specified in the CSI$2 macro call (see below). The CSI switch definition macro calls are described in detail in section 6.2.4.

The CSI$2 macro call is specified in the following format:

```
CSI$2   csiblk,io,swtab
```

where:

csiblk — represents the address of the CSI control block. If this parameter is not specified, R0 is assumed to contain the address of the CSI control block.

io — represents a symbol which explicitly identifies the type of file specifier to be parsed. Either of two symbolic arguments may be specified in this parameter field, as follows:

INPUT - Indicates that the next input file specifier in the command line is to be parsed.

OUTPUT - Indicates that the next output file specifier in the command line is to be parsed.

Offset location C.TYPR in the CSI control block (see section 6.2.2) must be initialized, either manually or through the CSI$2 macro call, with the type of file specifier being requested. If other than the symbolic arguments defined above are specified in the CSI$2 macro call, an .ERROR directive is generated during assembly which causes the error message "INCORRECT REQUEST TO .CSI2" to be printed in the assembly listing. This diagnostic message alerts the user to the presence of an invalid "io" parameter in the CSI$2 macro call.

swtab — represents the address of the associated switch descriptor table for this particular issuance of the CSI$2 macro call. This optional parameter need be specified only when the user anticipates the presence of a switch in the file specifier that is to be decoded. Specifying this parameter presumes that the user has previously created a corresponding switch descriptor table in the program through the CSI$SW macro call (see section 6.2.4.1). In addition, if the switch to be decoded has any associated switch values, the user must also have created an associated switch value descriptor table in the program through the CSI$SV macro call (see section 6.2.4.2).

This parameter initializes offset location C.SWAD in the CSI control block (see section 6.2.2); if not specified, any residual non-zero value in this cell is used by default as the address of the switch descriptor table.

Offset location C.SWAD may also be initialized manually prior to issuing the CSI$2 macro call, as shown in the following statement:

```
MOV     #SWTAB,CSIBLK+C.SWAD
```

where "SWTAB" is the symbolic address of the associated switch descriptor table.

If an error condition occurs during the parsing of the file specifier, the C-bit in the Processor Status Word is set, and control is returned to the calling program. The possible error conditions that may occur during command line parsing include the following:

1. The request type is invalid, i.e., offset location C.TYPR in the CSI control block (see section 6.2.2) has been incorrectly initialized.

2. A switch is present in a file specifier, but the address of the switch descriptor table has not been specified in the CSI$2 macro call, or the switch descriptor table does not contain a corresponding entry for the switch.

3. An invalid switch value is present in the file specifier.

4. More values accompany a given switch in the file specifier than there are corresponding entries in the switch value descriptor table for decoding those values.

5. A negative switch is present in the file specifier, but the corresponding entry in the switch descriptor table does not allow the switch to be negated (see the nflag parameter of the CSI$SW macro call in the next section).

Examples of the CSI$2 macro call are shown below:

```
CSI$2   #CSIBLK,INPUT,#SWTBL

CSI$2   R0,OUTPUT,#SWTBL

CSI$2   #CSIBLK,INPUT
```

The first example shows a request to parse an input file specifier which may include an associated switch. The second example, which assumes that R0 presently contains the address of the CSI control block, will parse an output file specifier that likewise may include a switch. Finally, the last example is a request to parse an input file specifier and to disallow any accompanying switch(es).

6.2.4 CSI Switch Definition Macro Calls

The following macro calls must be issued at assembly-time to create the requisite switch descriptor tables in the program for processing switches that appear in a file specifier:

CSI$SW - Creates an entry in the switch descriptor table for a particular switch that the user expects to encounter in a file specifier.

CSI$SV - Creates a matching entry in the switch value descriptor table for the switch defined through the CSI$SW macro call above.

CSI$ND - Terminates a switch descriptor table or a switch value descriptor table created through the CSI$SW or the CSI$SV macro call, respectively.

These macro calls are described separately in the following sections.

6.2.4.1 CSI$SW - Create Switch Descriptor Table Entry

To process each switch that the user expects to encounter in a file specifier, a matching entry in the switch descriptor table must be defined. When the address of a switch descriptor table is specified in any particular issuance of the CSI$2 macro call (see section 6.2.3.2), the following processing occurs:

1. For each switch encountered in a file specifier, CSI searches the switch descriptor table for a matching entry. If the switch descriptor table address is not specified, or a matching entry is not found in the table for the switch, that switch is considered to be invalid. As a result, the C-bit in the Processor Status Word is set, any remaining switches in the file specifier are bypassed, and control is returned to the calling program.

2. If a matching entry is found in the switch descriptor table, mask word 1 in the CSI control block is set according to the defined mask for that switch (see C.MKW1, section 6.2.2).

3. The negation status of the switch is determined. If the switch is not negated, the corresponding bits in mask word 2 (C.MKW2) in the CSI control block are set according to the defined mask for that switch. If the switch is negated, and negation is not allowed, then the switch is considered to be invalid. In this case, the error sequence described in Step 1 above applies. However, if the switch is negated, and negation is allowed, then the corresponding bits in C.MKW2 are cleared.

   The negation flag for a switch is established through the nflag parameter of the CSI$SW macro call (see below).

4. If the address of the optional user mask word is not present in the corresponding switch descriptor table entry, i.e., if the mkw parameter has not been specified in the associated CSI$SW macro call (see below), switch processing continues with Step 7. If, however, the address of the optional mask word is specified, switch processing continues with Step 5.

5. If "SET" has been specified as the clear/set flag in the corresponding switch descriptor table entry, and the switch is not negated, then the corresponding bits in the optional mask word are set according to the defined mask for that switch. If, however, the switch is negated, the corresponding bits in the optional mask word are cleared.

   The clear/set flag is specified as the csflg parameter in the CSI$SW macro call (see below).

6. If "CLEAR" has been specified as the clear/set flag in the corresponding switch descriptor table entry, and the switch is not negated, the corresponding bits in the optional mask word are cleared. Conversely, if the switch is negated, the corresponding bits in the optional mask word are set.

7. If a switch value accompanies a switch in a file specifier, the associated switch value descriptor table created through the CSI$SV macro call (see next section) is used to decode the value. There must be at least as many entries in the switch value descriptor table as there are such values accompanying the switch in the file specifier. If the switch value descriptor table is incomplete, if an invalid switch value is encountered, or if the address of the switch value descriptor table is not present in the associated switch descriptor table, then the switch is considered to be invalid, and the error sequence described in Step 1 again applies.

   The address of the switch descriptor value table is specified as the vtab parameter in the CSI$SW macro call (see below).

The CSI$SW macro call is specified in the following format:

```
label:  CSI$SW  sw,mk,mkw,csflg,nflg,vtab
```

where:

label
: represents an optional symbol which names the resulting switch descriptor table entry and defines its address. In order to establish the address of a switch descriptor table, the first CSI$SW macro call issued in the program must include a label. This label allows the table to be referenced by other instructions in the program.

sw
: represents the 2-character alphabetic switch name that is to be stored in the resulting switch descriptor table entry. This parameter is required. If not specified, an .ERROR directive is generated during assembly which causes the error message "MISSING SWITCH NAME" to be printed in the assembly listing.

mk
: represents a user-defined mask for the switch specified through the sw parameter above. To enable CSI to indicate the presence of a given switch in a file specifier, a mask value for the switch must be defined, as shown below:

```
ASMSK = 1
NUMSK = 2
  .
  .
  .
VWMSK = 40000
XYMSK = 100000
```

where the (octal) value assigned by the user to each symbol defines a unique bit configuration that is to be set in CSI mask word 1 (C.MKW1) of the control block when a defined switch is encountered in a file specifier.

By specifying the appropriate symbol as the "mk" parameter in the CSI$SW macro call, the corresponding mask value is stored in the resulting switch descriptor table entry. Thus, a mechanism is established through which the user can determine the particular combination of switches present in a file specifier. For every matching entry found in the switch descriptor table, the corresponding bits are set in C.MKW1.

mkw represents the address of an optional user mask word. If specified, this parameter causes CSI to set or clear bits in a word reserved in the user program. This word provides additional information to the user regarding the clear/set flags in the switch descriptor table in relation to the negation status of switches encountered in a file specifier.

Such an optional word may be reserved through a statement logically equivalent to that shown below:

```
MASKX:  .WORD   0
```

CSI then manipulates the bits in this word, as described in the sequence of switch processing operations at the beginning of this section.

csflg represents a symbolic argument which specifies the clear/set flag for a given switch. This parameter is optional; if not specified, SET is assumed (see below). Either one of two symbolic arguments may be specified for this parameter, as follows:

CLEAR - Indicates that the bits in the optional user mask word corresponding to the switch mask, are to be cleared provided that the switch is not negated. (If the switch is negated, the bits are set.)

SET - Indicates, conversely, that the bits in the optional user mask word corresponding to the switch mask are to be set provided that the switch is not negated. (If the switch is negated, the bits are cleared.)

If other than one of the above arguments is specified, an .ERROR directive is generated during assembly which causes the error message "INVALID SET/CLEAR SPEC" to be printed in the assembly listing.

nflg represents a symbolic argument which specifies an optional negation flag for the switch. If this parameter is specified, it indicates that the switch is allowed to be negated, e.g., /-LI or /NOLI.

If this parameter is specified as other than "NEG," an .ERROR directive is generated during assembly which causes the error message "INVALID NEGATE SPEC" to be printed in the assembly listing. If this parameter is not specified, the default assumption is that switch negation is not allowed.

vtab represents the address of the switch descriptor table associated with this switch. This optional parameter, if specified, allows CSI to decode any switch values accompanying the switch, provided that an associated switch value descriptor table entry has been defined for that switch. The switch value descriptor table is defined through the CSI$SV macro call, as described in the next section.

The format of the switch descriptor table entry that results from the issuance of the CSI$SW macro call is shown in Figure 6-2 below. One such switch entry must be defined for each switch appearing in the file specifier that the user intends to recognize. Each switch descriptor table entry consists of four words. The low-order byte of the first word contains the first character of the switch name; the high-order byte of this word contains the second character of the switch name. The second word contains the mask defined for the switch. The third word contains the address of the optional user mask word to receive the resultant value of switch processing. Finally, the fourth word contains the address of the switch value descriptor table associated with the switch.

16	0
SWITCH NAME CHARACTER 2	SWITCH NAME CHARACTER 1
MASK WORD FOR THIS SWITCH	
ADDRESS OF WORD TO BE MASKED *	
ADDRESS OF SWITCH VALUE TABLE **	

*If the low-order bit in this word is one (1), it indicates that the optional user mask word action is "CLEAR;" if it is zero (0), it indicates that the action is "SET."

**If the low-order bit in this word is one (1), it indicates that the switch may be negated.

Figure 6-2
Format of Switch Descriptor Table Entry

The following example shows a 2-entry switch descriptor table created through successive CSI$SW macro calls:

```
ASSWT:  CSI$SW  AS,ASMSK,MASKX,SET,,ASVTBL

        CSI$SW  NU,NUMSK,MASKX,CLEAR,NEG,NUVTBL

        CSI$ND                    ;END OF SWITCH DESCRIPTOR TABLE.
```

The first statement results in the creation of an entry in the switch descriptor table for the switch /AS. The second parameter is an equated symbol which defines the switch mask, and the third parameter (MASKX) is the address of an optional user mask word (see the mkw parameter above). The fourth parameter indicates that the bits in MASKX which correspond to the switch mask are to be set. The fifth parameter (the negation flag) is null. Finally, the last parameter is the address of the associated switch value descriptor table.

The second statement results in the creation of a switch descriptor table entry for the switch /NU. In contrast to the first statement, the fourth parameter (CLEAR) indicates that the bits in the optional user mask word (MASKX) which correspond to the switch mask are to be cleared. The fifth parameter (NEG) allows the switch to be negated, and the last parameter is the address of the value table associated with this switch.

Note that the switch descriptor macros are terminated with the CSI$ND macro call (see section 6.2.4.3).

6.2.4.2 CSI$SV - Create Switch Value Descriptor Table Entry

For every switch value that the user expects to encounter in connection with a given switch in a file specifier, a corresponding switch value descriptor table entry must be defined in the user program in order to allow the switch value(s) to be decoded. The CSI$SV macro call is provided for this purpose. When issued, this macro call results in the creation of a 2-word entry in the switch value descriptor table. The format of this table is shown in Figure 6-3 below.

The CSI$SV macro call is specified in the following format:

```
        CSI$SV  type,adr,len,vtab
```

where:

type
: represents a symbolic argument which specifies the conversion type for the switch value. Any one of four symbolic values may be specified in this parameter field to indicate the conversion type for the accompanying switch value. The possible conversion type arguments include the following:

  ASCII - Indicates that the switch value is to be treated as an ASCII string.

  NUMERIC - Indicates that a numeric switch value is to be converted to binary using octal as a default conversion radix.

  OCTAL - Indicates that a numeric switch value is to be converted to binary using octal as a default conversion radix.

  DECIMAL - Indicates that a numeric switch value is to be converted to binary using decimal as a default conversion radix.

  If any value other than those defined above is specified, an .ERROR directive is generated during assembly which causes the error message "INVALID CONVERSION TYPE" to be printed in the assembly listing. If none of the above parameters is specified, ASCII is assumed by default.

adr
: represents the address of the user program location which is to receive the resultant switch value at the conclusion of switch processing. This parameter is required; if not specified, an

.ERROR directive is generated during assembly which causes the error message "VALUE ADDRESS MISSING" to be printed in the assembly listing.

len — represents a numeric value which defines the length (in bytes) of the area which is to receive the switch value resulting from switch processing. This parameter is also required; if not specified, an .ERROR directive is also generated during assembly which causes the error message "LENGTH MISSING" to be printed in the assembly listing.

vtab — represents a symbol which names the switch value descriptor table and defines its address. This parameter is optional. The vtab parameter may also be specified in the CSI$SW macro call (see section 6.2.4.1) when the user anticipates the presence of a switch value in a file specifier that is to be decoded.

The format of a switch value descriptor table entry that results from the CSI$SV macro call is shown in Figure 6-3 below.

The low-order byte of the first word in the switch value descriptor table indicates whether the conversion type is ASCII or numeric. Bit 0 in this byte is set if "ASCII" is specified, bit 1 is set if "NUMERIC" or "OCTAL" is specified, and bit 2 is set if "DECIMAL" is specified. The high-order byte of this word indicates the maximum allowable length (in bytes) of the switch value.

If the conversion type is "ASCII," the len parameter reflects the maximum number of ASCII characters that can be deposited in the area defined through the adr parameter. The high-order byte of the first word in the switch value table then reflects the maximum length of the ASCII string. If the number of characters in the switch value exceeds the specified length, the extra characters are simply ignored. If, however, the actual number of ASCII characters present in the switch value falls short of the specified length, the remaining portion of the area receiving the resultant value is null padded.

If the conversion type is "NUMERIC," the resultant binary value is assumed to be two bytes in length, and the area receiving the value is assumed to be word-aligned. A numeric switch value is always evaluated as a signed number; an overflow into the high order bit (bit 16) results in an error condition.

On numeric conversions, the default conversion type specified for a switch value can be overridden by means of a pound sign (#) or a dot (.). A numeric value preceded by a pound sign (e.g., #10) forces the conversion type to octal; a numeric value followed by a dot (e.g., 10.) forces the conversion type to decimal. Note also that a numeric switch value may be preceded by a plus sign (+) or a minus sign (-). The plus sign is the default assumption. If an explicit octal switch value is specified using the pound sign (#), as described above, the arithmetic sign indicator (+ or -), if included, must precede the pound sign (e.g., -#10).

16	0
SWITCH VALUE LENGTH	CONVERSION TYPE
ADDRESS OF LOCATION RECEIVING SWITCH RESULT	

Figure 6-3
Format of Switch Value Descriptor Table Entry

Representative CSI$SV macro calls are shown below:

```
ASVTBL: CSI$SV  ASCII,ASVAL,3

        CSI$SV  ASCII,ASVAL+4,3

        CSI$ND                      ;END OF SWITCH VALUE TABLE.

NUVTBL: CSI$SV  OCTAL,NUVAL,2

        CSI$SV  DECIMAL,NUVAL+2,2

        CSI$ND                      ;END OF SWITCH VALUE TABLE.
```

In all cases above, the first parameter in the CSI$SV macro call defines the conversion type. The next two parameters, in all cases, define the address and the length of the user program location to receive the resultant switch value.

The required storage for the first switch value table above may be reserved as follows:

```
ASVAL   .BLKW   4                   ;ASCII VALUE STORAGE.
```

The required storage for the second switch value table may be similarly reserved through the following statement:

```
NUVAL:  .BLKW   2                   ;NUMERIC VALUE STORAGE.
```

Note again that switch value tables are terminated with the CSI$ND macro call.

6.2.4.3 CSI$ND - Define End of Descriptor Table

Switch descriptor tables and switch value descriptor tables must be terminated with a 1-word end-of-table entry. This word, which contains zero (0), may be created through the CSI$ND macro call.

This macro call takes no arguments, as shown below:

```
        CSI$ND
```

The examples near the end of the preceding section illustrate the use of this macro call.

CHAPTER 7

SPOOLING

FCS provides facilities at both the macro and subroutine level to queue files for subsequent printing.

7.1 PRINT$ MACRO CALL

A task issues the PRINT$ macro call to queue a file for printing on a specified device. The specified device must be a unit-record, carriage-controlled device such as a line printer or terminal. If the device is not specified, LP: is used.

The file to be spooled must be open when the PRINT$ macro is issued. PRINT$ closes the file. Error returns differ from normal FCS conventions and are described in Section 7.3.

The PRINT$ macro call has the following format:

```
PRINT$ fdb,err,,dev,(1)unit,(1)pri,(1)forms,(1)copies,(1)presrv(1)
```

fdb
: represents the address of the associated FDB.

err
: represents the address of an optional user-coded error handling routine. See Section 7.3.

The following parameters are not applicable to RSX-11M.

dev
: represents the 2-character device mnemonic of the device on which the file is to be printed. If dev is not specified, LP: is used by default.

unit
: represents the unit number of the device on which the file is to be printed. If unit is not specified, unit 0 is used by default.

The following parameters are used only by the IAS and RSX-11D multiple device despoolers. See the discussion below.

pri
: represents a number in the range 1 through 250 to indicate the priority of the request. The priority is used to determine the order in which spooled files are dequeued for printing. If pri is omitted, the task's priority is used by default.

---

(1) Does not apply to RSX-11M.

forms — represents the specific form type on which the file is to be printed as indicated by a number in the range 0 through 6. This parameter must be specified as a single integer without a preceding number sign (#). The numbers 0 through 6 are associated with the various forms for an installation by the system manager. If forms is omitted, form type 0 is used by default.

copies — represents a number in the range 1 through 32 to indicate the number of copies of the file to be produced. The number of copies must be specified as a 1- or 2-digit integer without a preceding number sign (#). If copies is omitted, one copy is printed.

presrv — should be specified if the file is not to be deleted after it is printed. Any parameter value results in the file's being preserved after printing.

The following points do not apply to RSX-11M.

1. A blank parameter is present between err and unit thus requiring an additional comma. This parameter exists to provide compatibility between RSX-11D Version 4 and RSX-11D Version 6.

2. The number of parameters that are meaningful for RSX-11D is determined by whether the single device despooler or the multiple device despooler is available in the system. The difference between the two despoolers is described in the RSX-11D User's Guide and the RSX-11D System Manager's Guide. In IAS, only the multiple device despooler is supported. This is described in the IAS System Management `Guide. The following parameters are used by the multiple device despooler and ignored by the single device despooler.

   pri

   forms

   copies

   presrv

7.2 .PRINT SUBROUTINE

The .PRINT subroutine is called to queue a file for printing. The file must be open when .PRINT is called. The .PRINT routine closes the file.

R0 must contain the address of the associated FDB.

The file is printed on LP:.

Section 7.3 describes error handling for the .PRINT file control routine.

7.3 ERROR HANDLING

The error returns provided in conjunction with PRINT$ and .PRINT differ from the standard FCS error returns in that error codes are placed in F.ERR or in the directive status word depending on when the failure occurred.

If the failure is FCS related, e.g., the PRINT$ macro cannot close the file, the C bit is set and F.ERR contains the error code.

If the failure is related to the SEND/REQUEST directive that queues the file, the C bit is set and the directive status word contains an error code.

Directive status word error codes are provided in the Executive Reference Manual of the host operating system.

Normally, user-coded error routines, upon determining that the C bit is set, should test F.ERR first and then test the directive status word.

APPENDIX A

FILE DESCRIPTOR BLOCK

A file descriptor block contains file information that is used by FCS and the file control primitives. The layout of an FDB is illustrated on the following page; Table A-1 defines the offset locations within the FDB.

The offset names in the file descriptor block may be defined either locally or globally, as shown below:

```
FDOF$L                  ;DEFINE OFFSETS LOCALLY.

FDOFF$  DEF$L           ;DEFINE OFFSETS LOCALLY.

FDOFF$  DEF$G           ;DEFINE OFFSETS GLOBALLY.
```

NOTE

When referring to FDB locations, it is essential to use the symbolic offset names, rather than the actual address of such locations. The position of information within the FDB may be subject to change from release to release, while the offset names themselves remain constant.

FILE DESCRIPTOR BLOCK

File Attribute Section	F.RATT	F.RTYP
	F.RSIZ	
	F.HIBK	
	F.EFBK	
Record or Block Access Section	F.FFBY	
	F.RCTL	F.RACC
	F.BKDS or F.URBD	
	F.NRBD or	
	F.BKST and F.BKDN	
	F.OVBS or F.NREC	
	F.EOBB	
	F.RCNM or	
	F.CNTG and F.STBK	
File Open Section	F.ALOC	
	F.FACC	F.LUN
	F.DSPT	
	F.DFNB	
Block Buffer Section	F.BKP1	F.EFN or F.BKEF
	FERR+1	F.ERR
	F.MBC1	F.MBCT
	F.BGBC	F.MBFG
	F.VBSZ	
	F.BBFS	
	F.BKVB or F.VBN	
	F.BDB	
	F.SPDV	
	not used	F.SPUN
	F.ACTL(1)	
	F.CHR(1)	
	F.FNB	

(1) Not used by RSX-11M.

Table A-1
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.RTYP	1	Record type byte. This byte is set, as follows, to indicate the type of records for the file: Bit 0 = 1 to indicate fixed-length records (R.FIX). Bit 1 = 1 to indicate variable-length records (R.VAR).
F.RATT	1	Record attribute byte. Bits 0 through 3 are set to indicate record attributes, as follows: Bit 0 = 1 to indicate that the first byte of a record is to contain a FORTRAN carriage-control character (FD.FTN); otherwise, it is 0. Bit 1 = 1 to indicate for a carriage-control device that a line feed is to be performed before the line is printed and a carriage return is to be performed after the line is printed (FD.CR); otherwise, it is 0. Bit 2 is not used. Bit 3 = 1 to indicate that records cannot cross block boundaries (FD.BLK); otherwise, it is 0.
F.RSIZ	2	Record size word. This location contains the size of fixed-length records or indicates the size of the largest record that currently exists in a file of variable-length records.
F.HIBK	4	Indicates the highest virtual block number allocated.
F.EFBK	4	Contains the end-of-file block number.

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.FFBY	2	Indicates the first free byte in the last block or the maximum block size for magnetic tape.
F.RACC	1	Record access byte. Bits 0 through 3 of this byte define the record access modes, as follows: Bit 0 = 1 to indicate READ$/WRITE$ mode (FD.RWM); otherwise, it is 0 to indicate GET$/PUT$ mode. Bit 1 = 1 to indicate random access mode (FD.RAN) for GET$/PUT$ record I/O; otherwise, it is 0 to indicate sequential access mode. Bit 2 = 1 to indicate locate mode (FD.PLC) for GET$/PUT$ record I/O; otherwise, it is 0 to indicate move mode. Bit 3 = 1 to indicate that PUT$ operation in sequential mode does not truncate the file (FD.INS); otherwise, it is 0 to indicate that PUT$ operation in sequential mode truncates the file.
F.RCTL	1	Device characteristics byte. Bits 0 through 5 define the characteristics of the device associated with the file, as follows: Bit 0 = 1 to indicate a record-oriented device (FD.REC), e.g., a Teletype or line printer; a value of 0 indicates a block-oriented device, e.g., a disk or DECtape. Bit 1 = 1 to indicate a carriage control device (FD.CCL); otherwise, it is 0. Bit 2 = 1 to indicate a teleprinter device (FD.TTY); otherwise, it is 0. Bit 3 = 1 to indicate a directory device (FD.DIR); otherwise, it is 0.

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.RCTL (cont.)		Bit 4 = 1 to indicate a single directory device (FD.SDI). An MFD is used, but no UFD's are present. Bit 5 = 1 to indicate a block-oriented device that is inherently sequential in nature (FD.SQD). A record-oriented device is assumed to be sequential in nature; therefore, this bit is not set for such devices.
F.BKDS	4	Contains the block I/O buffer descriptor.
or		
F.URBD		Contains the user record buffer descriptor.
F.NRBD	4	Contains the next record buffer descriptor.
or		
F.BKST	2	Contains the address of the I/O status block for block I/O.
and		
F.BKDN	2	Contains the address of the AST service routine for block I/O.
F.OVBS	2	Override block buffer size. This field has meaning only before the file is opened.
or		
F.NREC	2	Contains the number of the next record in the block.
F.EOBB	2	Contains a value defining the end of the block buffer.
F.RCNM	4	Contains the number of the record for random access operations.
or		
F.CNTG	2	Contains a numeric value defining the number of blocks to be allocated in creating a new file. This cell has meaning only before the file is opened. A value of 0 means leave the file empty; a positive value means

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.CNTG (cont.) and		allocate the specified number of blocks as a contiguous area and make the file contiguous; a negative value means allocate the specified number of blocks as a noncontiguous area and make the file noncontiguous.
F.STBK	2	Contains the address of the statistics block in the user program.
F.ALOC	2	Number of blocks to be allocated when the file must be extended. This cell has meaning only before the file is opened. A positive (+) value indicates contiguous extend, and a negative (-) value indicates noncontiguous extend.
F.LUN	1	Contains the logical unit number associated with the FDB.
F.FACC	1	File access byte. This byte indicates the access privileges for a file, as summarized below: Bit 0 = 1 if the file is accessed for read only (FA.RD). Bit 1 = 1 if the file is accessed for writing (FA.WRT). Bit 2 = 1 if the file is accessed for extending (FA.EXT). Bit 3 = 1 if a new file is being created (FA.CRE); otherwise, it is zero (0) to indicate an existing file. Bit 4 = 1 if the file is a temporary file (FA.TMP). Bit 5 = 1 if the file is opened for shared access (FA.SHR).

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.FACC (cont.)		If Bit 3 above is zero (0): Bit 6 = 1 if an existing file is being appended (FA.APD). If Bit 3 above is one (1): Bit 6 = 1 if not superseding an existing file at file-create time (FA.NSP).
F.DSPT	2	Contains the dataset descriptor pointer.
F.DFNB	2	Contains the default filename block pointer.
F.BKEF or F.EFN	1	Contains the block I/O event flag. Contains the record I/O event flag.
F.BKP1	1	Contains bookkeeping bits for FCS internal control.
F.ERR	1	Error return code byte. A negative value indicates an error condition.
F.ERR+1	1	Used in conjunction with F.ERR above. If F.ERR is negative, the following applies: F.ERR+1 = 0 to indicate that error code is an I/O error code (see IOERR$ error codes in Appendix I). F.ERR+1 = negative value to indicate that error code is a Directive Status Word error code (see DRERR$ error codes in Appendix I).
F.MBCT	1	Indicates the number of buffers to be used for multiple buffering.

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.MBC1	1	Indicates the actual number of buffers currently in use.
F.MBFG	1	Multiple buffering flag word. Contains either one of the multiple buffering flags, as follows: Bit 0 = 1 to indicate read-ahead (FD.RAH). Bit 1 = 1 to indicate write-behind (FD.WBH).
F.BGBC	1	Big buffer block count in number of blocks (not implemented).
F.VBSZ	2	Device buffer size word. Contains the virtual block size (in bytes).
F.BBFS	2	Indicates the block buffer size.
F.BKVB or	4	Contains the address of the virtual block number in the user program for block I/O.
F.VBN		Contains the virtual block number.
F.BDB	2	Contains the address of the block buffer descriptor block. This location always contains a non-zero value if the file is open and zero (0) if the file is closed.
F.SPDV	2	Spooler output device designation (IAS and RSX-11D only).
F.SPUN	1	Spooler output unit designation (IAS and RSX-11D only).
Spare	1	Not used.
F.ACTL	2	The low order byte of this word indicates the number of retrieval pointers to be used for the file. The control bits are in the high order byte and are defined as follows.

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
		Bit 15 = 1 to specify that control information is to be taken from F.ACTL (FA.ENB). Bit 12 = 0 to cause positioning to the end of a magnetic tape volume set upon open or close. Bit 12 = 1 to cause positioning of a magnetic tape volume set to just past the most recently closed file when the next file is opened (FA.POS). Bit 11 = 1 to cause a magnetic tape volume set to be rewound upon open or close (FA.RWD). Bit 9 = 1 to cause a file not to be locked if it is not properly closed when accessed for write (FA.DLK).
F.CHR	2	Reserved for system use.
F.FNB	-	Defines the beginning address of the filename block portion of the FDB.

APPENDIX B

FILENAME BLOCK

The format of a filename block is illustrated in Figure B-1. The offsets within the filename block are described in Table B-1.

The offset names in a filename block may be defined either locally or globally, as shown below:

```
NBOF$L              ;DEFINE OFFSETS LOCALLY.

NBOFF$  DEF$L       ;DEFINE OFFSETS LOCALLY.

NBOFF$  DEF$G       ;DEFINE OFFSETS GLOBALLY.
```

NOTE

When referring to filename block locations, it is essential to use the symbolic offset names, rather than the actual addresses of such locations. The position of information within the filename block may be subject to change from release to release, while the offset names themselves remain constant.

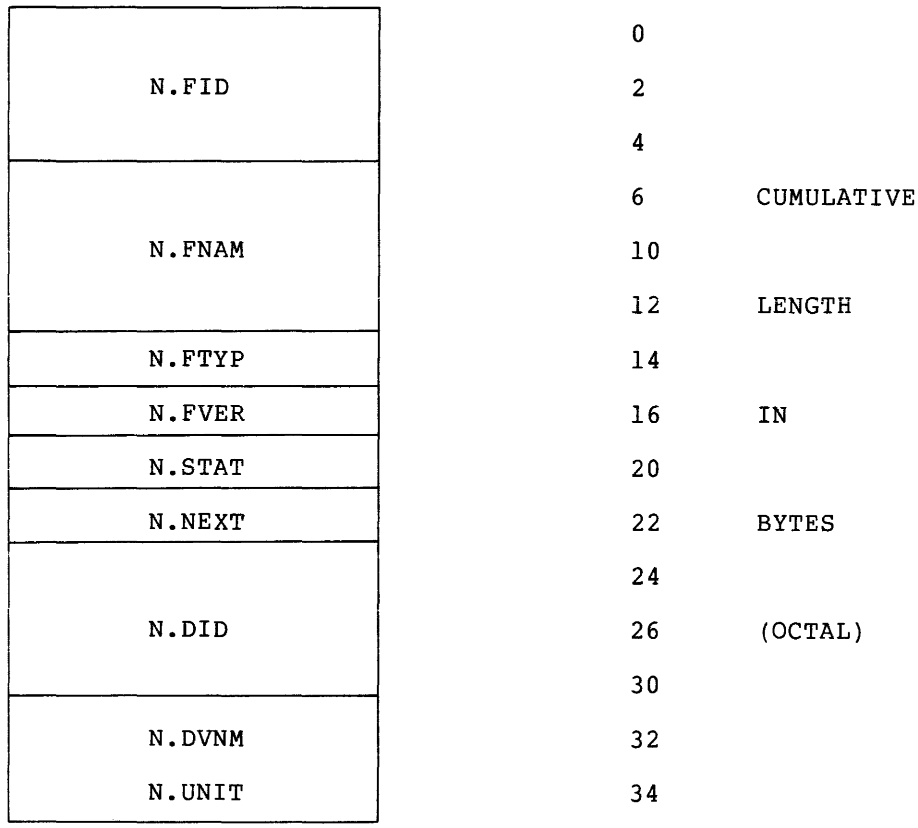

Figure B-1
Filename Block Format

Table B-1
Filename Block Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
N.FID	6	File identification field.
N.FNAM	6	Filename field; specified as nine characters which are stored in Radix-50 format.
N.FTYP	2	File type field; specified as three characters which are stored in Radix-50 format.
N.FVER	2	File version number field (binary).
N.STAT	2	Filename block status word (see bit definitions in Table B-2).
N.NEXT	2	Context for next .FIND operation.
N.DID	6	Directory identification field.
N.DVNM	2	ASCII device name field.
N.UNIT	2	Unit number field (binary).

The bit definitions of the filename block status word (N.STAT) in the FDB and their significance are described in Table B-2.

Those symbols marked with an asterisk (*) in Table B-2 indicate bits that are set if the associated information is supplied through an ASCII dataset descriptor.

Table B-2
Filename Block Status Word (N.STAT)

SYMBOL	VALUE (in octal)	MEANING
NB.VER*	1	Set if explicit file version number is specified.
NB.TYP*	2	Set if explicit file type is specified.
NB.NAM*	4	Set if explicit filename is specified.
NB.SVR	10	Set if wildcard file version number is specified.
NB.STP	20	Set if wildcard file type is specified.
NB.SNM	40	Set if wildcard filename is specified.
NB.DIR*	100	Set if explicit directory string (UIC) is specified.
NB.DEV*	200	Set if explicit device name string is specified.
NB.SD1	400	Set if group portion of UIC contains wildcard specification.
NB.SD2	1000	Set if owner portion of UIC contains wildcard specification.

APPENDIX C

SUMMARY OF I/O-RELATED SYSTEM DIRECTIVES

Table C-1 contains a summary of the I/O-related system directives in alphabetical order for ready reference. The parameters that may be specified with a directive are also described in the order of their appearance in the directive. These directives are described in detail in the Executive Reference Manual of the host operating system.

Table C-1
Summary of I/O-Related System Directives

DIRECTIVE	FUNCTION AND PARAMETERS
ALUN$	Assigns a logical unit number to a physical device: lun = Logical unit number. dev = Physical device name (2 ASCII characters). unt = Physical device unit number.
GLUN$	Fills a 6-word buffer with information about a physical unit: lun = Logical unit number. buf = Address of a 6-word buffer in which the LUN information is to be stored.
GMCR$	Transfers an 80-byte MCR/PDS command line to the issuing task. No parameters are required in this directive.
QIO$	Places an I/O request in the device queue associated with the specified logical unit number: fnc = I/O function code. lun = Logical unit number. efn = Event flag number. pri = Priority of the request (IAS and RSX-11D only). isb = Address of the I/O status block. ast = Entry point address of the AST service routine. prl = Parameter list in the form <P1,....,P6>.

Table C-1 (Cont.)
Summary of I/O-Related System Directives

DIRECTIVE	FUNCTION AND PARAMETERS
RCVD$	Receives a 13-word data block that has been queued (FIFO) by a send data directive (see SDAT$ and SDRQ$ below). tsk = Name of the sending task. This field is ignored by RSX-11M. The tsk parameter is specified as a null value (,) in RSX-11M for compatibility with IAS and RSX-11D (see the description of the RCVD$ directive in the RSX-11M Executive Reference Manual). buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block).
RCVS$	Receives a 13-word data block, if queued by a send data directive (see SDAT$ AND SDRQ$ below), or suspends task if no data is queued: tsk = Name of the sending task. buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block). This directive is not supported in RSX-11M.
RCVX$	Receives a 13-word data block, if queued by a send data directive (see SDAT$ and SDRQ$ below), or exits if data is not queued for the task: tsk = Name of the sending task. This field is ignored by RSX-11M. The tsk parameter is specified as a null value (,) in RSX-11M for compatibility with IAS and RSX-11D (see the description of the RCVX$ directive in the RSX-11M Executive Reference Manual. buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block).

Table C-1 (Cont.)
Summary of I/O-Related System Directives

DIRECTIVE	FUNCTION AND PARAMETERS
SDAT$	Queues (FIFO) a 13-word block of data for a task to receive: tsk = Name of the receiving task. buf = Address of the 13-word data buffer. efn = Event flag number.
SDRQ$	Queues (FIFO) a 13-word block of data for a task to receive; also requests or resumes the execution of the receiving task: tsk = Name of the receiving task. par = Partition name of the receiving task. pri = Priority of the request. ugc = UIC group code. upc = UIC owner code. buf = Address of the 13-word data buffer. efn = Event flag number. This directive is not supported in RSX-11M.

APPENDIX D

SAMPLE PROGRAMS

The sample programs that follow read records from an input device, strip off any blanks to the right of the data portion of the record, and write the data record on an output device. While the programs are intended primarily for card reader input and printer output, device independence is maintained.

The main program is CRCOPY; CRCOPA and CRCOPB are variations. CRCOPA uses a dataset descriptor instead of the default filename block used in CRCOPY. CRCOPB uses run-time initialization of the FDB.

```
        .TITLE  CRCOPY          ;CARD READER COPY ROUTINE
        .MCALL  FDBDF$,FDAT$A,FDRC$A,FDOP$A,NMBLK$,FSRSZ$
        .MCALL  OPEN$R,OPEN$W,GET$,PUT$,CLOSE$,EXIT$S
        .MCALL  FINIT$
        INLUN=3         ;ASSIGN CR OR FILE DEVICE
        OUTLUN=4                ;ASSIGN TO OUTPUT DEVICE
        FSRSZ$  2
FDBOUT: FDBDF$                  ;ALLOCATE SPACE FOR OUTPUT FDB
        FDAT$A  R.VAR,FD.CR     ;INIT FILE ATTRIBUTES
        FDRC$A  ,RECBUF,8Ø.     ;INIT RECORD ATTRIBUTES
        FDOP$A  OUTLUN,,OFNAM   ;INIT FILE OPEN SECTION
FDBIN:  FDBDF$                  ;ALLOCATE SPACE FOR INPUT FDB
        FDRC$A  ,RECBUF,8Ø.     ;INIT RECORD ATTRIBUTES
        FDOP$A  INLUN,,IFNAM    ;INIT FILE OPEN SECTION
RECBUF: .BLKB   8Ø.             ;RECORD BUFFER
OFNAM:  NMBLK$  OUTPUT,DAT      ;OUTPUT FILENAME
IFNAM:  NMBLK$  INPUT,DAT       ;INPUT FILENAME
START:  FINIT$                  ;INIT FILE STORAGE REGION
        OPEN$R  #FDBIN          ;OPEN THE INPUT FILE
        BCS     ERROR           ;BRANCH IF ERROR
        OPEN$W  #FDBOUT ;OPEN THE OUTPUT FILE
        BCS     ERROR           ;BRANCH IF ERROR
GTREC:  GET$    #FDBIN          ;NOTE - URBD IS ALL SET UP
        BCS     CKEOF           ;ERROR SHOULD BE EOF INDICATION
        MOV     F.NRBD(RØ),R1   ;R1=SIZE OF RECORD READ
        MOV     #RECBUF,R2
        ADD     R1,R2           ;R2=ADDRESS OF LAST BYTE+1
1Ø$:    CMPB    #4Ø,-(R2)       ;STRIP TRAILING BLANKS
        BNE     PTREC
        SOB     R1,1Ø$
;AT THIS POINT, R1 CONTAINS THE STRIPPED SIZE OF THE
;RECORD TO BE WRITTEN.  IF THE CARD IS BLANK,
;A ZERO-LENGTH RECORD IS WRITTEN.
PTREC:  PUT$    #FDBOUT,,R1     ;R1 IS NEEDED TO SPECIFY
        BCC     GTREC           ;THE RECORD SIZE.
ERROR:  NOP                     ;ERROR CODE GOES HERE
CKEOF:  CMPB    #IE.EOF,F.ERR(RØ) ;END OF FILE?
        BNE     ERROR           ;BRANCH IF OTHER ERROR
        CLOSE$  RØ              ;CLOSE THE INPUT FILE
        BCS     ERROR
        CLOSE$  #FDBOUT         ;CLOSE THE OUTPUT FILE
        BCS     ERROR
        EXIT$S                  ;ISSUE EXIT DIRECTIVE
        .END    START
```

```
        .TITLE  CRCOPY          ;CARD READER COPY ROUTINE
        .MCALL  FDBDF$,FDAT$A,FDRC$A,FDOP$A,NMBLK$,FSRSZ$
        .MCALL  OPEN$R,OPEN$W,GET$,PUT$,CLOSE$,EXIT$S
        .MCALL  FINIT$
        INLUN=3         ;ASSIGN CR OR FILE DEVICE
        OUTLUN=4                ;ASSIGN TO OUTPUT DEVICE
        FSRSZ$  2
FDBOUT: FDBDF$
        FDAT$A  R.VAR,FD.CR
        FDRC$A  ,RECBUF,80.
        FDOP$A  OUTLUN,OFDSPT
FDBIN:  FDBDF$
        FDRC$A  ,RECBUF,80.
        FDOP$A  INLUN,IFDSPT
RECBUF: .BLKB   80.
OFDSPT: .WORD   0,0             ;DEVICE DESCRIPTOR
        .WORD   0,0             ;DIRECTORY DESCIRPTOR
        .WORD   ONAMSZ,ONAM     ;FILENAME DESCRIPTOR
IFDSPT: .WORD   0,0             ;DEVICE DESCRIPTOR
        .WORD   0,0             ;DIRECTORY DESCRIPTOR
        .WORD   INAMSZ,INAM     ;FILENAME DESCRIPTOR
ONAM:   .ASCII  /OUTPUT.DAT/
        ONAMSZ=.-ONAM
        .EVEN
INAM:   .ASCII  /INPUT.DAT/
        INAMSZ=.-INAM
        .EVEN
START:  FINIT$                  ;INIT FILE STORAGE REGION
        OPEN$R  #FDBIN          ;OPEN THE INPUT FILE
        BCS     ERROR           ;BRANCH IF ERROR
        OPEN$W  #FDBOUT ;OPEN THE OUTPUT FILE
        BCS     ERROR           ;BRANCH IF ERROR
GTREC:  GET$    #FDBIN          ;NOTE - URBD IS ALL SET UP
        BCS     CKEOF           ;ERROR SHOULD BE EOF INDICATION
        MOV     F.NRBD(R0),R1   ;R1=SIZE OF RECORD READ
        MOV     #RECBUF,R2
        ADD     R1,R2           ;R2=ADDRESS OF LAST BYTE+1
10$:    CMPB    #40,-(R2)       ;STRIP TRAILING BLANKS
        BNE     PTREC
        SOB     R1,10$
;AT THIS POINT, R1 CONTAINS THE STRIPPED SIZE OF THE
;RECORD TO BE WRITTEN.  IF THE CARD IS BLANK,
;A ZERO-LENGTH RECORD IS WRITTEN.
PTREC:  PUT$    #FDBOUT,,R1     ;R1 IS NEEDED TO SPECIFY
        BCC     GTREC           ;THE RECORD SIZE.
ERROR:  NOP                     ;ERROR CODE GOES HERE
CKEOF:  CMPB    #IE.EOF,F.ERR(R0) ;END OF FILE?
        BNE     ERROR           ;BRANCH IF OTHER ERROR
        CLOSE$  R0              ;CLOSE THE INPUT FILE
        BCS     ERROR
        CLOSE$  #FDBOUT         ;CLOSE THE OUTPUT FILE
        BCS     ERROR
        EXIT$S                  ;ISSUE EXIT DIRECTIVE
        .END    START
```

```
        .TITLE  CRCOPB            ;CARD READER COPY ROUTINE
        .MCALL  FDBDF$,FDAT$A,FDRC$A,FDOP$A,NMBLK$,FSRSZ$
        .MCALL  OPEN$R,OPEN$W,GET$,PUT$,CLOSE$,EXIT$S
        .MCALL  FINIT$,FDAT$R
        INLUN=3         ;ASSIGN CR OR FILE DEVICE
        OUTLUN=4                  ;ASSIGN TO OUTPUT DEVICE
        FSRSZ$  2
FDBOUT: FDBDF$
FDBIN:  FDBDF$
RECBUF: .BLKB   80.
OFDSPT: .WORD   0,0               ;DEVICE DESCRIPTOR
        .WORD   0,0               ;DIRECTORY DESCRIPTOR
        .WORD   ONAMSZ,ONAM       ;FILENAME DESCRIPTOR
IFDSPT: .WORD   0,0               ;DEVICE DESCRIPTOR
        .WORD   0,0               ;DIRECTORY DESCRIPTOR
        .WORD   INAMSZ,INAM       ;FILENAME DESCRIPTOR
ONAM:   .ASCII  /OUTPUT.DAT/
        ONAMSZ=.-ONAM
        .EVEN
INAM:   .ASCII  /INPUT.DAT/
        INAMSZ=.-INAM
        .EVEN
START:  FINIT$                    ;INIT FILE STORAGE REGION
        OPEN$R  #FDBIN,#INLUN,#IFDSPT,,#RECBUF,#80.,
                                  ; RUNTIME INITIALIZATION
        BCS     ERROR             ;BRANCH IF ERROR
        FDAT$R  #FDBOUT,#R.VAR,#FD.CR    ;RUNTIME INITIALIZATION
        OPEN$W  R0,#OUTLUN,#OFDSPT,,#RECBUF,#80.,
        BCS     ERROR             ;BRANCH IF ERROR
GTREC:  GET$    #FDBIN            ;NOTE - URBD IS ALL SET UP
        BCS     CKEOF             ;ERROR SHOULD BE EOF INDICATION
        MOV     F.NRBD(R0),R1     ;R1=SIZE OF RECORD READ
        MOV     #RECBUF,R2
        ADD     R1,R2             ;R2=ADDRESS OF LAST BYTE+1
10$:    CMPB    #40,-(R2)         ;STRIP TRAILING BLANKS
        BNE     PTREC
        SOB     R1,10$
;AT THIS POINT, R1 CONTAINS THE STRIPPED SIZE OF THE
;RECORD TO BE WRITTEN.  IF THE CARD IS BLANK,
;A ZERO-LENGTH RECORD IS WRITTEN.
PTREC:  PUT$    #FDBOUT,,R1       ;R1 IS NEEDED TO SPECIFY
        BCC     GTREC             ;THE RECORD SIZE.
ERROR:  NOP                       ;ERROR CODE GOES HERE
CKEOF:  CMPB    #IE.EOF,F.ERR(R0) ;END OF FILE?
        BNE     ERROR             ;BRANCH IF OTHER ERROR
        CLOSE$  R0                ;CLOSE THE INPUT FILE
        BCS     ERROR
        CLOSE$  #FDBOUT           ;CLOSE THE OUTPUT FILE
        BCS     ERROR
        EXIT$S                    ;ISSUE EXIT DIRECTIVE
        .END    START
```

APPENDIX E

INDEX FILE FORMAT

The index file consists of virtual blocks, starting with virtual block 1, i.e., the bootstrap block. Virtual block 2 is the home block. The structure of a FILES-11 index file is shown below.

VIRTUAL BLOCK NUMBER	INDEX FILE ELEMENT
1	Bootstrap block.
2	Home Block.
3	Index file bit map (n blocks); the value of n is in the home block.
3+n	Index file header.
3+n+1	Storage map header.
3+n+2	Bad-block file header.
3+n+3	Master file directory header.
3+n+4	Checkpoint file header (not used by RSX-11M).
3+n+5	User file header 1.
3+n+6	User file header 2.
.	.
.	User file header n.

E.1 BOOTSTRAP BLOCK

A disk that is structured for FILES-11 has a 256-word block, starting at physical block 0. This block contains either a bootstrap routine or a message to the operator stating that the volume does not contain a bootstrappable system. The bootstrap routine brings a core image into memory from a predefined location on the disk. In IAS and RSX-11D, the core image is pointed to by a file header block in the index file.

E.2 HOME BLOCK

The home block contains volume identification information that is formatted as shown in Table E-1. This block is located either in logical block 1 or at any even multiple of 256 blocks.

The offset names in the home block may be defined either locally or globally, as shown below:

```
HMBOF$  DEF$L        ;DEFINES OFFSETS LOCALLY.

HMBOF$  DEF$G        ;DEFINES OFFSETS GLOBALLY.
```

E.3 INDEX FILE BIT MAP

The index file bit map controls the use of file header blocks in the index file. The bit map contains a bit for each file header block contained in the index file. The bit for a file header block is located by means of the file number of the file with which it is associated. The values of the bit map are as follows:

- 0 - Indicates that the file header block is available. The file control primitives can use this block to create a file.
- 1 - Indicates that the file header block is in use. This block has already been used to create a file.

E.4 PREDEFINED FILE HEADER BLOCKS

The first five file header blocks are described below.

FILE HEADER BLOCK	SIGNIFICANCE
Index File Header	This is the standard header associated with the index file.
Storage Map File Header	The storage map is a file that is used to control the assignment of disk blocks to files.
Bad Block File Header	The bad block file is a file that consists of unusable blocks (bad sectors) on the disk.
Master File Directory Header	This header block is associated with the master file directory for the disk. This directory contains entries for the index file, the storage map file, the bad block file, the master file directory (MFD), the checkpoint file, and all user file directories (UFD's).
Checkpoint File Header	This block, which is used only by IAS and RSX-11D, identifies the file that is used for the checkpoint areas for all checkpointable tasks.

The remainder of the index file consists of file header blocks for user files, as shown in the illustration at the beginning of this section.

Table E-1
Home Block Format

SIZE (in bytes)	CONTENT	OFFSET
2	Index bit map size.	H.IBSZ
4	Location of index bit map.	H.IBLB
2	Maximum files allowed.	H.FMAX
2	Storage bit map cluster factor.	H.SBCL
2	Disk device type.	H.DVTY
2	Structure level.	H.VLEV
12.	Volume name (12 ASCII characters).	H.VNAM
4	Reserved.	
2	Volume owner's UIC.	H.VOWN
2	Volume protection code.	H.VPRO
2	Volume characteristics.	H.VCHA
2	Default file protection word.	H.FPRO
2	Volume file sequence number (updated by the DISMOUNT command).	H.FVSQ
2	Volume flags word.	H.FLGS
1	Default number of retrieval pointers in a window.	H.WISZ
1	Default number of blocks to extend files.	H.FIEX
14.	Available space.	--
2	Checksum of words 0-28.	H.CHK1
14.	Creation date and time.	H.VDAT

Table E-1 (Cont.)
Home Block Format

SIZE (in bytes)	CONTENT	OFFSET
100.	Volume header label (not used).	--
82.	System specific information (not used).	--
254.	Relative volume table (not used).	--
2	Checksum of home block (words 0 through 255).	H.CHK2

APPENDIX F

FILE HEADER BLOCK FORMAT

Table F-1 shows the format of the file header block. The various areas within the file header block are described in detail in the following sections. The offset names in the file header block may be defined either locally or globally, as shown in the following statements:

```
FHDOF$  DEF$L           ;DEFINE OFFSETS LOCALLY.

FHDOF$  DEF$G           ;DEFINE OFFSETS GLOBALLY.
```

Table F-1
File Header Block

AREA	SIZE (in bytes)	CONTENT	OFFSET
HEADER AREA	1	Identification area offset in words.	H.IDOF
	1	Map area offset in words.	H.MPOF
	2	File number.	H.FNUM
	2	File sequence number.	H.FSEQ
	2	Structure level and system number.	H.FLEV
	-	Offset to file owner information, consisting of member number and group number.	H.FOWN
	1	Member number.	H.PROG
	1	Group number.	H.PROJ
	2	File protection code.	H.FPRO
	1	User-controlled file characteristics.	H.UCHA
	1	System-controlled file characteristics.	H.SCHA
	32.	User file attributes.	H.UFAT
	-	Size in bytes of header area of file header block.	S.HDHD
IDENTIFICATION AREA	6	Filename (Radix-50).	I.FNAM
	2	File type (Radix-50).	I.FTYP
	2	File version number (binary).	I.FVER
	2	Revision number.	I.RVNO
	7	Revision date.	I.RVDT

Table F-1 (Cont.)
File Header Block

AREA	SIZE (in bytes)	CONTENT	OFFSET
IDENTIFICATION AREA (cont.)	6	Revision time.	I.RVTI
	7	Creation date.	I.CRDT
	6	Creation time.	I.CRTI
	7	Expiration date.	I.EXDT
	1	To round up to word boundary.	
	-	Size (in bytes) of identification area of file header block.	S.IDHD
MAP AREA	1	Extension segment number.	M.ESQN
	1	Extension relative volume number (not implemented).	M.ERVN
	2	Extension file number.	M.EFNU
	2	Extension file sequence number.	M.EFSQ
	1	Size (in bytes) of the block count field of a retrieval pointer (1 or 2); only 1 is used.	M.CTSZ
	1	Size (in bytes) of the logical block number field of a retrieval pointer (2, 3, or 4); only 3 is used.	M.LBSZ
	1	Words of retrieval pointers in use in the map area.	M.USE
	1	Maximum number of words of retrieval pointers available in the map area.	M.MAX
	-	Start of retrieval pointers.	M.RTRV
	-	Size in bytes of map area of file header block.	S.MPHD

Table F-1 (Cont.)
File Header Block

AREA	SIZE (in bytes)	CONTENT	OFFSET
CHECKSUM WORD	2	Checksum of words 0 through 255.	H.CKSM

NOTE

The checksum word is the last word of the file header block. Retrieval pointers occupy the space from the end of the map area to the checksum word.

F.1 HEADER AREA

The information in the header area of the file header block consists of the following:

IDENTIFICATION AREA OFFSET - Word 0, bits 0-7. This byte locates the start of the identification area relative to the start of the file header block. This offset contains the number of words from the start of the header to the identification area.

MAP AREA OFFSET - Word 0, bits 8-15. This byte locates the start of the map area relative to the start of the file header block. This offset contains the number of words from the start of the header area to the map area.

FILE NUMBER - The file number defines the position this file header block occupies in the index file, e.g., the index file is number 1, the storage bit map is file number 2, etc.

FILE SEQUENCE NUMBER - The file number and the file sequence number constitute the file identification number used by the system. This number is different each time a header is re-used.

STRUCTURE LEVEL - This word identifies the system that created the file and indicates the file structure. A value of [1,1] is associated with all current FILES-11 volumes.

FILE OWNER INFORMATION	- This word contains the group number and owner number constituting the user identification code (UIC) for the file. Legal UIC's are within the range [1,1] to [377,377]. UIC [1,1] is reserved for the system.
FILE PROTECTION CODE	- This word specifies the manner in which the file can be used and who can use it. When creating the file, the user specifies the extent of protection desired for the file.
FILE CHARACTERISTICS	- This word, consisting of two bytes, defines the status of the file. Byte 0 defines the user-controlled characteristics, as follows: UC.CON = 200 - Logically contiguous file. UC.DLK = 100 - File improperly closed. Byte 1 defines system-controlled characteristics, as follows: SC.MDL = 200 - File marked for delete. SC.BAD = 100 - Bad data block in file.
USER FILE ATTRIBUTES	- This area consists of 16 words. The first seven words of this area are a direct image of the first seven words of the FDB when the file is opened. The other nine words of the record I/O control area are not used.

F.2 IDENTIFICATION AREA

The information in the identification area of the file header block consists of the following:

FILENAME	- The file's creator specifies a filename of up to nine Radix-50 characters in length. This name is placed in the name field. The unused portion of the field (if any) is zero-filled.
FILE TYPE	- This word contains the file type in Radix-50 format.
FILE VERSION NUMBER	- This word contains the file version number, in binary, as specified by the creator of the file.

REVISION NUMBER	- This word is initialized to zero when the file is created; it is incremented each time a file is closed after being updated or modified.
REVISION DATE	- Seven bytes are used to maintain the date on which the file was last revised. The revision date is kept in ASCII form in the format day, month, year (2 bytes, 3 bytes, and 2 bytes, respectively). This date is meaningful only if the revision number is a non-zero value.
REVISION TIME	- Six bytes are used to record the time at which the file was last revised. This information is recorded in ASCII form in the format hour, minute, and second (2 bytes each).
CREATION DATE	- The date on which the file was created is kept in a 7-byte field having the same format as the revision date (see above).
CREATION TIME	- The time of the file's creation is maintained in a 6-byte field having the same format as the revision time (see above).
EXPIRATION DATE	- The date on which the file becomes eligible to be deleted is kept in a 7-byte field having the same format as the revision date (see above). Use of expiration is not implemented.

F.3 MAP AREA

The map area contains the information necessary to map virtual block numbers to logical block numbers. This is done by means of pointers, each of which points to an area of contiguous blocks. A pointer consists of a count field and a number field. The count field defines the number of blocks contained in the contiguous area pointed to, and the logical block number (LBN) field defines the block number of the first logical block in the area.

A value of "n" in the count field (see below) means that n+1 blocks are allocated, starting at the specified block number.

The retrieval pointer format used in the FILES-11 file structure is shown below:

15	0
COUNT-1	HIGH LBN

31	16
LOW	LBN

NOTE

The remaining paragraphs in this appendix do not apply to RSX-11M.

The map area normally has space for 102 retrieval pointers. It can map up to 102 discontiguous segments or up to 26112 blocks if the file is contiguous. If more retrieval pointers are required because the file is too large or consists of too many discontiguous segments, extension headers are allocated to hold additional retrieval pointers. Extension headers are allocated within the index file. They are identified by a file number and a file sequence as are other file headers; however, extension file headers do not appear in any directory.

A nonzero value in the extension file number field of the map area indicates that an extension header exists. The extension header is identified by the extension file number and the extension file sequence number. The extension segment number is used to number the headers of the file sequentially starting with a zero for the first.

Extension headers of a file contain a header area and identification area that are a copy of the first header as it appeared when the first extension was created. Extension headers are not updated when the first header of the file is modified.

Extension headers are created and handled by the file control primitives as needed; their use is transparent to the user.

APPENDIX G

SUPPORT OF ANSI MAGNETIC TAPE STANDARD

This appendix defines the IAS and RSX-11D magnetic tape labeling standard, which is a level three implementation of the June 19, 1974 Proposed Revision to the ANSI standard Magnetic Tape Labels and File Structure for Information Interchange (X3.27-1969). The only exception is that IAS and RSX-11D do not support spanned records.

G.1 VOLUME AND FILE LABELS

Tables G-1, G-2, and G-3 present the format of volume labels and file header labels.

G.1.1 Volume Label Format

Table G-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.
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 G.1.1.1 below.
52-79	Reserved	28	Spaces
80	Label standard version	1	1

G.1.1.1 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 [8,255])
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 (indicates space)

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].

G.1.2 User Volume Labels

User volume labels never are written or passed back to the user. If present, they are skipped.

G.1.3 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 G.1.3.1.

Tables G-2 and G-3 describe the HDR1 and HDR2 labels respectively.

Table G-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 (indicates space) or 00000 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 G.1.1.1. Examples: DECFILE11A DECSYSTEM10 Pad name with spaces.
74	Reserved	7	Spaces

Table G-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 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

G.1.3.1 File Identifier Processing by Files-11 - The following steps describe the processing of a file identifier by Files-11.

1. The first nine characters at a maximum are processed by an ASCII to Radix-50 converter. The filename results 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. $\text{generation number} = \frac{\text{version \# - 1 + 1}}{100}$

   b. $\text{generation version \#} = \frac{\text{version \# - 1}}{\text{Modulo 100}}$

NOTE

In both calculations, remainders are ignored.

The following are examples.

FILES-11 VERSION #	GENERATION #	GENERATION VER #
1	1	0
50	1	49
100	1	99
101	2	0
1010	11	9

G.1.4 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 instead of HDR1,

2. The block count field contains the number of records in the last file section on the volume.

G.1.5 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.

G.1.6 User File Labels

User file labels never are written or passed back to the user. If present, they are skipped.

G.2 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 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.

G.2.1 Single File Single Volume

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**

G.2.2 Single File Multi-Volume

BOT,VOL1,HDR1,HDR2*---DATA---*EOV1,EOV2**

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**

G.2.3 Multi-File Single Volume

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2*HDR1,HDR2---DATA--*EOF1,EOF2**

G.2.4 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**

G.3 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.

G.4 ANSI MAGNETIC TAPE FILE HEADER BLOCK (FCS COMPATIBLE)

Figure G-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

H.MPOF	MAP OFFSET	IDENT OFFSET	H.IDOF
HEADER AREA	FILE SEQUENCE NUMBER		H.FNUM
	FILE SECTION NUMBER		H.FSEQ
	STRUCTURE LEVEL = 4Ø1(8)		H.FLEV
	UIC (FOR VOLUME)		H.FOWN=H.PROG
	PROTECTION CODE (FOR VOLUME)		H.FPRO
	RECORD ATTRIBUTES	RECORD TYPE CODE	H.UFAT
	RECORD SIZE IN BYTES		
	N WORDS OF ZERO'S		
IDENTIFICATION AREA	FILE NAME RAD5Ø		X+I.FNAM (IDENT OFFSET *2)=X
	FILE TYPE RAD5Ø		I.FTYP
	FILE VERSION NUMBER		X+I.FVER
	ZERO'S (REVISION DATE & TIME)		X+I.RVNO
	CREATION DATE & TIME (ØØØØØØ)		X+I.CRDT
	EXPIRATION DATE		X+I.EXDT
	PAD BYTE OF Ø		X+47.
	COPY OF THE HDR1 LABEL		X+5Ø.
	COPY OF THE HDR2 LABEL (if byte 1 of label = Ø, label is not present)		X+13Ø.
MAP AREA	NULL MAP, I.E., ZERO'S (1Ø BYTES LONG)		X+21Ø.= (MAP OF OFFSET 2)

Figure G-1
ANSI Magnetic Tape File Header Block
(FCS Compatible)

APPENDIX H

STATISTICS BLOCK

The format of the statistics block is shown in Figure H-1 below. The statistics block is allocated manually in the user program as described in Item 3.d of section 3.1.2.

Word 0	HIGH LOGICAL BLOCK NUMBER (0 if file is noncontiguous)
Word 1	LOW LOGICAL BLOCK NUMBER (0 if file is noncontiguous)
Word 2	SIZE (high)
Word 3	SIZE (low)
Word 4	LOCK COUNT ACCESS COUNT

Figure H-1
Statistics Block Format

APPENDIX I

ERROR CODES

This appendix lists the Directive Status Word error codes and the I/O error codes returned by the system.

```
                                     .TITLE  QIOMAC - QIOSYM MACRO DEFINITION
                               ; ALTERED SUNDAY 24-NOV-74 13:00
                               ; ALTERED TUESDAY 28-JAN-75 13:50:00
                               ; ALTERED THURSDAY 06-FEB-75 15:50
                               ; ALTERED MONDAY 24-FEB-75 15:40:00 BY ED MARISON
                               ; ALTERED TUE 25-MAR-75 15:30 EDIT # +001
                               ;
                               ; ***** ALWAYS UPDATE THE FOLLOWING TWO LINES TOGETHER
                                     .IDENT  /0304/
         000304                      QI.VER=0304
                               ;
                               ; COPYRIGHT 1974,1975, DIGITAL EQUIPMENT CORP., MAYNARD MASS.

                               ; THIS SOFTWARE IS FURNISHED TO PURCHASER UNDER A LICENSE FOR USE
                               ; ON A SINGLE COMPUTER SYSTEM AND CAN BE COPIED (WITH INCLUSION
                               ; OF DEC'S COPYRIGHT NOTICE) ONLY FOR USE IN SUCH SYSTEM, EXCEPT
                               ; AS MAY OTHERWISE BE PROVIDED IN WRITING BY DEC.

                               ; THE INFORMATION IN THIS DOCUMENT IS SUBJECT TO CHANGE WITHOUT
                               ; NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL
                               ; EQUIPMENT CORPORATION.

                               ; DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY
                               ; OF ITS SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
                               ;
                               ; PETER H. LIPMAN 1-OCT-73
                               ;
                               ;+
                               ; MACRO TO DEFINE STANDARD QUEUE I/O DIRECTIVE FUNCTION VALUES
                               ; AND IOSB RETURN VALUES.  TO INVOKE AT ASSEMBLY TIME (WITH LOCAL
                               ; DEFINITION) USE:
                               ;
                               ;     QIOSYS          ;DEFINE SYMBOLS
                               ;
                               ; TO OBTAIN GLOBAL DEFINITION OF THESE SYMBOLS USE:
                               ;
                               ;     QIOSYS DEF$G    ;SYMBOLS DEFINED GLOBALLY
                               ;
                               ; THE MACRO CAN BE CALLED ONCE ONLY AND THEN
                               ; REDEFINES ITSELF AS NULL.
                               ;-

                                     .MACRO  QIOSYS  $$$GBL,$$$MSG
                                     .IIF    IDN,<$$$GBL>,<DEF$G>,   .GLOBL  QI.VER
                                     .IF     IDN,<$$$MSG>,<DEF$S>
                                     $$$MAX=0
                                     $$MSG=1
```

```
                        .IFF
                        $$MSG=0
                        .ENDC
                        .MCALL  IOERR$
                        IOERR$  $$$GBL          ;I/O ERROR CODES FROM HANDLERS, FCP, FCS
                        .MCALL  DRERR$
                        DRERR$  $$$GBL          ;DIRECTIVE STATUS WORD ERROR CODES
                        .IF     DIF,<$$$MSG>,<DEF$S>
                        .MCALL  FILIO$
                        FILIO$  $$$GBL          ;DEFINE GENERAL QI/O FUNCTION CODES

QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710  25-MAR-75 14:23  PAGE  1-1

                        .MCALL  SPCIO$
                        SPCIO$  $$$GBL          ;DEVICE DEPENDENT I/O FUNCTION CODES
                        .MACRO  QIOSY$  ARG,ARG1,ARG2   ;RECLAIM MACRO STORAGE
                        .ENDM   QIOSY$
                        .ENDC
                        .ENDM   QIOSY$

QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710  25-MAR-75 14:23  PAGE  2

                ;
                ; DEFINE THE ERROR CODES RETURNED BY DEVICE HANDLER AND FILE PRIMITIVES
                ; IN THE FIRST WORD OF THE I/O STATUS BLOCK
                ; THESE CODES ARE ALSO RETURNED BY FILE CONTROL SERVICES (FCS) IN THE
                ; BYTE F.ERR IN THE FILE DESCRIPTOR BLOCK (FDB)
                ;       THE BYTE F.ERR+1 IS 0 IF F.ERR CONTAINS A HANDLER OR FCP ERROR CODE.
                ;
                        .MACRO  IOERR$  $$$GBL
                        .MCALL  .IOER.,DEFIN$
                        .IF     IDN,<$$$GBL>,<DEF$G>
                        ...GBL=1
                        .IFF
                        ...GBL=0
                        .ENDC
                        .IIF    NDF,$$MSG,$$MSG=0
                ;
                ; SYSTEM STANDARD CODES, USED BY ALL FUNCTIONS
                ;
                        .IOER.  IE.BAD,-01.,<BAD PARAMETERS>
                        .IOER.  IE.IFC,-02.,<INVALID FUNCTION CODE>
                        .IOER.  IE.DNR,-03.,<DEVICE NOT READY>
```

```
	.IOER.	IE.VER,-04.,<PARITY ERROR ON DEVICE>
	.IOER.	IE.ONP,-05.,<HARDWARE OPTION NOT PRESENT>
	.IOER.	IE.SPC,-06.,<ILLEGAL USER BUFFER>
	.IOER.	IE.DNA,-07.,<DEVICE NOT ATTACHED>
	.IOER.	IE.DAA,-08.,<DEVICE ALREADY ATTACHED>
	.IOER.	IE.DUN,-09.,<DEVICE NOT ATTACHABLE>
	.IOER.	IE.EOF,-10.,<END OF FILE DETECTED>
	.IOER.	IE.EOV,-11.,<END OF VOLUME DETECTED>
	.IOER.	IE.WLK,-12.,<WRITE ATTEMPTED TO LOCKED UNIT>
	.IOER.	IE.DAO,-13.,<DATA OVERRUN>
	.IOER.	IE.SRE,-14.,<SEND/RECEIVE FAILURE>
	.IOER.	IE.ABO,-15.,<REQUEST TERMINATED>
	.IOER.	IE.PRI,-16.,<PRIVILEGE VIOLATION>
	.IOER.	IE.RSU,-17.,<SHARABLE RESOURCE IN USE>
	.IOER.	IE.OVR,-18.,<ILLEGAL OVERLAY REQUEST>
	.IOER.	IE.BYT,-19.,<ODD BYTE COUNT (OR VIRTUAL ADDRESS)>
	.IOER.	IE.BLK,-20.,<LOGICAL BLOCK NUMBER TOO LARGE>
	.IOER.	IE.MOD,-21.,<INVALID UDC MODULE #>
	.IOER.	IE.CON,-22.,<UDC CONNECT ERROR>
	.IOER.	IE.BBE,-56.,<BAD BLOCK ON DEVICE>
	.IOER.	IE.STK,-58.,<NOT ENOUGH STACK SPACE (FCS OR FCP)>
	.IOER.	IE.FHE,-59.,<FATAL HARDWARE ERROR ON DEVICE>
	.IOER.	IE.EOT,-62.,<END OF TAPE DETECTED>
	.IOER.	IE.OFL,-65.,<DEVICE OFF LINE>
	.IOER.	IE.BCC,-66.,<BLOCK CHECK OR CRC ERROR>


;
; FILE PRIMITIVE CODES
;

	.IOER.	IE.NOD,-23.,<CALLER'S NODES EXHAUSTED>
	.IOER.	IE.DFU,-24.,<DEVICE FULL>
	.IOER.	IE.IFU,-25.,<INDEX FILE FULL>
	.IOER.	IE.NSF,-26.,<NO SUCH FILE>
```

```
        .IOER.  IE.LCK,-27.,<LOCKED FROM WRITE ACCESS>
        .IOER.  IE.HFU,-28.,<FILE HEADER FULL>
        .IOER.  IE.WAC,-29.,<ACCESSED FOR WRITE>
        .IOER.  IE.CKS,-30.,<FILE HEADER CHECKSUM FAILURE>
        .IOER.  IE.WAT,-31.,<ATTRIBUTE CONTROL LIST FORMAT ERROR>
        .IOER.  IE.RER,-32.,<FILE PROCESSOR DEVICE READ ERROR>
        .IOER.  IE.WER,-33.,<FILE PROCESSOR DEVICE WRITE ERROR>
        .IOER.  IE.ALN,-34.,<FILE ALREADY ACCESSED ON LUN>
        .IOER.  IE.SNC,-35.,<FILE ID, FILE NUMBER CHECK>
        .IOER.  IE.SQC,-36.,<FILE ID, SEQUENCE NUMBER CHECK>
        .IOER.  IE.NLN,-37.,<NO FILE ACCESSED ON LUN>
        .IOER.  IE.CLO,-38.,<FILE WAS NOT PROPERLY CLOSED>
        .IOER.  IE.DUP,-57.,<ENTER - DUPLICATE ENTRY IN DIRECTORY>
        .IOER.  IE.BVR,-63.,<BAD VERSION NUMBER>
        .IOER.  IE.BHD,-64.,<BAD FILE HEADER>
        .IOER.  IE.EXP,-75.,<FILE EXPIRATION DATE NOT REACHED>
        .IOER.  IE.BTF,-76.,<BAD TAPE FORMAT>

;
; FILE CONTROL SERVICES CODES
;

        .IOER.  IE.NBF,-39.,<OPEN - NO BUFFER SPACE AVAILABLE FOR FILE>
        .IOER.  IE.RBG,-40.,<ILLEGAL RECORD SIZE>
        .IOER.  IE.NBK,-41.,<FILE EXCEEDS SPACE ALLOCATED, NO BLOCKS>
        .IOER.  IE.ILL,-42.,<ILLEGAL OPERATION ON FILE DESCRIPTOR BLOCK>
        .IOER.  IE.BTP,-43.,<BAD RECORD TYPE>
        .IOER.  IE.RAC,-44.,<ILLEGAL RECORD ACCESS BITS SET>
        .IOER.  IE.RAT,-45.,<ILLEGAL RECORD ATTRIBUTES BITS SET>
        .IOER.  IE.RCN,-46.,<ILLEGAL RECORD NUMBER - TOO LARGE>
        .IOER.  IE.MBK,-47.,<MULTIPLE BLOCK READ/WRITE - NOT IMPLEMENTED YET>
        .IOER.  IE.2DV,-48.,<RENAME - 2 DIFFERENT DEVICES>
        .IOER.  IE.FEX,-49.,<RENAME - NEW FILE NAME ALREADY IN USE>
        .IOER.  IE.BDR,-50.,<BAD DIRECTORY FILE>
        .IOER.  IE.RNM,-51.,<CAN'T RENAME OLD FILE SYSTEM>
        .IOER.  IE.BDI,-52.,<BAD DIRECTORY SYNTAX>
        .IOER.  IE.FOP,-53.,<FILE ALREADY OPEN>
        .IOER.  IE.BNM,-54.,<BAD FILE NAME>
        .IOER.  IE.BDV,-55.,<BAD DEVICE NAME>
        .IOER.  IE.NFI,-60.,<FILE ID WAS NOT SPECIFIED>
        .IOER.  IE.ISQ,-61.,<ILLEGAL SEQUENTIAL OPERATION>
        .IOER.  IE.NNC,-77.,<NOT ANSI 'D' FORMAT BYTE COUNT>
;
; NETWORK ACP CODES
;
        .IOER.  IE.AST,-67.,<NO AST SPECIFIED IN CONNECT>
```

```
                        .IOER.  IE.NNN,-68,,<NO SUCH NODE>
                        .IOER.  IE.NFW,-69,,<PATH LOST TO PARTNER> ;+001 THIS CODE MUST BE ODD
                        .IOER.  IE.BLB,-70,,<BAD LOGICAL BUFFER> ;+001
                        .IOER.  IE.TMM,-71,,<TOO MANY OUTSTANDING MESSAGES>
                        .IOER.  IE.NDR,-72,,<NO DYNAMIC SPACE AVAILABLE>
                        .IOER.  IE.CNR,-73,,<CONNECTION REJECTED>
                        .IOER.  IE.TMO,-74,,<TIMEOUT ON REQUEST>
                        .IOER.  IE.NNL,-78,,<NOT A NETWORK LUN> ;+001

                ;
                ; SUCCESSFUL RETURN CODES---
```

```
QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710  25-MAR-75 14:23  PAGE  2-2

                ;

                        DEFINS  IS.PND,+00,      ;OPERATION PENDING
                        DEFINS  IS.SUC,+01,      ;OPERATION COMPLETE, SUCCESS
                        DEFINS  IS.BV,+05,       ;ON A/D READ, AT LEAST ONE BAD VALUE
                                                 ;WAS READ (REMAINDER MAY BE GOOD),
                                                 ;BAD CHANNEL IS INDICATED BY A
                                                 ;NEGATIVE VALUE IN THE BUFFER,


                ;
                ; TTY SUCCESS CODES:
                ;

                        DEFINS  IS.CR,<15*400+1> ;CARRIAGE RETURN WAS TERMINATOR
                        DEFINS  IS.ESC,<33*400+1> ;ESCAPE (ALTMODE) WAS TERMINATOR


                ; ******
                ;
                ; THE NEXT AVAILABLE ERROR NUMBER IS: -79,
                ; ALL LOWER NUMBERS ARE IN USE!!
                ;
                ; *****
                        .IF     EQ,$$MSG
                        .MACRO  IOERR$  A
                        .ENDM   IOERR$
                        .ENDC
                        .ENDM   IOERR$
```

```
;
; DEFINE THE DIRECTIVE ERROR CODES RETURNED IN THE DIRECTIVE STATUS WORD
;
;       FILE CONTROL SERVICES (FCS) RETURNS THESE CODES IN THE BYTE F.ERR
; OF THE FILE DESCRIPTOR BLOCK (FDB).  TO DISTINGUISH THEM FROM THE
; OVERLAPPING CODES FROM HANDLER AND FILE PRIMITIVES, THE BYTE
; F.ERR+1 IN THE FDB WILL BE NEGATIVE FOR A DIRECTIVE ERROR CODE.
;
        .MACRO  DRERR$  $$$GBL
        .MCALL  .QIOE.,DEFIN$
        .IF     IDN,<$$$GBL>,<DEF$G>
        ...GBL=1
        .IFF
        ...GBL=0
        .ENDC
        .IIF    NDF,$$MSG,$$MSG=0
;
; STANDARD ERROR CODES RETURNED BY DIRECTIVES IN THE DIRECTIVE STATUS WORD
;
        .QIOE.  IE.UPN,-01.,<INSUFFICIENT DYNAMIC STORAGE>
        .QIOE.  IE.INS,-02.,<SPECIFIED TASK NOT INSTALLED>
        .QIOE.  IE.ULN,-05.,<UN-ASSIGNED LUN>
        .QIOE.  IE.HWR,-06.,<HANDLER TASK NOT RESIDENT>
        .QIOE.  IE.ACT,-07.,<TASK NOT ACTIVE>
        .QIOE.  IE.ITS,-08.,<DIRECTIVE INCONSISTENT WITH TASK STATE>
        .QIOE.  IE.CKP,-10.,<ISSUING TASK NOT CHECKPOINTABLE>
;
;
;
        .QIOE.  IE.AST,-80.,<DIRECTIVE ISSUED/NOT ISSUED FROM AST>
        .QIOE.  IE.LNL,-90.,<LUN LOCKED IN USE>
        .QIOE.  IE.IDU,-92.,<INVALID DEVICE OR UNIT>
        .QIOE.  IE.ITI,-93.,<INVALID TIME PARAMETERS>
        .QIOE.  IE.IPR,-95.,<INVALID PRIORITY ( .GT. 250.)>
        .QIOE.  IE.ILU,-96.,<INVALID LUN>
        .QIOE.  IE.IEF,-97.,<INVALID EVENT ( .GT. 64.)>
        .QIOE.  IE.ADP,-98.,<PART OF DPB OUT OF USER'S SPACE>
        .QIOE.  IE.SDP,-99.,<DIC OR DPB SIZE INVALID>
;
; SUCCESS CODES FROM DIRECTIVES - PLACED IN THE DIRECTIVE STATUS WORD
;
        DEFIN$  IS.CLR,0        ;EVENT FLAG WAS CLEAR
                                ;FROM CLEAR EVENT FLAG DIRECTIVE
        DEFIN$  IS.SET,2        ;EVENT FLAG WAS SET
                                ;FROM SET EVENT FLAG DIRECTIVE
        DEFIN$  IS.SPD,2        ;TASK WAS SUSPENDED
;
```

```
                              ;
                                      .IF     EQ,$$MSG
                                      .MACRO  DRERR$  A
                                      .ENDM   DRERR$
                                      .ENDC
                                      .ENDM   DRERR$
```

QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710 25-MAR-75 14:23 PAGE 4

```
                              ;
                              ; DEFINE THE GENERAL QI/O FUNCTION CODES - DEVICE INDEPENDENT
                              ;
                                      .MACRO  FILIO$  $$$GBL
                                      .MCALL  .WORD.,DEFIN$
                                      .IF     IDN,<$$$GBL>,<DEF$G>
                                      ...GBL=1
                                      .IFF
                                      ...GBL=0
                                      .ENDC
                              ;
                              ; GENERAL QI/O QUALIFIER BYTE DEFINITIONS
                              ;
                                      .WORD.  IQ.X,001,000    ;NO ERROR RECOVERY
                                      .WORD.  IQ.Q,002,000    ;QUEUE REQUEST IN EXPRESS QUEUE
                              ;;;     .WORD.  IQ.,004,000     ;RESERVED
                              ;
                              ; EXPRESS QUEUE COMMANDS
                              ;

                                      .WORD.  IO.KIL,012,000  ;KILL CURRENT REQUEST
                                      .WORD.  IO.RDN,022,000  ;I/O RUNDOWN
                                      .WORD.  IO.UNL,042,000  ;UNLOAD I/O HANDLER TASK
                                      .WORD.  IO.LTK,050,000  ;LOAD A TASK IMAGE FILE
                                      .WORD.  IO.RTK,060,000  ;RECORD A TASK IMAGE FILE
                              ;
                              ; GENERAL DEVICE HANDLER CODES
                              ;
                                      .WORD.  IO.WLB,000,001  ;WRITE LOGICAL BLOCK
                                      .WORD.  IO.RLB,000,002  ;READ LOGICAL BLOCK
                                      .WORD.  IO.LOV,010,002  ;LOAD OVERLAY (DISK DRIVER)
                                      .WORD.  IO.ATT,000,003  ;ATTACH A DEVICE TO A TASK
                                      .WORD.  IO.DET,000,004  ;DETACH A DEVICE FROM A TASK
                              ;
                              ; DIRECTORY PRIMITIVE CODES
                              ;
                                      .WORD.  IO.FNA,000,011  ;FIND FILE NAME IN DIRECTORY
```

```
                .WORD.  IO.RNA,000,013  ;REMOVE FILE NAME FROM DIRECTORY
                .WORD.  IO.ENA,000,014  ;ENTER FILE NAME IN DIRECTORY
        ;
        ; FILE PRIMITIVE CODES
        ;
                .WORD.  IO.CLN,000,007  ;CLOSE OUT LUN
                .WORD.  IO.ACR,000,015  ;ACCESS FOR READ
                .WORD.  IO.ACW,000,016  ;ACCESS FOR WRITE
                .WORD.  IO.ACE,000,017  ;ACCESS FOR EXTEND
                .WORD.  IO.DAC,000,020  ;DE-ACCESS FILE
                .WORD.  IO.RVB,000,021  ;READ VIRITUAL BLOCK
                .WORD.  IO.WVB,000,022  ;WRITE VIRITUAL BLOCK
                .WORD.  IO.EXT,000,023  ;EXTEND FILE
                .WORD.  IO.CRE,000,024  ;CREATE FILE
                .WORD.  IO.DEL,000,025  ;DELETE FILE
                .WORD.  IO.RAT,000,026  ;READ FILE ATTRIBUTES
                .WORD.  IO.WAT,000,027  ;WRITE FILE ATTRIBUTES
                .WORD.  IO.APV,010,030  ;PRIVILEGED ACP CONTROL
                .WORD.  IO.APC,000,030  ;ACP CONTROL
        ;
```

QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710 25-MAR-75 14:23 PAGE 4-1

```
        ;
                .MACRO  FILIOS  A
                .ENDM   FILIOS
                .ENDM   FILIOS
```

QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710 25-MAR-75 14:23 PAGE 5

```
        ;
        ; DEFINE THE QI/O FUNCTION CODES THAT ARE SPECIFIC TO INDIVIDUAL DEVICES
        ;
                .MACRO  SPCIOS  $$$GBL
                .MCALL  .WORD.,DEFIN$
                .IF     IDN,<$$$GBL>,<DEF$G>
                ...GBL=1
                .IFF
                ...GBL=0
                .ENDC
        ;
        ; QI/O FUNCTION CODES FOR SPECIFIC DEVICE DEPENDENT FUNCTIONS
```

```
                        ;
        .WORD.  IO.WLV,100,001  ;WRITE LOGICAL REVERSE (DECTAPE)
        .WORD.  IO.WLS,010,001  ;(COMMUNICATIONS) WRITE PRECEDED BY SYNC TRAIN
        .WORD.  IO.WNS,020,001  ;(COMMUNICATIONS) WRITE, NO SYNC TRAIN
        .WORD.  IO.RLV,100,002  ;READ REVERSE (DECTAPE)
        .WORD.  IO.RNC,040,002  ;READ - NO LOWER CASE CONVERT (TTY)
        .WORD.  IO.RAL,010,002  ;READ PASSING ALL CHARACTERS (TTY)
        .WORD.  IO.RNE,020,002  ;READ WITHOUT ECHO (TTY)
        .WORD.  IO.RDB,200,002  ;READ BINARY MODE (CARD READER)
        .WORD.  IO.RHD,010,002  ;(COMMUNICATIONS) READ, STRIP SYNC
        .WORD.  IO.RNS,020,002  ;(COMMUNICATIONS) READ, DON'T STRIP SYNC
        .WORD.  IO.CRC,040,002  ;(COMMUNICATIONS) READ, DON'T CLEAR CRC
        .WORD.  IO.R1C,000,005  ;READ SINGLE CHANNEL (AFC, AD01, UDC)
        .WORD.  IO.INL,000,005  ;(COMMUNICATIONS) INITIALIZATION FUNCTION
        .WORD.  IO.TRM,010,005  ;(COMMUNICATIONS) TERMINATION FUNCTION
        .WORD.  IO.RBC,000,006  ;READ MULTICHANNELS (BUFFER DEFINES CHANNELS)
        .WORD.  IO.MOD,000,006  ;(COMMUNICATIONS) SETMODE FUNCTION FAMILY
        .WORD.  IO.HDX,010,006  ;(COMMUNICATIONS) SET UNIT HALF DUPLEX
        .WORD.  IO.FDX,020,006  ;(COMMUNICATIONS) SET UNIT FULL DUPLEX
        .WORD.  IO.SYN,040,006  ;(COMMUNICATIONS) SPECIFY SYNC CHARACTER
        .WORD.  IO.RTC,000,007  ;READ CHANNEL - TIME BASED
        .WORD.  IO.RWD,000,005  ;REWIND (MAGTAPE, DECTAPE)
        .WORD.  IO.SPB,020,005  ;MAGTAPE, SPACE "N" BLOCKS
        .WORD.  IO.SPF,040,005  ;MAGTAPE, SPACE "N" EOF MARKS
        .WORD.  IO.EOF,000,006  ;MAGTAPE, WRITE EOF
        .WORD.  IO.STC,100,005  ;MAGTAPE, SET CHARACTERISTIC
        .WORD.  IO.SEC,120,005  ;MAGTAPE, SENSE CHARACTERISTIC
        .WORD.  IO.RWU,140,005  ;REWIND AND UNLOAD (MAGTAPE, DECTAPE)
        .WORD.  IO.SMO,160,005  ;MAGTAPE, MOUNT & SET CHARACTERISTICS
        .WORD.  IO.SAO,000,010  ;UDC SINGLE CHANNEL ANALOG OUTPUT
        .WORD.  IO.SSO,000,011  ;UDC SINGLE SHOT, SINGLE POINT
        .WORD.  IO.MSO,000,012  ;UDC SINGLE SHOT, MULTI-POINT
        .WORD.  IO.SLO,000,013  ;UDC LATCHING, SINGLE POINT
        .WORD.  IO.MLO,000,014  ;UDC LATCHING, MULTI-POINT
        .WORD.  IO.LED,000,024  ;LPS11 WRITE LED DISPLAY LIGHTS
        .WORD.  IO.SDO,000,025  ;LPS11 WRITE DIGITAL OUTPUT REGISTER
        .WORD.  IO.SDI,000,026  ;LPS11 READ DIGITAL INPUT REGISTER
        .WORD.  IO.SCS,000,026  ;UDC CONTACT SENSE, SINGLE POINT
        .WORD.  IO.REL,000,027  ;LPS11 WRITE RELAY
        .WORD.  IO.MCS,000,027  ;UDC CONTACT SENSE, MULTI-POINT
        .WORD.  IO.ADS,000,030  ;LPS11 SYNCHRONOUS A/D SAMPLING
        .WORD.  IO.CCI,000,030  ;UDC CONTACT INT - CONNECT
        .WORD.  IO.MDI,000,031  ;LPS11 SYNCHRONOUS DIGITAL INPUT
        .WORD.  IO.DCI,000,031  ;UDC CONTACT INT - DISCONNECT
        .WORD.  IO.XMT,000,031  ;(COMMUNICATIONS) TRANSMIT SPECIFIED BLOCK WITH ACK
```

```
                        .WORD.  IO.XNA,010,031  ;(COMMUNICATIONS) TRANSMIT WITHOUT ACK
                        .WORD.  IO.HIS,000,032  ;LPS11 SYNCHRONOUS HISTOGRAM SAMPLING
                        .WORD.  IO.RCI,000,032  ;UDC CONTACT INT - READ
                        .WORD.  IO.RCV,000,032  ;(COMMUNICATIONS) RECEIVE DATA IN BUFFER SPECIFIED
                        .WORD.  IO.MDO,000,033  ;LPS11 SYNCHRONOUS DIGITAL OUTPUT
                        .WORD.  IO.CTI,000,033  ;UDC TIMER - CONNECT
                        .WORD.  IO.CON,000,033  ;(COMMUNICATIONS) COMMUNICATIONS CONNECT FUNCTION
                        .WORD.  IO.CPR,010,033  ;(COMMUNICATIONS) COMMUNICATIONS CONNECT NO TIMEOUTS
                        .WORD.  IO.CAS,020,033  ;(COMMUNICATIONS) COMMUNICATIONS CONNECT WITH AST
                        .WORD.  IO.CRJ,040,033  ;(COMMUNICATIONS) COMMUNICATIONS CONNECT REJECT
                        .WORD.  IO.CBO,110,033  ;+001 (COMMUNICATIONS) COMMUNICATIONS BOOT CONNECT
                        .WORD.  IO.CTR,210,033  ;+001 (COMMUNICATIONS) COMMUNICATIONS TRANSPARENT CONNECT
                        .WORD.  IO.GNI,010,035  ;(COMMUNICATIONS) COMMUNICATIONS GET NODE INFO
                        .WORD.  IO.GLI,020,035  ;(COMMUNICATIONS) COMMUNICATIONS GET LINK INFO
                        .WORD.  IO.GLC,030,035  ;(COMMUNICATIONS) GET LINK INFO CLEAR COUNTERS
                        .WORD.  IO.GRI,040,035  ;(COMMUNICATIONS) GET REMOTE NODE INFO
                        .WORD.  IO.GRC,050,035  ;+001 (COMMUNICATIONS) GET REMOTE NODE ERROR COUNTS
                        .WORD.  IO.GRN,060,035  ;+001 (COMMUN.) GET REMOTE NODE NAME
                        .WORD.  IO.CSM,070,035  ;+001 (COMMUNICATIONS) CHANGE SOLO MODE
                        .WORD.  IO.CIN,100,035  ;+001 (COMMUN.) CHANGE CONNECTION INHIBIT
                        .WORD.  IO.CBN,110,035  ;+001 (COMMUNICATIONS) CIRCULAR BUFFER NCS
                        .WORD.  IO.CBD,120,035  ;+001 (COMMUNICATIONS) CIRCULAR BUFFER DDCMP
                        .WORD.  IO.DTI,000,034  ;UDC TIMER - DISCONNECT
                        .WORD.  IO.DIS,000,034  ;(COMMUNICATIONS) COMMUNICATIONS DISCONNECT FUNCTION
                        .WORD.  IO.MDA,000,034  ;LPS11 SYNCHRONOUS D/A OUTPUT
                        .WORD.  IO.RTI,000,035  ;UDC TIMER - READ
                        .WORD.  IO.CTL,000,035  ;(COMMUNICATIONS) NETWORK CONTROL FUNCTION
                        .WORD.  IO.STP,000,035  ;LPS11 STOP IN PROGRESS FUNCTION
                        .WORD.  IO.ITI,000,036  ;UDC TIMER - INITIALIZE
                        .WORD.  IO.WPB,040,001  ; RX01 - FLOPPY DISK WRITE PHYSICAL BLOCK
                        .WORD.  IO.RPB,040,002  ; RX01 - FLOPPY DISK READ PHYSICAL BLOCK
                        .WORD.  IO.SHT,010,005  ;SET HORIZONTAL TAB POSITIONS
                        .WORD.  IO.SST,030,005  ;SET SPECIAL TERMINATOR CHARACTERS
                        .WORD.  IO.SEM,040,005  ;SET TERMINAL MODE (CHARACTERISTICS)
                        .WORD.  IO.SNM,050,005  ;SENSE TERMINAL MODE
                        .WORD.  IO.CCT,060,005  ;CONNECT TO REMOTE TERMINAL (AUTO DIALOUT)
                        .WORD.  IO.DCT,070,005  ;DISCONNECT FROM REMOTE TERMINAL (HANGUP)
                        .WORD.  IO.ESA,100,005  ;ENABLE STATUS AST
                    ;
                    ;
                        .MACRO  SPCIOS  A
                        .ENDM   SPCIOS
                        .ENDM   SPCIOS
```

```
QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710  25-MAR-75 14:23  PAGE  6

                                ;
                                ; HANDLER ERROR CODES RETURNED IN I/O STATUS BLOCK ARE DEFINED THROUGH THIS
                                ; MACRO WHICH THEN CONDITIONALLY INVOKES THE MESSAGE GENERATING MACRO
                                ; FOR THE QIOSYM.MSG FILE
                                ;
                                        .MACRO  .IOER.  SYM,LO,MSG
                                        DEFINS  SYM,LO
                                        .IF     GT,$$MSG
                                        .MCALL  .IOMG.
                                        .IOMG.  SYM,LO,<MSG>
                                        .ENDC
                                        .ENDM   .IOER.
                                ;
                                ; QI/O ERROR CODES ARE DEFINED THOUGH THIS MACRO WHICH THEN INVOKES THE
                                ; ERROR MESSAGE GENERATING MACRO, ERROR CODES -129 THROUGH -256
                                ; ARE USED IN THE QIOSYM.MSG FILE
                                ;
                                        .MACRO  .QIOE.  SYM,LO,MSG
                                        DEFINS  SYM,LO
                                        .IF     GT,$$MSG
                                        .MCALL  .IOMG.
                                        .IOMG.  SYM,<LO-128.>,<MSG>
                                        .ENDC
                                        .ENDM   .QIOE.
                                ;
                                ; CONDITIONALLY GENERATE DATA FOR WRITING A MESSAGE FILE FOR MO
                                ;
                                        .MACRO  .IOMG.  SYM,LO,MSG
                                        .WORD   -^O<LO>
                                        .ASCIZ  ^MSG^
                                        .EVEN
                                        .IIF    LT,^O<$$$MAX+<LO>>,$$$MAX=-^O<LO>
                                        .ENDM   .IOMG.
                                ;
                                ; DEFINE THE SYMBOL SYM WHERE LO IS IS THE LOW ORDER BYTE, HI IS THE HIGH BYTE
                                ;
                                        .MACRO  .WORD.  SYM,LO,HI
                                        DEFINS  SYM,<^O<HI*400+LO>>
                                        .ENDM   .WORD.


QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710  25-MAR-75 14:23  PAGE  7

  000000                                QIOSYS  DEFSG
          000001                        .END
```

```
QIOMAC - QIOSYM MACRO DEFINITIO MACRO D0710  25-MAR-75 14:23  PAGE  7-1
SYMBOL TABLE

IE.ABO= 177761 G    IE.IFU= 177747 G    IE.UPN= 177777 G    IO.FNA= 004400 G    IO.RWU= 002540 G
IE.ACT= 177771 G    IE.ILL= 177726 G    IE.VER= 177774 G    IO.GLC= 016430 G    IO.R1C= 002400 G
IE.ADP= 177636 G    IE.ILU= 177640 G    IE.WAC= 177743 G    IO.GLI= 016420 G    IO.SAO= 004000 G
IE.ALN= 177736 G    IE.INS= 177776 G    IE.WAT= 177741 G    IO.GNI= 016410 G    IO.SCS= 013000 G
IE.AST= 177660 G    IE.IPR= 177641 G    IE.WER= 177737 G    IO.GRC= 016450 G    IO.SDI= 013000 G
IE.BAD= 177777 G    IE.ISQ= 177703 G    IE.WLK= 177764 G    IO.GRI= 016440 G    IO.SDO= 012400 G
IE.BBE= 177710 G    IE.ITI= 177643 G    IE.2DV= 177720 G    IO.GRN= 016460 G    IO.SEC= 002520 G
IE.BCC= 177676 G    IE.ITS= 177770 G    IO.ACE= 007400 G    IO.HDX= 003010 G    IO.SEM= 002440 G
IE.BDI= 177714 G    IE.LCK= 177745 G    IO.ACR= 006400 G    IO.HIS= 015000 G    IO.SHT= 002410 G
IE.BDR= 177716 G    IE.LNL= 177646 G    IO.ACW= 007000 G    IO.INL= 002400 G    IO.SLO= 005400 G
IE.BDV= 177711 G    IE.MBK= 177721 G    IO.ADS= 014000 G    IO.ITI= 017000 G    IO.SMO= 002560 G
IE.BHD= 177700 G    IE.MOD= 177753 G    IO.APC= 014000 G    IO.KIL= 000012 G    IO.SNM= 002450 G
IE.BLB= 177672 G    IE.NBF= 177731 G    IO.APV= 014010 G    IO.LED= 012000 G    IO.SPB= 002420 G
IE.BLK= 177754 G    IE.NBK= 177727 G    IO.ATT= 001400 G    IO.LOV= 001010 G    IO.SPF= 002440 G
IE.BNM= 177712 G    IE.NDR= 177670 G    IO.CAS= 015420 G    IO.LTK= 000050 G    IO.SSO= 004400 G
IE.BTF= 177664 G    IE.NFI= 177704 G    IO.CBD= 016520 G    IO.MCS= 013400 G    IO.SST= 002430 G
IE.BTP= 177725 G    IE.NFW= 177673 G    IO.CBN= 016510 G    IO.MDA= 016000 G    IO.STC= 002500 G
IE.BVR= 177701 G    IE.NLN= 177733 G    IO.CBO= 015510 G    IO.MDI= 014400 G    IO.STP= 016400 G
IE.BYT= 177755 G    IE.NNC= 177663 G    IO.CCI= 014000 G    IO.MDO= 015400 G    IO.SYN= 003040 G
IE.CKP= 177766 G    IE.NNL= 177662 G    IO.CCT= 002460 G    IO.MLO= 006000 G    IO.TRM= 002410 G
IE.CKS= 177742 G    IE.NNN= 177674 G    IO.CIN= 016500 G    IO.MOD= 003000 G    IO.UNL= 000042 G
IE.CLO= 177732 G    IE.NOD= 177751 G    IO.CLN= 003400 G    IO.MSO= 005000 G    IO.WAT= 013400 G
IE.CNR= 177667 G    IE.NSF= 177746 G    IO.CON= 015400 G    IO.RAL= 001010 G    IO.WLB= 000400 G
IE.CON= 177752 G    IE.OFL= 177677 G    IO.CPR= 015410 G    IO.RAT= 013000 G    IO.WLS= 000410 G
IE.DAA= 177770 G    IE.ONP= 177773 G    IO.CRC= 001040 G    IO.RBC= 003000 G    IO.WLV= 000500 G
IE.DAO= 177763 G    IE.OVR= 177756 G    IO.CRE= 012000 G    IO.RCI= 015000 G    IO.WNS= 000420 G
IE.DFU= 177750 G    IE.PRI= 177760 G    IO.CRJ= 015440 G    IO.RCV= 015000 G    IO.WPB= 000440 G
IE.DNA= 177771 G    IE.RAC= 177724 G    IO.CSM= 016470 G    IO.RDB= 001200 G    IO.WVB= 011000 G
IE.DNR= 177775 G    IE.RAT= 177723 G    IO.CTI= 015400 G    IO.RDN= 000022 G    IO.XMT= 014400 G
IE.DUN= 177767 G    IE.RBG= 177730 G    IO.CTL= 016400 G    IO.REL= 013400 G    IO.XNA= 014410 G
IE.DUP= 177707 G    IE.RCN= 177722 G    IO.CTR= 015610 G    IO.RHD= 001010 G    IQ.Q  = 000002 G
IE.EOF= 177766 G    IE.RER= 177740 G    IO.DAC= 010000 G    IO.RLB= 001000 G    IQ.X  = 000001 G
IE.EOT= 177702 G    IE.RNM= 177715 G    IO.DCI= 014400 G    IO.RLV= 001100 G    IS.BV = 000005 G
IE.EOV= 177765 G    IE.RSU= 177757 G    IO.DCT= 002470 G    IO.RNA= 005400 G    IS.CLR= 000000 G
IE.EXP= 177665 G    IE.SDP= 177635 G    IO.DEL= 012400 G    IO.RNC= 001040 G    IS.CR = 006401 G
IE.FEX= 177717 G    IE.SNC= 177735 G    IO.DET= 002000 G    IO.RNE= 001020 G    IS.ESC= 015401 G
IE.FHE= 177705 G    IE.SPC= 177772 G    IO.DIS= 016000 G    IO.RNS= 001020 G    IS.PND= 000000 G
IE.FOP= 177713 G    IE.SQC= 177734 G    IO.DTI= 016000 G    IO.RPB= 001040 G    IS.SET= 000002 G
IE.HFU= 177744 G    IE.SRE= 177762 G    IO.ENA= 006000 G    IO.RTC= 003400 G    IS.SPD= 000002 G
IE.HWR= 177772 G    IE.STK= 177706 G    IO.EOF= 003000 G    IO.RTI= 016400 G    IS.SUC= 000001 G
IE.IDU= 177644 G    IE.TMM= 177671 G    IO.ESA= 002500 G    IO.RTK= 000060 G    QI.VER= 000304 G
IE.IEF= 177637 G    IE.TMO= 177666 G    IO.EXT= 011400 G    IO.RVB= 010400 G    $$MSG = 000000
IE.IFC= 177776 G    IE.ULN= 177773 G    IO.FDX= 003020 G    IO.RWD= 002400 G    ...GBL= 000001

. ABS.  000000    000
        000000    001
ERRORS DETECTED:  0

FREE CORE:  5669. WORDS
,LP:=[156,133]QIOMAC,TI:
```

APPENDIX J

FIELD SIZE SYMBOLS

Definitions for these symbols are contained in the System Library.

S.BFHD - size of FSR block buffer header in bytes

S.FATT - size of FDB file attribute area in bytes

S.FDB - size of FDB in bytes (including name block)

S.FNAM - size of filename in bytes (stored in RAD-50)

S.FNB - size of filename block in bytes

S.FNBW - size of filename block in words

S.FNTY - size of filename and file type in words (stored in RAD-50)

S.FSR2 - size of FSR2 (basic impure area)

S.FTYP - size of file type in bytes (in RAD-50)

S.NFEN - size of a complete filename in bytes -- file ID, name, type, and version

INDEX

IAS/RSX-11
I/O Operations
Reference Manual
DEC-11-OIORA-A-D

Please cut along this line.

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (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 ____________________

If you do not require a written reply, please check here. ☐

--- 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 Communications
P. O. Box F
Maynard, Massachusetts 01754

digital equipment corporation

Printed in U.S.A.