RSX-11M/M-PLUS and Micro/RSX I/O Operations Reference Manual

Order No. AA-FD14A-TC

RSX-11M Version 4.2
RSX-11M-PLUS Version 3.0
Micro/RSX Version 3.0

digital equipment corporation · maynard, massachusetts

First Printing, December 1975
Revised, December 1976
Revised, December 1977
Revised, June 1979
Revised, November 1981
Revised, July 1985

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

ZK2565

HOW TO ORDER ADDITIONAL DOCUMENTATION

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

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

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6215 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

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

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

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary or approved distributor

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

CONTENTS

PREFACE

MANUAL OBJECTIVES

The purpose of this manual is to familiarize the users of an RSX-11M, RSX-11M-PLUS, or Micro/RSX operating system with the File Control Services (FCS) facility provided with the system.

INTENDED AUDIENCE

Because the File Control Services described in this manual pertain to both MACRO-11 and FORTRAN programs, the reader is assumed to be familiar with these languages. Also, because the development of programs in an RSX-11M/M-PLUS and Micro/RSX environment requires the use of the Task Builder, the reader is also assumed to be familiar with the contents of the RSX-11M/M-PLUS and Micro/RSX Task Builder Manual.

STRUCTURE OF THIS MANUAL

Chapter 1 describes the FCS features available for RSX-11M/M-PLUS and Micro/RSX users, and defines some of the terminology used throughout the manual. This chapter is vital to understanding the balance of the manual.

Chapter 2 describes the actions you must take at assembly time to prepare adequately for all intended file I/O processing through FCS. This chapter describes the data structures and working storage areas that you must define within a particular program to use any of the File Control Services. Until you are thoroughly familiar with this chapter, you are advised to postpone reading subsequent chapters.

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

Chapter 4 describes a set of run-time routines that perform I/O functions on files, such as reading and writing directory entries, and renaming or extending files.

Chapter 5 describes the structure of files for disk, DECtapes and magnetic tapes supported by the RSX-11M/M-PLUS and Micro/RSX systems.

Chapter 6 describes two collections of object library routines. The Get Command Line (GCML) routine and the Command String Interpreter (CSI) routine may be linked with the user task to perform operations that request command line input. Such input consists of file specifications that identify and control the files to be processed by your program.

Chapter 7 describes the table-driven parser (TPARS), which provides you with the means to define and parse command lines in a unique user-designed syntax.

Chapter 8 describes queuing files for printing. You can queue files for printing at both the MACRO and subroutine levels.

Finally, the appendixes provide detailed information to help you better understand I/O operations. Appendix A and Appendix B outline the File Descriptor Block (FDB) and the filename block, respectively. Appendix C summarizes a number of I/O-related system directives that form a part of the total resource management capabilities of the RSX-11M/M-PLUS Executive. Through simplified sample programs, Appendix D illustrates the use of the macro calls that create and initialize the FDB. These sample programs also include some of the macro calls used for processing files. Appendix E illustrates the structure of the index file of a Files-11 volume. The format and content of the file header block, magnetic tape labels, and the statistics block are described in Appendixes F, G, and H, respectively. The error codes returned by the system are listed in Appendix I. Field-size symbols are listed in Appendix J. Appendix K lists RSX-11M/M-PLUS FCS library system generation options and provides a brief description of each.

ASSOCIATED MANUALS

The following manuals may be useful:

- RSX-11M/M-PLUS Information Directory and Master Index
- RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual
- RSX-11M/M-PLUS and Micro/RSX Task Builder Manual
- PDP-11 MACRO-11 Language Reference Manual

In addition, documentation for programming in any of the PDP-11 languages may be helpful.

CONVENTIONS USED IN THIS MANUAL

Unless otherwise noted, the term "RSX-11" refers to the RSX-11M, RSX-11M-PLUS, and Micro/RSX operating systems.

SUMMARY OF TECHNICAL CHANGES

This revision of the RSX-11M/M-PLUS and Micro/RSX I/O Operations Reference Manual reflects the following software technical changes and additions:

- Support for logical names, which affects information in this manual on file and ANSI magnetic tape specification syntax, and filename block routines
- Support for logical name parsing
- Support for the CSI$4 command semantic parser
- Changes in parameters for the FDAT$A, FDOP$A, and OPEN$x macros
- New information on block locking
- A new file control routine, .FLUSH (buffer flush)
- Changes to the C.EXPS user task expansion buffer size offset and to the C.CMLD command line descriptor offset in the CSI control block
- Changes to offsets within the File Descriptor Block
- New volume label offsets
- New file header format

In addition to these changes, the manual has been reorganized to make the information more easily accessible to the reader. Appendix C has also been revised to include all I/O-related system Executive directives.

CHAPTER 1

FILE CONTROL SERVICES

This chapter describes the File Control Services (FCS) features available for RSX-11/M-PLUS and Micro/RSX users. It defines some of the terminology used throughout the manual. FCS enables you to perform record-oriented and block-oriented I/O operations, as well as additional operations required for file control. Open, close, wait, and delete are some of these additional operations. The term FCS as used in this manual is a substitute for FCSRES, a memory-resident library. This memory-resident library contains commonly used routines that are linked with your task at task-build time. These routines may also be linked with your task from a system object module library (SYSLIB.OLB). There are three kinds of FCS:

- ANSI - supports ANSI format magnetic tape and big buffers
- Non-ANSI - does not support ANSI tape or big buffers
- Multibuffered - supports ANSI tape, big buffers, and multiple buffers

When your task uses commonly requested functions such as OPEN$, which opens a file, and CLOSE$, which closes a file, the Task Builder resolves the address of these routines in FCSRES, thereby eliminating these routines from your task image. Thus, commonly used FCS routines do not significantly increase the size of your task image. If you do not link your task with FCSRES at task-build time, the routines must come from SYSLIB and are included in your task image, increasing its size. These routines, consisting of pure, position-independent code, provide an interface to the file system, enabling you to read and write files on file-structured devices and to process files by using logical records.

Your program regards logical records as data units that are structured in accordance with application requirements, rather than as physical blocks of data on a particular storage medium. To meet the application's requirements, FCS allows a collection of data--distinct logical records--to be written to a file in a way that enables you to retrieve the data from the file without having to know the exact format in which it was written to the file. FCS, therefore, is transparent to your task; records can be read or written in logical units that are consistent with particular application requirements.

To invoke FCS functions from your task or application, your task 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 that you request (see Figure 1-1).

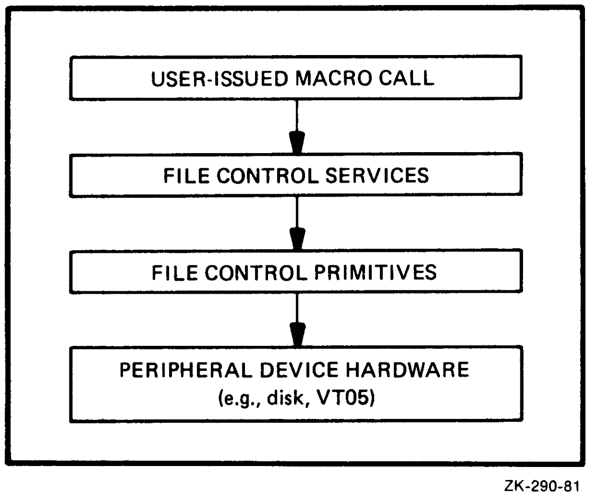


Figure 1-1 File Access Operation

1.1 KEY TERMS USED THROUGHOUT THIS MANUAL

Listed next are terms used throughout this manual, which have unique definitions in the context of FCS operations.

File Descriptor Block (FDB)

The 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 your program by issuing the FDBDF$ macro call (see Section 2.3.1.1). Each file to be opened simultaneously by your program must have an associated FDB. Portions of the FDB, which may be defined by you or the system, are maintained by FCS. Assembly-time or run-time macro calls are provided for you to initialize 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 (see the File Specification entry in this section) that FCS uses. Initially, as a file is opened, FCS fills in the filename block with information that you specify and that is taken from the dataset descriptor or the default filename block (see the next entry). Section 2.4 describes how FCS fills in the filename block from a file specification; the format and content of the filename block are described in Appendix B.

Default Filename Block

An area allocated within your program by issuing the NMBLK$ macro call (see Section 2.5.2) that contains the various elements of a file specification. You create the default filename block, whereas the filename block within the FDB is maintained by FCS. You create the default filename block to supply file specifications to FCS that are not otherwise available through the dataset descriptor (see the next entry). FCS takes these file specifications and creates a parallel structure in the FDB that contains information that FCS requires during execution time 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 in mind whenever these terms appear in this manual. These areas are structurally identical, but they are created and used differently, and may contain different information at different times.

Dataset Descriptor

A 6-word block in your program that contains the sizes and the addresses of ASCII data strings that together constitute a file specification (see Section 1.9). This 6-word block, which you also create, is described in detail in Section 2.4.1. Unless the filename block in the FDB has been initialized, you must provide FCS with dataset-descriptor or default filename block information before the specified file can be opened.

Dataset Descriptor Pointer

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

File Specification

The unique file identification that names a file, specifies the location, and allows it to be explicitly referenced by any task. The operating system, or your task, must refer to files by using a file specification. The file specification contains specific information that must be made available to FCS before that file can be opened. See Section 1.9 for a description of a file specification.

File Storage Region (FSR)

An area of memory that you reserve for use in I/O operations (see Section 1.7.3). You can allocate this area by issuing the FSRSZ$ macro call in your program (see Section 2.7).

1.2 IMPORTANT FCS CHARACTERISTICS

You should be aware of the following FCS characteristics when using its I/O facilities:

- I/O operations initiated by READ$ and WRITE$ macros are asynchronous; you are responsible for coordinating all block I/O activity.
- I/O operations initiated by GET$ and PUT$ macros are synchronized entirely by FCS; control is not returned to your program until the requested GET$ or PUT$ operation is complete.
- FCS macro calls save and restore all registers, with the following exceptions:
  - The file-processing macro calls (see Chapter 3) place the File Descriptor Block (FDB) address in R0.
  - Many of the file control routines (see Chapter 4) return requested information in the general registers.

- The macro that defines and allocates the space for the File Descriptor Block (FDB) is the FDBDF$ macro (see Section 2.3.1.1). Once the FDB is allocated, necessary information can be placed in this data construct through any logical combination of assembly-time or run-time macro calls (see Sections 2.3.1 and 2.3.2, respectively). Certain information must be present in the FDB before FCS can open and operate on a specified file.

- For each assembly-time FDB initialization macro call, a corresponding run-time macro call is provided that supplies identical information. Although both sets of macro calls (see Table 2-1 in Section 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 that create specific data, while the run-time calls generate MOV or MOVB instructions that place desired information in the FDB during program execution.

- 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 Carry bit in the Processor Status Word is set, and an error indicator is returned to FDB offset location F.ERR.

NOTE

When you use the READ$ or WRITE$ macros to execute system I/O, the IOSB parameter must be specified for F.ERR and the Carry bit to be properly returned (see Section 3.15).

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

1.3 FCS DATA STRUCTURES

In addition to generating calls to FCS subroutines, FCS macros issued by your task create and maintain certain data structures that file I/O operations require. These required data structures include the following:

- A File Descriptor Block (FDB) that contains information necessary for processing the file.

- A dataset descriptor that FCS accesses to obtain ASCII file name information required to open a specified file.

- A default filename block that FCS accesses to obtain default file name information to open a specified file. FCS accesses the default filename block when complete file information is not specified in the dataset descriptor.

- A file storage region (FSR) that FCS uses for working storage. The FSR is described in Section 1.3.3.

1.3.1 File Descriptor Block (FDB)

The File Descriptor Block (FDB) contains information that FCS uses to open and process files. One FDB is required for each file that your program opens simultaneously. You initialize 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 information that your task or the system defines:

- File attribute section
- Record or block access section
- File open section
- Block buffer section
- Filename block portion

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 2.3. Appendix A describes the format and the content of the FDB.

1.3.2 Dataset Descriptor and Default Filename Block

You must specify either a dataset descriptor or a default filename block for each file that you intend to open. These data structures provide FCS with the file specifications required for opening a file. Although either the dataset descriptor or the default filename block is usually specified, you may also specify both for the same file. The dataset descriptor and the default filename block are further described in detail in Sections 2.5.1 and 2.5.2.

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 an efficient means of opening files. Section 2.6 describes this process in detail.

1.3.3 File Storage Region (FSR)

The file storage region (FSR) is an area allocated in your program as working storage for record I/O operations (see Section 1.7). The FSR consists of two program sections that are always contiguous. 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. You determine the size of this area at assembly time by issuing the FSRSZ$ macro call (see Section 2.7). The number of block buffers and associated headers is based on the number of files that you intend to open simultaneously for record I/O operations.

$$FSR2 - This area of the FSR contains impure data that FCS uses and maintains when performing both record and block 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 2.8 shows you how to do this.

1.4 FILE ACCESS METHODS

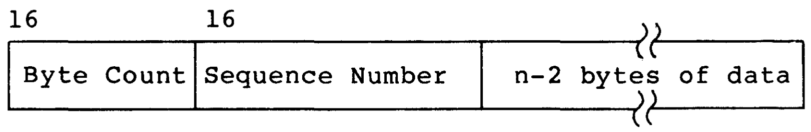
RSX-11M/M-PLUS and Micro/RSX systems support both sequential and random access to data in files on sequential access devices (such as magnetic tapes and card readers) and random access devices (such as disks). The sequential access method is device independent; that is, sequential access is usable on both record-oriented and random access devices (for example, card reader and disk). You can use the random access method only for random access devices.

1.5 DATA FORMATS FOR FILE-STRUCTURED DEVICES

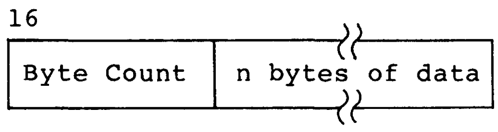
Data is transferred between peripheral devices and memory in blocks. A data file consists of virtual blocks, each of which may contain one or more logical records created by your program. In FCS terms, a virtual block in a file consists of 512(decimal) bytes for random access devices. The size of the logical records in the virtual blocks is under the control of the program that originally wrote the records.

When creating a new file, your program can specify that the records in the file will differ in size. Such records are known as variable-length records. Conversely, if your program indicates that all records in the new file will be equal in size, the records are known as fixed length.

There are two types of variable-length records: sequenced and nonsequenced. Both must be word aligned. Sequenced variable-length records are preceded by a 2-word record header. The first word contains the length of the record, and the second word contains the value of the sequence number:



Nonsequenced variable-length records are preceded by a single-word record header containing the length of the record:



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 1. A virtual block number is a file relative value, whereas a logical block number is a volume relative value. Ordinarily, records may cross block boundaries. Crossing block boundaries means that the beginning of a record can fill out the end of a block, while the rest of the record occupies the beginning of the next block.

1.5.1 Data Formats for ANSI Magnetic Tape

You can use both fixed- and variable-length records on magnetic tape; their format conforms to the ANSI standard.

On magnetic tape, a virtual block corresponds to a physical record. The default length of a block is 512 bytes. Its length can be changed to any value from 8 through 2048 bytes (14 through 2048 bytes for a write function) with the use of the FDBF$ macro (see Section 2.3.1.6). Records are not allowed to cross block boundaries.

Fixed-length records are packed into a block with no control information and no padding for alignment. The block is truncated so that it ends at the word boundary immediately following the last record that will fit in the block buffer.

Variable-length records are preceded by a 4-byte count field, expressed in decimal in ASCII characters. The count includes the length of the record and the 4-byte count field. After the last record in a block (if there is any space left in the block), a caret character ("^", ASCII code 136) appears where the next byte count should be, signaling the end of data in that block.

1.6 BLOCK I/O OPERATIONS

Block I/O operations provide an efficient means of processing file data, because such operations do not involve the blocking and deblocking of records within the file. Also, block I/O operations permit your task to read or write files in an asynchronous manner; that is, control may be returned to your program before the requested I/O operation is completed.

The read and write macro calls (READ$ and WRITE$) allow your task to read and write virtual blocks of data to and from a file without regard to logical records within the file. (See Sections 3.15 and 3.16 for a description of READ$ and WRITE$ macro calls.) When your task uses block I/O, the number of the virtual block to be processed is specified as a parameter in the appropriate READ$ or WRITE$ macro call. The virtual blocks so specified are processed directly in a reserved buffer in your task's memory space. Your task can use READ$ and WRITE$ only on block-structured devices.

You are responsible for synchronizing all block I/O operations. Such asynchronous operations can be coordinated through an event flag (see Section 2.9.1) specified in the READ$/WRITE$ macro call. The system uses the event flag to signal the completion of a specified block I/O transfer, enabling you to coordinate those block I/O operations that are dependent on each other.

1.7 RECORD I/O OPERATIONS

Sequential access mode I/O operations can be performed for both fixed- and variable-length records. Random access mode I/O operations can be performed only for fixed-length records. Your program accesses records randomly by specifying a record number. This number represents the position of the desired record within the file (viewing the file as an array of fixed-sized records, with the number 1 representing the first record physically present in the file, and n the last).

The GET$ and PUT$ macro calls (see Sections 3.9 and 3.12, respectively) are provided for processing individual records in files. Using the FSR block buffers (see Section 1.3.3), the GET$ and PUT$ routines perform the necessary blocking and deblocking of records within the virtual blocks of the file, allowing your program to access logical records. Successive GET$ or PUT$ operations in random access mode can access records anywhere within the file. To do so, your program need only modify the record number specified as part of the random record operation. After each such random operation, FCS increases by one the record number used in the operation. If your program does not again modify this number prior to issuing another record operation, the record actually accessed is the next sequential record in the file.

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

Because GET$ or 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 (for example, when the end of a data block is reached). Therefore, all GET$ or PUT$ I/O requests do not necessarily involve an actual physical transfer of data.

The data flow during record I/O operations is shown 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 deblocking of records during input occurs in the FSR block buffer, and the blocking of records occurs in the FSR block buffer during output. Note also that FCS serves as your task's interface to the FSR block buffer pool. All record I/O operations, which are initiated through GET$ and PUT$ macro calls, are synchronized by FCS unless multibuffering is in use.

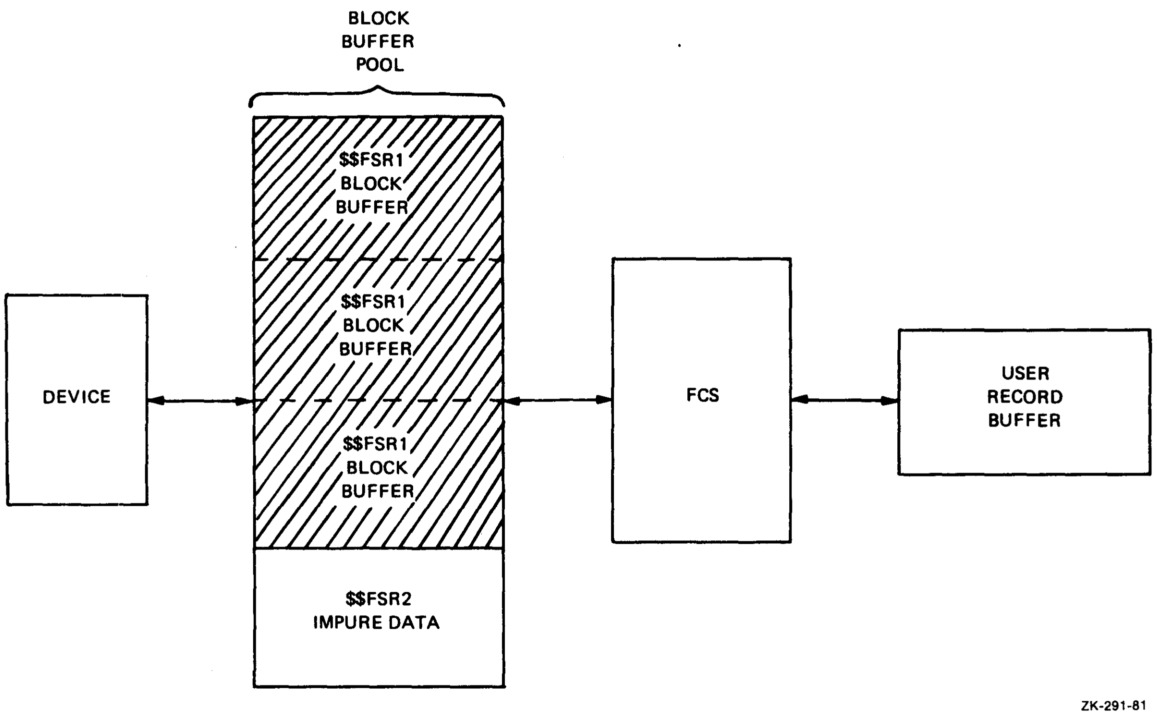


Figure 1-2 Record I/O Operations

1.7.1 Record I/O Data-Transfer Modes

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

- In move mode, by specifying that individual records are to be moved from the FSR block buffer to a record buffer that you have defined (see Figure 1-2)

- In locate mode, by referencing a location in the File Descriptor Block (see Section 1.3.1) that contains a pointer to the desired record within the FSR block buffer

1.7.1.1 **Move Mode** - Move mode requires that data be moved between the FSR block buffer and a record buffer that you have defined. For input, data is first read into the FSR block buffer from a peripheral device and then moved to your task's record buffer for processing. For output, your program builds a record in your task's record buffer; FCS then moves the record to the FSR block buffer, from which it is written to a peripheral device when the entire block is filled.

Move mode simulates the reading of a record directly into your task's record buffer; thus the blocking and deblocking of records is transparent.

1.7.1.2 **Locate Mode** - Locate mode enables your task to access records directly in the FSR block buffer. Consequently, there is normally no need to transfer data from the FSR block buffer to your task's record buffer. To access records directly in the FSR block buffer, refer to locations in the File Descriptor Block (see Section 1.3.1 and Appendix A) that contain values defining the length and the address of the desired record within the FSR block buffer. These values are present in the FDB as a result of FCS macro calls that you issued.

Program overhead is reduced in locate mode because records can be processed directly within the FSR block buffer. Moving data to your task's record buffer in locate mode is required only when the last record of a virtual block crosses block boundaries.

1.7.2 Multiple Buffering for Record I/O

By supporting multiple buffers for record I/O, FCS provides the ability in multibuffered FCS (see Appendix K) to read data into buffers in anticipation of user program requirements, and to write the contents of buffers while your program is building records for output. You can thus overlap the internal processing of data with file I/O operations, as illustrated in Figure 1-3.

When your task uses read-ahead multiple buffering, the file must be sequentially accessed to derive full benefit from multiple buffering. For write-behind multiple buffering, you can use any file access method with full benefit.

When your task uses multiple buffering, you must allocate sufficient space in the FSR for the total number of block buffers in use at any given time. The FSRSZ$ macro call (see Section 2.6.1) allocates space for FSR block buffers.

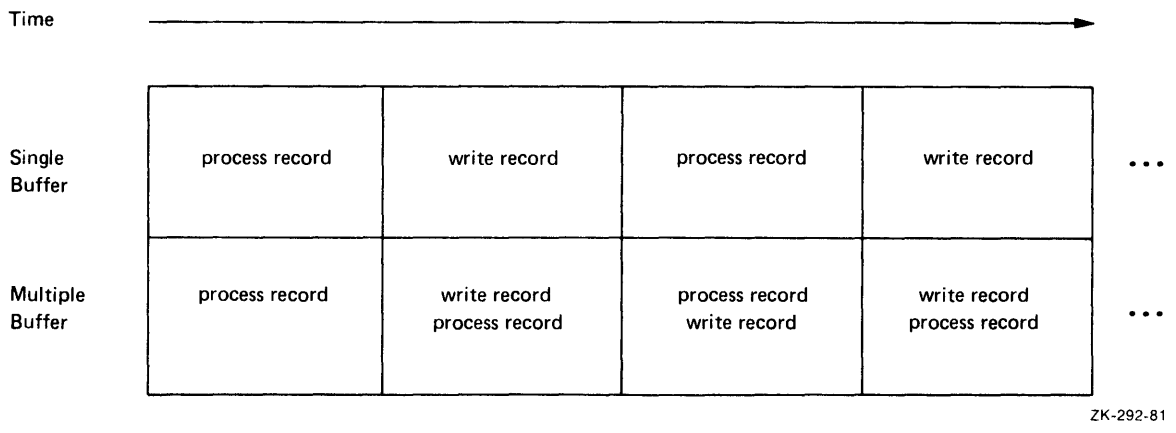


Figure 1-3 Single Buffering Versus Multiple Buffering

1.7.2.1 **Multiple Buffering Performance** - Multiple buffering can improve performance for I/O-bound tasks under certain circumstances. However, multibuffer processing in random mode is not very efficient. When using multibuffering in random mode, a user record buffer is always required. If one is not supplied, the task's low memory may be overwritten and the task may abort.

For example, consider an I/O-bound task running as the dedicated or highest priority application on a system. For such a task, multiple buffering can decrease execution time by enabling overlap of I/O and task execution.

However, if other tasks run at the same priority as that of the application task described previously, then an overlap of I/O and task execution is already achieved among these tasks without multiple buffering. In this case, multiple buffering would use up address space and pool without improving execution speed. If virtual and physical address space is available, big buffering would improve performance (see Section 1.7.3). However, big buffer processing in random mode is not very efficient.

1.7.3 Big Buffering for Record I/O

If the task uses large records or operates on clusters of records, big buffering is advantageous. The use of big buffering assumes that it is reasonable to use more task address space and physical memory for increased buffer space, and more pool for the increased number of outstanding I/O packets.

Big buffering reduces the number of disk accesses by allowing multiblock input and output. Normally, the disk accesses for GET$ or PUT$ operations are performed one sector at a time. Using FCS big buffers allows you to read or write a specified number of sectors in a single operation.

When using big buffering in random mode, a user record buffer is always required. If one is not supplied, the task's low memory may be overwritten and the task may abort. Using big buffering with random GET$ and PUT$ can cause data to be lost from the end of a file. In this case, a directory of the file would indicate more blocks in use than it had allocated. To prevent this condition from happening, follow these steps:

- Preallocate enough space to make writing an extension unnecessary.
- Execute a FLUSH operation after the highest-numbered record is written by a PUT$ macro.
- After a PUT$ macro, arrange not to execute any GET$ macro that could cause the file to extend.

To use big buffers, you must select the buffer size and specify that buffer size in the parameter lists for each occurrence of both the FSRSZ$ macro and the FDBDF$ macro in your program.

You should choose a buffer size that is a multiple of 512(decimal) bytes, the size of one disk block. Because the default amount allocated by a file extend is five blocks and disks often contain many 5-block files or parts of files, a buffer size of five blocks is generally a good choice. Larger amounts may increase performance, but note that you are trading large amounts of memory for speed.

You must reserve the buffer space in your program and you must specify the buffer size to the FDB. The FSRSZ$ macro allows you to specify the total buffer space needed. Specify 512(decimal) bytes for each normal disk file, plus the buffer size that you have selected for each big buffered file. For example, assume that a program has three files: one normal file (512-byte buffer); one file with a big buffer size of three blocks; and one file with a big buffer size of five blocks. The following call to the FSRSZ$ macro reserves the space properly:

```
    FSRSZ$  3,<<1+3+5>*512.>
```

In the FDB of each file that has a big buffer, you must override the default buffer size, using either the FDBF$A macro or the FDBF$R macro. For a file with five blocks as a big buffer, the assembly-time macro call is:

```
    FDBF$A  ,<5*512.>
```

On RSX-11M-PLUS and Micro/RSX systems, the SYSLIB provided as the default library contains all the proper FCS modules for big buffer support. RSX-11M user tasks must link to ANSLIB for these modules.

1.8 SHARED ACCESS TO FILES

The Files-11 disk architecture permits shared access to files according to established conventions. You can issue one of two macro calls, among several available in FCS for opening files, to invoke these conventions. The OPNS$x macro call (see Section 3.2) specifically opens a file for shared access. The OPEN$x macro call (see Section 3.1), on the other hand, invokes generalized open functions that have shared-access implications only in relation to other I/O requests subsequently issued. Both macro calls take an alphabetic suffix that 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, whereas the suffixes W, M, U, and A all apply to the writing of a file. You can use the OPNS$x and OPEN$x macro calls as follows for shared access to files:

1. When the OPNS$R macro call is issued, read access to the file is granted unconditionally, regardless of the presence of one or more concurrent write-access requests to the file. (The OPNS$R macro call permits concurrent write accesses to the file while it is being read.) Subsequent write-access requests for this same file are honored. Thus, several active read-access requests and one or more write-access requests may be present for the same file. However, multiple tasks simultaneously accessing the file for write operations are subject to certain restrictions, as detailed in number 2.

2. While FCS allows concurrent write-access requests through the use of the OPNS$W, OPNS$M, OPNS$U, and OPNS$A macro, synchronizing access to the file is your task's

responsibility. To avoid the retrieval or storage of inconsistent data, each task must implement and use some mechanism, which you define, ensuring that the file is serially accessed.

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

Note from the previous text that 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 mean that the access requests are all from separate tasks. A file could also be shared by a single task that has opened the file on several different logical unit numbers.

Table 1-1 shows the circumstances under which Files-11 permits a second file access when the file is opened for shared access.

Table 1-1
Shared File Access

	First Access			
Second Access	Read	Shared read	Write	Shared write
Read	Yes	Yes	No	No
Shared Read	Yes	Yes	Yes	Yes
Write	No	Yes	No	No
Shared Write	No	Yes	No	Yes

1.9 FILE SPECIFICATION SYNTAX

A full file specification has the following elements, in the order listed:

device
directory
name
type
version

A file specification has the following format:

```
device:[directory]filename.filetype;version
```

An example of a full file specification follows:

LB:[1,*]SUPLIB.OLB;0 is a full file specification.

1.9.1 Device

The device element of the file specification names the device on which the file resides. For unit-record devices, such as terminals and line printers, this is the only significant element in the file specification.

Except for logical names, the device specification consists of two alphabetic characters specifying the device name, followed by 0- to 3-character octal numeric string specifying the device unit number, followed by a colon (:). FCS converts lowercase alphabetic characters to uppercase before passing them to the operating system. The device unit number must not exceed 377 octal; if no unit number is given, FCS assumes unit 0.

For example:

db2: and DB02: are equivalent device specifications.

SY: and sy00: are equivalent device specifications.

login: and LOGIN: are equivalent logical device specifications.

1.9.2 Directory

The directory element of the file specification names the directory through which the file can be found on the device. For ANSI magnetic tape files, this element is not significant (see Section A.2).

If you use numbered directories, the directory specification can take either of the following forms:

[group,member]

or

<group,member>

Note that the delimiting characters ([] or <>) and the comma (,) must appear as shown. The group and member subelements each consist of a 1- to 3-digit octal number in the range of 0 to 377 octal. In situations where wildcards are permitted, you can substitute a single asterisk (*) character for the group or member subelement, or both, to indicate that all such elements are acceptable.

You can explicitly request the current default directory by specifying [] or <> as the directory specification.

The following are equivalent directory specifications:

[27,36] or <027,036>

The following show the use of various wildcard substitutions:

[27,*] indicates all members in group 27.

[*] indicates all directories.

[] indicates the current default directory.

If you use named directories, the directory specification can take any of the following forms:

```
[namenamen]  or  <namenamen>

or

[001009036]  or  <001009036>

or

[name09030]  or  <name09030>
```

Note that the delimiting characters ([] or <>) must appear as shown. The name may consist of as many as nine characters. The characters must be only the 36 alphanumeric characters from A through Z and 0 through 9. In situations where wildcards are permitted, you can substitute a single asterisk (*) character for the named directory.

1.9.3 Name

The name element of the file specification is the name by which the file is known in the directory. The name specification is a 0- to 9-character alphanumeric string. That is, the alphabetic characters A to Z, the numbers 0 to 9, the underscore (_), and the dollar sign ($) are all allowed. FCS converts lowercase alphabetic characters to uppercase before passing them to the operating system.

In situations where wildcards are permitted, you can substitute an asterisk (*) character in the name string for any string including the null string.

For example, the following names are acceptable within a file specification:

MyFile.; is interpreted as MYFILE..

*.; matches all names.

.; is interpreted as the null name of 0 length.

1.9.4 Type

The type element of the file specification is the type by which the file is known in the directory. The type specification consists of a period (.) followed by a 0- to 3-character alphanumeric string. FCS converts the lowercase alphabetic characters to uppercase before passing them to the operating system. In situations where wildcards are permitted, you can substitute asterisk (*) characters for any string including the null string.

The following examples show some of the conversions that FCS makes:

.dat	is interpreted as .DAT
.*	is interpreted as all types
.	is interpreted as the null type

1.9.5 Version

The version element of the file specification provides the version number by which the file is known in the directory. The version specification consists of a semicolon (;) followed by a 0- to 5-digit octal number in the range of 0 to 77777.

NOTE

> On RSX-11M-PLUS and Micro/RSX systems, decimal numbers are a system generation option. Decimal numbers can range from 0 through 32767.

In situations where wildcards are permitted, you can substitute a single asterisk (*) character for the octal number to indicate that all versions are acceptable. In situations where you are specifying a file that already exists, you can substitute the two characters "-1" for the octal number to specify the lowest-numbered version of the file that is known to the directory.

You can specify a version number of 0 or the null version to indicate either of the following:

- The highest-numbered version of the file that is known to the directory, when the file already exists
- A version number one greater than the highest-numbered version of the file (if any) known to the directory, when you are creating a new directory entry

The following show some conversions that FCS makes regarding version numbers:

```
;5  and ;0005 are equivalent versions.

;*  indicates all versions.

;-1 indicates the lowest-numbered version.

;   indicates the null version; this is equivalent to ;0.
```

For compatibility with other systems, FCS access methods can process version specifications beginning with a period (.) instead of semicolon (;) when the presence of a type specification eliminates ambiguity.

1.10 ANSI MAGNETIC TAPE FILE SPECIFICATION SYNTAX

The file specification format specific to magnetic tapes consists of the following elements, in the order listed:

```
device
directory
quoted string
version
```

1.10.1 Device

The device element is the same as that described in Section 1.9.1. The device must be a magnetic tape device.

1.10.2 Directory

The directory element is the same as that described in Section 1.9.2. This element has no meaning for ANSI magnetic tape files, and it is ignored if present.

1.10.3 Quoted String

FCS treats a quoted string as a unit representing both the name and type elements of a standard file specification. This mechanism allows expression of tape file names up to 17 characters in length that include the full set of ANSI "a" characters (some of which would otherwise be ignored or treated as element delimiters in a standard file specification).

You specify an ANSI name by including the name in quotation characters ("name"). If the name itself contains full quotation characters ("), you must also precede each such character with an additional full quotation character ("). FCS converts any lowercase alphabetic characters to uppercase, strips the full-quotation characters that you have added, and passes the result to the operating system without further modification (including ANSI "a" characters such as SPACE).

The following examples show the results of FCS-processed quoted strings:

```
"My File" is interpreted as MY FILE.

"""Don't Panic""" is interpreted as "DON'T PANIC".
```

1.10.4 Version

The version element of a magnetic tape file specification is the same as that for a conventional file specification (see Section 1.9.5). A version specification of ;0, ;-1, or the null version, is interpreted as any version for magnetic tape files.

1.10.4.1 Example Magnetic Tape File Specification

- An example of an ANSI magnetic tape file specification follows:

```
MM1:"MULP's file" specifies any version of MULP'S FILE on device MM1:
```

The standard file specification format described in Section 1.9 can also be used with magnetic tapes; this permits file transport to nontape devices and file accessibility by the widest possible range of software. See Appendix G of this manual for additional information concerning the use of names in ANSI magnetic tape files.

1.11 GENERATION OF A FULL FILE SPECIFICATION

When you specify the target file for an FCS operation, FCS generates a full file specification in the following manner:

1. FCS parses the filename string to determine which elements are present. You need not provide a full file specification in the filename string; however, any elements present must be

syntactically correct and in the proper order. FCS ignores any NULL, SPACE, or TAB characters that may be present in the string unless they occur within an ANSI magnetic tape quoted-string name.

2. FCS processes the default name block to determine which elements are present. You need not provide a full file specification in the default name block.

3. If the filename string does not provide a full file specification, FCS obtains missing elements from the default name block; if any elements are lost as a result of this merge, FCS provides default values for them as follows:

   - Device -- defaults to the device to which the specified logical unit is currently assigned; if the specified logical unit is not assigned to any device, defaults to SY:

   - Directory -- defaults to the current directory

   - Name, type, version -- defaults to null

1.12 LOGICAL NAMES

A logical name is a name that you or the system defines for:

- All or part of a file specification
- A physical device

To keep your program and command procedures independent of physical file specifications, you can substitute a logical name for all or part of a file specification, either interactively or from within a program or command procedure.

A logical name may contain 1 through 63 alphanumeric characters, including the special characters dollar sign ($) and underscore (_).

A logical name must have an entry in the logical name table.

If the first character of the logical name is an underscore (_), the translation process that replaces the logical name with its equivalent string removes the underscore only. Thus, the input string is not translated, the translation stops, and the resultant string remains.

A logical name may be a device name or a file name. If a logical name is a device name, it must be terminated by a colon (:).

You can assign logical names to devices such as tape drives, terminals, and line printers. The system manager may assign logical names to public disk volumes, so that you do not have to be concerned with the physical location of those volumes.

In addition, to reduce typing you can use logical names as a shorthand way of specifying files or directories that you refer to frequently. For example, you might assign the logical name HOME to your task's default disk and directory.

1.12.1 Using Logical Names for Program Input and Output

Programs that read and write data can be designed to read from or write (perform I/O) to different files or devices each time they are run. Performing I/O to different files or devices at different times is called device and file independence and is accomplished through the use of logical names.

When you write a program, you can refer to an input or output file by using a logical name. For instance, you might use INFILE to represent the data file or input device from which the program is to read, and OUTFILE to represent the file or device to which the program is to write. After your program is compiled and linked, but before it is run, you use system commands to associate the logical names you used in the program and the actual files or devices you want to use when you run the program.

The DEFINE and ASSIGN commands associate the logical names with the files or devices. They establish the correspondence between a logical name (that is, the name that you used in the program) and an equivalence name (that is, the actual file or device name that you want the program to use).

1.12.1.1 **Logical Name Tables** - The system maintains logical name and equivalence name pairs in three logical name tables.

- User logical name table -- contains logical name entries that are local to a particular task. By default, the DEFINE and ASSIGN commands place a logical name in the user logical name table.

- Group logical name table -- contains logical name entries that are qualified by a group number. These entries can be accessed only by tasks that execute with the same group number in their user identification codes (UICs) as the task that assigned the logical name.

- System logical name table -- contains entries that can be accessed by any task in the system.

1.12.1.2 **Specifying Logical Names** - Logical names and their equivalence name strings can have a maximum of 63 characters, and can form all or part of a file specification. If only part of a file specification is a logical name, it must be the leftmost component of the file specification. You can then specify the logical name in place of the device name or device and directory name in subsequent file specifications. A logical name can contain both a device name and a directory name.

The equivalence name for a logical name must contain the proper punctuation for a file specification (colons, brackets, periods). If the equivalence name is a device name, it must be terminated by a colon (:).

Logical name translation is discussed in Chapter 4.

1.13 ROUTINES INCLUDED IN FCSRES

The following lists the routines contained in all forms of FCS. However, the routines included in the overlaid version FCSRES are placed into two overlay segments. The first overlay segment for open, close, and associated user-accessible routines includes:

Routine Name	Module Name
ASCII UIC to Binary Conversion	ASCPPN
Assign Logical Unit Number	ASSLUN
Binary UIC to ASCII Conversion	PPNASC
Close	CLOSE
Delete File	DELJMP, DELETE
Delete File by File Name Block	DEL
Directory Primitives	DIRECT
Extend File	EXTEND
Expand Logical Name and Return Pointer to Expanded String	.EXPLG
File Storage Region Initialization	FINIT
Get Directory	GETDIR
Get Directory ID	GETDID
Mark for Deletion (Internal)	MKDL
Mark for Deletion (User Interface)	MRKDL
Octal to Decimal Conversion	.ODCVT
Open	OPNJMP, OPENR
Parse	PARSE
Parse Device	PARSDV
Parse Directory	PARSDI
Parse File Name	PARSFN
Print	$PRINT
Rename	RENAME
Request Logical Core Block	RQLCB
Send Data to and Start a Subsidiary Task	DSPAT
Truncate and Close File	TRNCLS
User Directive Primitives	UDIREC

The second overlay for get, put, read, write, and other user-accessible routines includes:

Routine Name	Module Name
Arithmetic Routines	ARITH
ASCII to Binary Conversion	CATB
Binary to ASCII Conversion	CBTA
Convert Double Precision to Decimal	CODMG
Double Precision Arithmetic Routines	DARITH
Edit Message	EDTMG
Edit Time and Date	EDDAT
Exit With Status	EXST
Read/Write File Storage Region 2	RWFSR2
Flush	FLUSH
Get Record	GETJMP, GET
Obtain Library Attributes	FCSTYP
Octal to Binary Conversion	.OD2CT
Parse Command Line	.CSI1, .CSI2, .CSI4, .EXPLG
Point and Mark	PNTMRK
Position Record	POSREC
Put Record	GETJMP, PUT
QIO	XQIOU
Read Block	READ
Return Position	POSIT
User Device Control Function	CONTRL
Wait	WAITU
Write Block	WRITE

CHAPTER 2

PREPARING FOR I/O

This chapter describes the macro calls that your task must invoke to provide the necessary file-processing information for the file descriptor block (FDB).

2.1 PREPARING FOR I/O -- GENERAL INFORMATION

The MACRO-11 programmer must establish the proper data base and working storage areas within the particular program to perform input/output operations. You must do the following:

1. Define a File Descriptor Block (FDB) for each file that your program is to open simultaneously (see Section 2.2).

2. Define a dataset descriptor and a default filename block, or both (see Sections 2.5.1 or 2.5.2, respectively) if you intend to access these structures to provide file specifications that FCS requires.

3. Establish a file storage region (FSR) within the program (see Section 2.6). (The initialization procedures for FORTRAN tasks are described in detail in the PDP-11 FORTRAN-77 User's Guide.)

Your task can place such information in the FDB in one of three ways:

- By the assembly-time FDB initialization macro calls (see Section 2.3.1)

- By the run-time FDB initialization macro calls (see Section 2.3.2)

- 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. Similarly, 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$

2.2 .MCALL DIRECTIVE - LISTING NAMES OF REQUIRED MACRO DEFINITIONS

You must list as arguments in an .MCALL directive all the assembly-time, run-time, and file-processing macro calls (see Table 2-1) that you intend to issue in a program. Doing so allows the required macro definitions to be read in from the system macro library during assembly.

You must write the .MCALL directive and associated arguments in the program prior to writing any macro call in the execution code of the program. If the list of macro names is lengthy in the .MCALL statement, you must specify several .MCALL directives, each appearing on a separate source line. The availability of space within an 80-byte line of source code limits the number of such names that may appear in any one .MCALL statement.

Format

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

Arguments

arg1,arg2,...,argn

A list of symbolic names that identify the macro definitions that you use in your program. If more than one source line is required to list the names of all desired macros, each additional line must begin with an .MCALL directive.

For clarity in your source code, you may list the assembly-time, run-time, and file-processing macro names in each of three separate .MCALL statements; you may list the macro names alphabetically, or mix them. None of these optional arrangements have any effect whatever on retrieving macro definitions from the system macro library.

If you are planning to invoke the command line processing capabilities of the Get Command Line (GCML) routine and the Command String Interpreter (CSI), you must list all the names of the associated macros 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 detail in the PDP-11 MACRO-11 Language Reference Manual. The sample programs in Appendix D also illustrate the use of the .MCALL directive. Note that these .MCALL directives appear as the 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 your program at task-build time from the system object library ([1,1]SYSLIB.OLB). Consult Section 4.1 for a description of these routines.

The following statements show sample uses of the .MCALL directive:

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

NOTE

You can use the macro FCSMC$ to declare the most commonly used FCS macros within the .MCALL format:

```
   .MCALL  FCSMC$
   FCSMC$
```

FCS macros declared in this manner include: OPEN$x, OPNS$x, CLOSE$, READ$, WRITE$, WAIT$, GET$, PUT$, DELET$, FINIT$, FSRSZ$, FDBDF$, FDAT$x, FDRC$x, FDOP$x, FDBF$x, FDBK$x, and NMBLK$. If other macros are required, explicit .MCALL directives must be issued. One disadvantage of using this method to declare .MCALL directives is that unused macros may take up possibly critical assembler symbol table space, thus slowing down the assembly process.

2.3 FILE DESCRIPTOR BLOCK (FDB)

The File Descriptor Block (FDB) is the data structure that provides the information FCS needs for all file I/O operations. Two sets of macro calls are available for FDB initialization: you can use one set for assembly-time initialization (see Section 2.3.1) and the other set for run-time initialization (see Section 2.3.2). Use the run-time macros to supplement or override information specified during assembly. The FDB sections are described in Appendixes A and B.

2.3.1 Assembly-Time FDB Initialization Macros

Assembly-time initialization requires that the FDBDF$ macro call be issued (see Section 2.3.1.1) 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 that accomplish these functions are described in the following sections.

Format

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

Macro Name

mcnam$A

The symbolic name of the macro.

Parameter

p1,p2,...,pn

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
```

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 as

```
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, that is, if any field in the macro call contains a null specification, the corresponding cell in the FDB is not initialized and thus remains 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. Specifying multiple values in this manner is mentioned throughout this manual if applicable to the macro call.

Throughout the descriptions of the assembly-time macros in this section and elsewhere in this manual, symbols of the form F.xxx or F.xxxx are referenced (for example, 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, you can reference or modify information within the FDB without having to calculate word or byte offsets to specific locations.

Using such symbols in either system software or your software also permits the relative position of cells within the FDB to be changed (in a subsequent release, for example) without affecting your current programs or the coding style employed in developing new programs.

2.3.1.1 **FDBDF$ - Allocate File Descriptor Block (FDB)** - The FDBDF$ macro call is specified in a MACRO-11 program to allocate space within the program for an FDB. This macro call must be specified in the source program once for each input or output file that your program simultaneously opens during execution. Any associated assembly-time macro calls (see Sections 2.3.1.2 through 2.3.1.6) must then be specified immediately following the FDBDF$ macro if you want to initialize certain portions of this FDB during assembly.

Macro Name and Label

label: FDBDF$

label

A symbol, which you specify, that names this particular FDB and defines its beginning address. This label is particularly significant 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 show how the FDBDF$ macro calls might appear in your 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 these example macro calls for your program to access each file simultaneously. FDBs can be reused 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 simultaneously opened file.

2.3.1.2 **FDAT$A - Initialize File Attribute Section of FDB** - The FDAT$A macro call initializes 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 first four parameters of the FDAT$A initialization macro need not be specified because FCS obtains the necessary information from the first 14 bytes of the file attribute section. The file attribute section is in the header block of the specified file. (See Appendix F.)

Format

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

Parameter

rtyp

A symbolic value that defines the type of records to be built as the new file is created. One of three 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
- R.SEQ - Indicates variable-length sequenced records are to be written in creating the file

The rtyp parameter initializes FDB offset location F.RTYP. Because symbols R.FIX, R.VAR, and R.SEQ initialize the same location in the FDB, these values are mutually exclusive.

ratt

Symbolic values that may be specified to define the attributes of the records as the new file is created.

The following 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 (for example, FD.CR!FD.BLK).

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 will 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 (for example, a line printer or a terminal).
- FD.BLK - Indicates that records are not allowed to cross block boundaries.
- FD.PRN - Indicates that the record is preceded by a word containing carriage control information; this value is the print file format attribute. Files that have this attribute set must also be sequenced files; that is, files that have the bit R.SEQ set in byte F.RTYP in the FDB.

  In a file with attribute FD.PRN, also known as COBOL carriage control, each record is associated with its own print format word, which describes the carriage control for that record, if the record is output to a unit record device such as a terminal or line printer. A program using FCS can read or write a file with attribute FD.PRN, but FCS ignores and does not interpret the format word if the file is written to a terminal. Thus, PIP correctly copies such a file from disk to disk, but a copy to TI: may not achieve the desired carriage control.

  FCS does not interpret the FD.PRN format word because such an enhancement would make FCS larger. Files with the FD.PRN attribute on RSX systems are rare.

  Files with the print file format attribute are a subset of sequenced files. Sequenced files are identified by record type R.SEQ in FDB field R.RTYP. Sequenced files have records of variable length; each record is associated with a 1-word sequence number. (Note that sequential is not the same as sequenced. Sequential means that the file is not an RMS indexed or relative file. All sequenced files are also sequential.)

When a program is reading a sequenced file with FCS in record mode, FCS returns the record in the normal manner on a GET$; the sequence number is returned in FDB field F.SEQN. Conversely, when writing a sequenced file with FCS in record mode, FCS writes the record in the normal manner and writes the associated sequence number from F.SEQN.

The sequence number field can contain any pattern of bits. A frequent application of this field is its use as a line number for text files.

The difference between a file with attribute FD.PRN and any other sequenced file is that the sequence number is considered to be the carriage control format word. This word has a particular meaning in a file with attribute FD.PRN. Each byte of the format word describes the carriage control for the associated record. The low byte describes carriage control action that should occur before the record is printed; the high byte describes carriage control action that should occur after the record is printed.

FCS operates on files with attribute FD.PRN in the same way that it operates on any other sequenced file, by using the FDB field F.SEQN for the format word. Each byte of the format word is defined as follows:

Bits 0-6			Bit 7	Meaning
0			0	No carriage control.
1-127			0	Bits 0-6 are a count of line records.
Bits 0-4	**Bit 5**	**Bit 6**	**Bit 7**	**Meaning**
1-31.	0	0	1	Bits 0-4 define a 7-bit ASCII control character to be output.
1-31.	1	0	1	Bits 0-4 are translated as an 8-bit ASCII control character ranging from 128. to 159. to be output.
0	1	1	1	Reserved for future use.

Because print format files must be sequenced files, FCS allows FD.PRN as an attribute of a new file only if record type R.SEQ is also specified. For example:

```
    FDBDF$                      ;Allocate space for FDB
    FDAT$A                      ;Print file format
```

FCS does not create a file with attribute FD.PRN that has a record type other than R.SEQ. In this case, FCS returns an error -45., IE.RAT, "illegal attribute bits set."

rsiz

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 (for variable-length records). If R.VAR or R.SEQ 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

A signed numeric value that defines the number of blocks that are 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 when the file is created; also indicates 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 when the file is created, not necessarily contiguously; also indicates that the file is to be noncontiguous

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

(You can specify an allocation of up to 24 bits by using the .EXTND routine.)

If you can estimate how long the file might be, it is more efficient to allocate the required number of blocks through this parameter when the file is created than to require FCS to extend the file when the file is written. (See the aloc parameter in the following text.)

If this parameter is not specified, an empty file is created; that is, 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 0. Thus, the usual procedure is to initialize this location at run time just before opening the file. Reinitialization is necessary if the FDB is reused.

aloc

A signed numeric value that defines the number of blocks by which the file is extended, if FCS determines that file extension is necessary as records are written to 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; also indicates 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 noncontiguously as additional space within the file; also indicates that the file is to be noncontiguous.

NOTE

> Once a file has had blocks allocated, all future file extensions cause the file to become noncontiguous, even when aloc is a positive value.

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:

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

The volume default extend size is established through the INITIALIZE, INITVOLUME, or MOUNT command. 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 example statement shows a sample of an FDAT$A macro call. This statement initializes the FDB in preparation for creating a new file containing fixed-length, 80-byte records that will be allowed to cross block boundaries.

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

In the previous example statement, 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, follows the last explicit specification, and their absence need not be indicated by trailing commas in the parameter string. Because the aloc parameter has been omitted, file extension (if it becomes necessary) is 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 in the macro following:

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

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

2.3.1.3 **FDRC$A - Initialize Record Access Section of FDB** - The FDRC$A macro call initializes the record access section of the FDB, and indicates whether to use record or block I/O operations in processing the associated file.

If you want to use record I/O operations (GET$ and PUT$ macro calls), the FDRC$A or the FDRC$R macro call (see Section 2.3.2) establishes the FDB information necessary for record-oriented I/O. However, if you want to use block I/O operations (READ$ and WRITE$ macro calls), the FDBK$A macro call (see Section 2.3.1.4) or the FDBK$R macro call (see Section 2.3.2) must also be specified 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.

You must appropriately initialize the FDB to indicate whether record or block I/O operations are to process the associated file, prior to issuing the OPEN$ macro call to initialize file operations.

Format

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

Parameter

racc

Specifies which variation of block or record I/O is to process the file. This parameter initializes the record access byte (offset location F.RACC) in the FDB. The first value shown next, FD.RWM, applies only for block I/O (READ$ or WRITE$) operations; all remaining values are specific to record I/O (GET$ or PUT$) operations:

- FD.RWM - Indicates that READ$ or WRITE$ (block I/O) operations are to process the file. If this value is not specified, GET$ or PUT$ (record I/O) operations process the file 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 process the file. If this value is not specified, sequential access mode processes the file by default. See Section 1.5 for a description of random access mode.

  The following statement shows a sample FDRC$A macro call issued for a file that may be accessed in random mode:

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

  You specify the address of the task's record buffer through the symbol BUF1, and the size of the buffer (in bytes) by the numeric value 160(decimal).

- FD.PLC - Indicates that locate mode is to process the file. If this value is not specified, move mode processes the file.

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

If you specify more than one value in the record access (racc) field, an exclamation point (!) must separate the multiple values, as shown here:

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

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

If you want your task to perform a PUT$ operation within the body of a file, the .POINT routine described in Section 4.10.1 may be called. This routine positions the file to a byte you specify within a virtual block in preparation for the PUT$ operation. The .POINT routine also permits a limited degree of random access to a file.

If FD.INS is not specified, a PUT$ operation within the file truncates the file at the point of insertion; that is, 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-file resets the logical end of the file to a point just beyond the inserted record.

urba

The symbolic address of your task's record buffer 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, and is specified only for record I/O operations.

urbs

A numeric value that defines the size (in bytes) of your task's record buffer used for GET$ operations in move and locate modes, and for PUT$ operations in locate mode. This parameter initializes FDB offset location F.URBD, and is specified only for record I/O operations.

You allocate and label a record buffer in a program by issuing a .BLKB or .BLKW directive. The address and the size of this area are then passed to FCS as the urba and the urbs parameters shown previously. For example, a task's record buffer may be defined through a statement that is logically equivalent to:

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

RECBUF is the address of the buffer and 82(decimal) is its size (in bytes).

Beginning a task's record buffers on a word boundary can improve performance by allowing FCS to move the data with MOV instructions rather than MOVB instructions.

Under certain conditions, you 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.

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

Use of the FDBK$A macro call implies that the FDRC$A macro call has also been specified, because the FD.RWM parameter of the FDRC$A macro call initially declares block I/O operations. Thus, for block I/O operations, the FDRC$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 that is functional for either record or block I/O operations.

The block I/O and record I/O FDB-initialization macros use the same area of the FDB for different data. Therefore, if record I/O operations are to be employed, neither the FDBK$A nor the FDBK$R macro call must be issued.

Format

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

Parameter

bkda

The symbolic address of an area in your task's memory space to be employed as a buffer for block I/O operations. This parameter initializes FDB offset location F.BKDS+2.

bkds

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 size specified must be an even, positive (the sign bit must not be set) value; the maximum number of bytes that can be specified is 32766. If an integral number of blocks is to be specified, the practical maximum number of bytes that can be specified is equal to 63 virtual blocks, or 32256(decimal) bytes.

bkvb

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 2.3.2.1. In short, assembly-time initialization of FDB offset locations F.BKVB+2 and F.BKVB with the virtual block number is meaningless, because any version of the generalized OPEN$x macro call resets the virtual block number in these cells to 1 as the file is opened. Therefore, these cells can be initialized only at run time through either the FDBK$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 should 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 to maintain the proper position of any remaining field(s) in the parameter string.

bkef

A numeric value that specifies an event flag to be used during READ$ or 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(decimal) is used by default.

The function of an event flag is described in further detail in Section 2.9.1.

bkst

The symbolic address of a 2-word I/O status block in your 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 you specify the bkst parameter, 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.

NOTE

If an error 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, the error is not reported unless you specify an I/O status block address. You are advised to specify this parameter, which allows the return of block I/O status information and permits normal error reporting.

The creation and function of the I/O status block are described in detail in Section 2.9.2.

bkdn

The symbolic address of an optional AST service routine, which you code. 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 2.9.3.

The following example shows an FDBK$A macro call that uses 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 your program that will accommodate a 240(decimal)-byte block. The virtual block number is null (for the reasons stated previously in the description of this parameter), and the event flag to be set upon block I/O completion is 20(decimal). 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.

2.3.1.5 **FDOP$A - Initialize File-Open Section of FDB** - The FDOP$A macro call initializes the file-open section of the FDB. In addition to a logical unit number, you would normally specify a dataset descriptor pointer and a default filename block address, or both, for each file that is to be opened. The latter two parameters provide FCS with the linkage necessary to retrieve file specifications from these data structures that you created 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 an efficient method of opening a file, is described in detail in Section 2.6.

The dspt and dfnb parameters represent address values that point to data structures that you created in the program. These data structures, which are described in detail in Section 2.5, provide file specifications to the FCS file-processing routines.

Format

```
FDOP$A  lun,dspt,dfnb,facc,actl
```

Parameter

lun

A numeric value that specifies a logical unit number. This parameter initializes FDB offset location F.LUN. All I/O operations performed 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 1 through the largest value specified to the Task Builder through the UNITS option. This option specifies the number of logical units that the task is to use (see the RSX-11M/M-Plus and Micro/RSX Task Builder Reference Manual.

dspt

The symbolic address of a 6-word block in your task containing the dataset descriptor. This data structure, which you created, consists of a 2-word device descriptor, a 2-word directory descriptor, and a 2-word file name descriptor, as outlined in Section 2.5.1.

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.

When the Command String Interpreter (CSI) processes 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

The symbolic address of the default filename block. This structure is allocated within your task 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

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, that an old file having the same file specification is not to be superseded by the new file. Rather, an error code is to be returned if a file of the same file name, type, and version exists.
- FA.TMP - Indicates, in combination with FO.WRT, that the created file is to be a temporary file.
- FA.SHR - Indicates that the file is to be opened for shared access. Shared access is also a precondition for block locking.

The facc parameter initializes FDB offset location F.FACC. The symbolic values FO.xxx, described previously, 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

A symbolic value that specifies the following control information in FDB location F.ACTL:

- Magnetic tape position
- Whether a disk file that is opened for write is to be locked if it is not properly closed; for example, the file may not be properly closed if the task terminates abnormally
- Number of retrieval pointers to allocate for a disk file window
- Whether to enable block locking

Normally, FCS supplies default values for F.ACTL. However, if FA.ENB is specified in combination with any of the symbolic values described in the following text, 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 following values can be used 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 creating 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 for more information on tape positioning.
- FA.RWD - Is specified to cause a magnetic tape to be rewound when the file is opened or closed.

  Examples of using FA.ENB with FA.POS and FA.RWD are provided in Section 5.2.7.
- 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 default number of retrieval pointers is the file-window mapping pointer count parameter (/WIN) included in the INITIALIZE or MOUNT MCR commands; the default value for this parameter is 7. Retrieval pointers point to contiguous blocks of the file on disk. Access to fragmented files may be optimized by increasing the number of retrieval pointers, that is, by increasing the size of the window. Similarly, because retrieval pointers use up pool space, additional memory can be freed by reducing the number of pointers for files with little or no fragmentation, for example, contiguous files.

- FA.LKL!FA.EXL - Is specified to lock all accessed blocks. FCS permits limited block locking to coordinate the access of the same file by two or more tasks. All tasks accessing the file must open the file for shared access by setting bit FA.SHR in FDB field F.FACC (the field access byte).

  See the RSX-11M/M-PLUS I/O Drivers Reference Manual or the Micro/RSX I/O Drivers Reference Manual for further information on block locking. Also, see Section 2.8.4 of this manual.

As noted, if neither the dspt nor the dfnb parameter is specified, the corresponding offset locations F.DSPT and F.DFNB contain 0. In this case, no file is currently associated with this FDB. Any attempt to open a file with this FDB results 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$A macro call; but they may also be initialized at run time through the FDOP$R or the generalized OPEN$x macro call (see Section 3.1).

The following examples show how the FDOP$A macro call may be used in your source program:

```
     FDOP$A  1,,DFNB
```

In this example that the dataset descriptor pointer parameter (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.

```
     FDOP$A  2,OFDSPT
```

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

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

This 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 the default filename block, or both, 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.

```
     FDOP$A  1,CSIBLK+C.DSDS
```

This example shows a macro call that takes as its second parameter a symbolic value that 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 invoking the CSI$ macro call. All considerations relevant to the use of CSI as a dynamic command line processor are presented in Section 6.2.

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

This example shows the use of the parameter actl to increase the number of retrieval pointers in the file window to 16. FA.ENB causes the contents of F.ACTL, rather than the defaults, to be used.

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

2.3.1.6 **FDBF$A - Initialize Block Buffer Section of FDB** - The FDBF$A macro call initializes the block buffer section of the FDB when record I/O operations (GET$ and PUT$ macro calls) process files. 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.

Format

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

efn

A numeric value that specifies the event flag that FCS uses to synchronize record I/O operations. This numeric value initializes FDB offset location F.EFN. FCS uses this event flag internally; you must not set, clear, or test it.

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

ovbs

A numeric value that specifies an FSR block buffer size, in bytes, that overrides the standard block size for the particular device associated with the file. This parameter initializes FDB offset location F.OVBS with the specified block buffer size.

When you use ovbs to specify an FSR block buffer size for disks, specify the desired number of bytes in integral multiples of 512. bytes, overriding the one-sector, standard 512. bytes block buffer size. You can specify block buffer sizes up to 63 sectors (32256. bytes) for disks. Increasing the block buffer size in this manner greatly reduces average disk access time, because several contiguous sectors are generally read or written during a typical disk access operation. An override block size of 2048. bytes (4 sectors) or 2560. bytes (5 sectors) is recommended, because 2048. bytes also provides ANSI magnetic tape buffer capability, and 2560. bytes is the Files-11 default extend size. Note that once the file has been opened, FCS uses the ovbs field for other purposes. Thus, if your task uses the FDB for additional disk I/O operations, the ovbs parameter must be issued in an FDBF$R macro prior to accessing the disk.

NOTE

When you specify block buffer sizes greater than 1 sector (512. bytes), you must increase accordingly the size of $$FSR1. This is done by specifying an appropriate value for the bufsiz parameter in the FSRSZ$ macro call (see Section 2.7.1).

Routines that read ANSI-standard magnetic tape without prior knowledge of the format of the files to be read must specify an override block size of 2048. bytes. This value is sufficient for the largest ANSI-standard tape blocks.

Issuing the CLOSE$ macro call (see Section 3.8) resets offset location F.OVBS in the associated FDB to 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.

On certain devices, such as line printers and terminals, the block size should not exceed the device's line width. The task can obtain the proper block size for these devices by issuing the Get LUN Information system directive for each device. (See the description for the Get LUN Information directive in the RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual.) The standard block size for each device is established at system generation time, or by the MCR SET/BUF command.

mbct

A numeric value that specifies the multiple buffer count, that is, the number of buffers FCS uses in processing the associated file. This parameter initializes FDB offset location F.MBCT. If this value is greater than 1, multiple buffering is effectively declared for file processing. In this case, FCS employs either read-ahead or write-behind operations, depending on which of two symbolic values is specified as the mbfg parameter (see the following entry).

If the mbct parameter is specified as null or 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 1. If desired, this value can be modified, as noted in the discussion of the mbfg parameter in the following entry.

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 can 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.

You 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 you.

When input is from record devices (for example, a card reader), F.MBCT should not be greater than 2.

mbfg

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; that is, one or the other, but not both, may be specified.

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

If these default buffering conventions are not desired, you 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 0 each time the associated file is closed.

NOTE

When using write-behind multibuffering, there is no gain in efficiency if the size of the file must be increased to make room for the data to be written. If a file is being written at the end, using default extension, there will be one extend operation for each five write operations; thus, only 80% of the write-behind operations will actually be overlapped with processing. This percentage can be increased as follows:

- Space for the file can be completely preallocated, either by using the cntg parameter in the FDAT$A macro, or by using the .EXTND subroutine.
- The default extension amount can be increased from five blocks by using the aloc parameter of the FDAT$A macro call. For example, if an aloc parameter of 10(decimal) is specified, the number of write-behind operations that will be overlapped will increase to 90%.
- The file can be accessed using random I/O. Because issuing PUT$R macros to access random preexisting locations in the file does not require extends, the percentage of overlapped operations is increased.

You can change the default buffer count, 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 1. This default value may be changed, as follows:

- Apply a global patch to .MBFCT at task-build time to specify the desired number of buffers.
- For MACRO-11 programs, use the EXTSCT option of the Task Builder (see Section 2.7.1) to allocate more space for the FSR block buffers; for FORTRAN programs, use the ACTFIL option of the Task Builder (see Section 2.8.2) to allocate more space for the FSR block buffers.

Because the previous procedure alters the default buffer count for all files to be processed by your 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, you can set the buffer count in F.MBCT for a specific file to 1 by issuing the following example 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 previous example that the event flag (efn) and the override block buffer size (ovbs) parameters are null; these null values are for illustrative purposes only and should not be interpreted as conditional specifications for establishing single-buffered operations.

The following examples show how the FDBF$A macro call may be used in a program:

```
     FDBF$A  25.,,1
```

This example specifies that event flag 25(decimal) synchronizes record I/O operations and that single buffering is used in processing the file.

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

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

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

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

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

2.3.2 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 in Section 2.2.1 following).

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

There are no default values for run-time FDB macros (except for the FDB address). At run time, the values currently in the FDB are used unless they are explicitly overridden. For example, values stored in the FDB at assembly time are used at run time unless they are overridden.

2.3.2.1 **Run-Time FDB Macro 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 here:

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

  - If the parameter is an address value or a literal value that is to be placed in the FDB, that is, 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
    ```

  - If the parameter is the address of a location containing an argument that is to be placed in the FDB, the parameter must 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.

- But, if the parameter is a register specifier (for example, R4), the parameter must not be preceded by the number sign (#). Register specifiers are defined MACRO-11 symbols and are valid expressions in any context.

NOTE

R0 can only be specified in the first parameter (FDB address). Any other use of R0 will fail. (See Section 2.2.2.2, items 1 and 2.)

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.

- As noted in the description of the FDBK$A macro call in Section 2.3.1.4, assembly-time initialization of the FDB with the virtual block number is meaningless, because issuing the OPEN$x macro call to prepare a file for processing resets the virtual block number in the FDB to 1. For this reason, the virtual block number can be specified only at run time after the file has been opened. Do this by issuing either the FDBK$R macro call or the I/O-initiating READ$ or 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, you must reserve and label a 2-word block in the program to temporarily store the virtual block number appropriate for intended block I/O operations. Because your task 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 (that is, the label) of this field as the bkvb parameter in the selected run-time macro call, you can make the virtual block number available to FCS.

  In preparing for block I/O operations, you must follow these procedures:

  1. At assembly time, reserve a 2-word block in your 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.

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

     ```
     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. You must ensure that the sign bit in the high-order word is not set.

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

4. Issue either the FDBK$R macro call or the READ$ or 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 that supplies 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 0 if 16 bits are sufficient to express the desired virtual block number. The I/O-initiating READ$ or WRITE$ macro call may then be issued.

   Should you choose, however, 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 following:

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

   The symbol VBNADR represents the address of the 2-word block in your program containing the virtual block number.

2.3.2.2 **Specifying the FDB Address in Run-Time Macros** - In relation to the second item of exceptions noted previously, the address of the FDB associated with the file to be processed corresponds to the address value of the symbol that you defined appearing in the label field of the FDBDF$ macro call (see Section 2.3.1.1). For example, the following statement:

```
FDBOUT: FDBDF$
```

not only allocates space for an FDB at assembly time, but also 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:

- The address of the appropriate FDB may be specified as an explicit parameter in a run-time macro call, as indicated in the following example 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 0 (R0). Whenever this method of specifying the FDB address is employed, the previous contents of R0 are overwritten (and thus destroyed). Therefore, you must exercise care in issuing subsequent run-time macro calls to ensure that the present value of R0 is suitable to current purposes.

- You may use a general register specifier as the initial parameter in a run-time macro call. When you use a register other than R0, the contents of the specified register are moved to R0. The previous contents of R0 are overwritten (and thus destroyed).

  The following statement reflects the use of a general register to specify the FDB address:

  ```
  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).

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

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

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

These three methods of specifying the FDB address also apply to all the FCS file-processing macro calls described in Chapter 3.

2.4 GLOBAL VERSUS LOCAL DEFINITIONS FOR FDB OFFSETS

Although the FDB offsets can be defined either locally or globally, the design of FCS does not require that you be concerned with the definition of FDB offsets locally. To some extent, this design consideration is based on the manner in which MACRO-11 handles symbols.

Whenever a symbol appears in the source program, MACRO-11 assumes that it is a global symbol unless it is 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's not defining the FDB offsets and bit values locally in coding the program. Such undefined symbols thus become global references, which are reduced to absolute definitions at task-build time.

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

2.4.1 Specifying Global Symbols in the Source Code

Throughout the descriptions of the assembly-time macros (see Sections 2.3.1.2 through 2.3.1.6), 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 2.3.1.3). 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 your task's 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 2.3.1.1) has been issued in the source program to allocate space for an FDB, as follows:

```
     FDBIN:  FDBDF$
```

The coding sequence shown in the following text 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, you 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 similarly 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.

2.4.2 Defining FDB Offsets and Bit Values Locally

If you want your task to declare explicitly that all FDB offsets and bit values are to be defined locally, there are two macro calls in the source program you can invoke. The first of these, FDOF$L, causes the offsets for FDBs to be defined within your program. Similarly, bit values for all FDB parameters may be defined locally by invoking the FCSBT$ macro call. You can invoke these macro calls anywhere in your program.

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

```
     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 the FDOF$L and FCSBT$ macros 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, that if the FDOF$L macro is issued, the NBOF$L macro for the local definition of the filename block need not be issued (see Section 2.5.2). The FDOF$L macro defines all FDB offsets locally, including those for the filename block.

If any of the previously named macros is to be issued in your program, it must first be listed as an argument in an .MCALL directive (see Section 2.2).

2.5 CREATING FILE SPECIFICATIONS WITHIN YOUR 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 that contains the following:

1. A device name and unit number.
2. A directory string consisting of a group number and a member number that specify 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 file name.
4. A file type.
5. A file version number.

A file specification describing the file to be processed is communicated to FCS through two data structures that you create:

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 your 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 2.3.1.5, 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 the default filename block, or both, 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 nonnull 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, because these values are defaulted to the device and unit assigned to the LUN at task-build time 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, refer to Appendix B.

2.5.1 Dataset Descriptor

The dataset descriptor is often oriented toward the use of a fixed (built-in) file name in your program. A given application program, for example, may require access only to a limited and nonvariable 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, executes to completion without requiring additional file specifications.

This structure, a 6-word block of storage that you can create manually within your program by using .WORD directives, contains information describing a file that you intend to open during the course of program execution. In creating this structure, you can define any one or all of three possible string descriptors for a particular file, as follows:

- A 2-word descriptor for an ASCII device name string
- A 2-word descriptor for an ASCII file directory string
- A 2-word descriptor for an ASCII filename string

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

DEVICENAME 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 octal unit number and an optional colon. You can create these strings by issuing statements such as these:

```
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. You can create a directory string by issuing statements such as these:

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

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

If you want your task to specify an explicit file directory different from the UIC under which you are 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 file name up to 9 characters in length, an optional 3-character file type designator, and an optional file version number. The file name and file type must be separated by a period (.), and the file version number must be preceded by a semicolon. A filename string may be created as shown following:

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

For Files-11, only the characters A through Z and 0 through 9 may be used in composing an ASCII filename string. An ANSI magnetic tape filename string may contain, in addition, the following special characters:

```
SP !  " % & ' ( ) * + , - .  / :  ; < = > ?
```

A name that contains any of these characters must be enclosed in quotation marks. If a quotation mark is part of the name, the string must contain two quotation marks. An ANSI filename string may be created as shown in the next example:

```
FILNM:  .ASCII /"PROG""2"";%&,";7/
```

The file name created in the previous example is:

```
PROG"2";%&;    ;7
```

NOTE

The semicolon is a legal character in the name string. To delimit a version number, the semicolon must be outside the quoted string.

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

A length specification of 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 your program. For example, the following code creates a dataset descriptor containing only a 2-word ASCII filename string descriptor:

```
FDBOUT: FDBDF$                    ;CREATES FDB.
        FDAT$A  R.VAR,FD.CR       ;INITIALIZES FILE-ATTRIBUTE SECTION.
        FDRC$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 labeled 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 stored in FDB offset location F.DSPT. This value enables the .PARSE routine (see Section 4.7.2) 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 that contain the size and the address of the filename string, respectively. The filename string itself is explicitly defined in the .ASCII directive that follows.

Note that the statements defining the filename string need not be physically contiguous to 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 your program may be defined in this manner.

2.5.2 Default Filename Block - NMBLK$ Macro

As noted earlier, you may also define a default filename block in the program as a means of providing required file information to FCS. For this purpose, you can issue the NMBLK$ macro call 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 your 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 in the following text 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.

Format

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

Parameter

label

A symbol, which you define, that names the default filename block and defines its address. This label is the symbolic value normally specified as the dfnb parameter when the FDOP$A or the FDOP$R macro call is issued. This causes FDB offset location F.DFNB to be initialized with the address of the default filename block.

fnam

The default file name. 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

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

The default file version number (binary). 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

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

unit

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 alphanumeric characters A through Z and 0 through 9 may be used in composing the filename and filetype strings discussed previously. Although the file version number and the unit number discussed previously are binary values, these numbers are normally represented in octal form when printed, when input by 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 that 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:

- 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 4.7.2), which is invoked as a result of issuing any version of the generalized OPEN$x macro call.
- A dataset descriptor has not been created in your program.
- A dataset descriptor is present in your 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 that initialize FDB offset location F.DSPT.

The following code 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 FILE NAME AND FILE TYPE.
     IFNAM:  NMBLK$  INPUT,DAT,,DT,1 ;ESTABLISHES FILE NAME, FILE TYPE,
                                         ;DEVICE NAME, AND UNIT NUMBER.
```

The first NMBLK$ macro call in the previous coding sequence 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 your 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 file name 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.

You can define the offsets for a filename block locally in your program 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, because the filename block offsets are defined locally as a result of issuing the FDOF$L macro call.

If you want, you may initialize fields in the default filename block directly with appropriate values. You can do this by placing 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 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. You can manually initialize any of the offsets within the default filename block in this manner to establish desired values or to override previously initialized values.

NOTE

The NMBLK$ macro cannot be used to create a file name containing non-Radix-50 characters or a file name that is not in the normal filenam.typ format. A program that uses the file name format permitted for ANSI magnetic tape must set up the file name in a dataset descriptor.

2.5.3 Dynamic Processing of File Specifications

If you want your task to make use of routines available from the system object library ([1,1]SYSLIB.OLB) for processing command line input dynamically, consult Chapter 6. Chapter 6 describes the Get Command Line (GCML) routine and the Command String Interpreter (CSI) routine, both of which may be linked with your program to provide all the logical capabilities required in processing dynamic terminal input or indirect command file input.

2.6 OPTIMIZING FILE ACCESS

When certain information is present in the filename block beginning at the symbolic F.FNB 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 your 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:

- 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. The file identification field is maintained by the system and ordinarily need not be of concern to you.

- 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.

- 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 one of three ways:

- By issuing any version of the generalized OPEN$x macro call for a file associated with the FDB in question.
- By initializing the filename block manually by using the .PARSE routine (see Section 4.7.1) and the .FIND routine (see Section 4.8.1).
- By moving the necessary values into the filename block.

Opening an existing file by file ID is a special case (see Section 3.5).

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

To understand how to effect the process of opening a file by file ID, note that the initial issuance of the generalized OPEN$x macro call (see Section 3.1) for a given file first invokes the .PARSE routine (see Section 4.7.2). The .PARSE routine is linked into your program, along with the code for OPEN$x. This routine first zeros the filename block and then fills it in with information taken from the dataset descriptor and the default filename block, or both.

Thus, issuing the generalized OPEN$x macro call invokes the .PARSE routine each time a file is opened. The .PARSE function, however, can be bypassed altogether in subsequent OPEN$x calls by saving and restoring the filename block before attempting to reopen 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 reopening that file exists within the filename block. Therefore, before closing that file, the entire filename block can be copied into your task's memory space and later restored to the FDB at the desired point in program flow for use in reopening that same file.

Your task can reopen files in this manner because FCS is sensitive to the presence of any nonzero value in the first word of the file identification field of the filename block. When your task invokes the OPEN$x function, FCS first examines offset location N.FID of the filename block. If the first word of this field contains a value other than 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$x or CLOSE$ sequence, the first word of this field is zeroed as the file is closed.

In opening files by file ID, you need only ensure that 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 proceed as follows:

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

2. Save the filename block by copying it into your task's memory space with appropriate MOV instructions. The filename block begins at offset location F.FNB in the file descriptor block (FDB).

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 2.5.2) may be invoked in your 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. Moreover, you must 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 reopened, restore the filename block to the FDB by reversing step 2.

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

Although you must save only the file identification, device name, and unit number fields of the filename block in anticipation of reopening a file by file ID, you are advised to save the entire filename block. The file name, file type, file version, and directory-ID fields, and so forth, 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.

Though you may be logically finished with file processing and may want to delete the file, the delete operation will not work properly unless the entire filename block has been saved and restored.

2.6.2 Manually Initializing the Filename Block

In addition to saving and restoring the filename block in anticipation of reopening a file by file ID, you can also initialize the filename block manually. If you choose to do so, the .PARSE and .FIND routines (see Sections 4.7.2 and 4.8.1, 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 that make such manual operations desirable, especially if your program is operating in an overlaid environment. In this case, it is highly desirable that the code for opening a file be broken into small segments in the interest of conserving memory space. Because 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 you want your task 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 your 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; 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 nonzero value in the first word (N.FID) of the filename block. If this field contains any value other than 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 nonzero value.

Because many programs continually reuse FDBs, the CLOSE$ function (see Section 3.8) puts zeros in 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 your task 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.

2.7 INITIALIZING THE FILE STORAGE REGION

The file storage region (FSR) is an area allocated in your 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$) operations, you must issue the FSRSZ$ macro 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 next.

2.7.1 FSRSZ$ - Initialize FSR at Assembly Time

The MACRO-11 programmer establishes the size of the FSR at assembly time by issuing an FSRSZ$ macro call. This macro call does not generate any executable code. It merely allocates space for a block-buffer pool in a program section named $$FSR1. The amount of space allocated depends on information provided by you, or defaulted, during the macro call.

NOTE

The FSRSZ$ macro allocates the FCS impure area that is pointed to by a fixed location in your task's virtual memory. This pointer is not altered when overlays are loaded; therefore, the FSRSZ$ macro must be invoked in the root segment of a task. Unpredictable results may occur if the FSRSZ$ macro is invoked in more than one parallel overlay.

Format

```
     FSRSZ$  fbufs,bufsiz,psect
```

Parameter

fbufs

A numeric value that you establish as follows:

- If no record I/O processing is to be done, fbufs equals 0. A value of 0 indicates that an unspecified number of files may be open simultaneously for block I/O processing. For example, if you intend to access three files for block I/O operations and no files for record I/O operations, the FSRSZ$ macro call takes 0 as an argument:

  ```
  FSRSZ$  0
  ```

  No other parameters need be specified unless the function of the psect parameter is required.

- If record I/O, using a single buffer for each file, is to be done, fbufs represents the maximum number of files that can be open simultaneously for record I/O processing. For example, you might want to access simultaneously three files for block I/O and two files for record I/O. You would specify the following FSRSZ$ macro call:

  ```
  FSRSZ$  2
  ```

  Additional parameters, bufsiz and psect (described subsequently) could also be specified as required.

- If record I/O with multiple buffering is to be done, fbufs represents the maximum number of buffers ever in use simultaneously among all files open concurrently for record I/O. Assume, for example, that your program will simultaneously access four disk files for record I/O operations. Assume further that you want double-buffering for three of the disk files and have, therefore, specified a multiple buffer count of 2 in the FDBF$A macro calls (refer to Section 2.3.1.6) for the associated files. You would then issue the following FSRSZ$ macro call:

  ```
  FSRSZ$  7
  ```

  This macro call indicates that a maximum of seven buffers will be in use simultaneously. This total is calculated as follows: one buffer for the single-buffered file and two buffers for each of the three double-buffered files. Additional parameters, bufsiz and psect (described next), could also be specified as required.

bufsiz

A numeric value defining the total block buffer pool space (in bytes) needed to support the maximum number of files that can be open simultaneously for record I/O. If this parameter is omitted, FCS obtains a total block buffer pool requirement by multiplying the value specified in the fbufs parameter with a default buffer size of 512 bytes. If, for example, a maximum of two single-buffered disk files will be open simultaneously for record I/O, either of the following FSRSZ$ macro calls could be issued:

```
FSRSZ$  2

FSRSZ$  2,1024.
```

If you want your task to explicitly specify block buffer pool requirements, the following formula must be applied:

```
bufsiz=(bsize1*mbc1)[+(bsize2*mbc2)...+(bsizen*mbcn)]
```

bsize1,bsize2,...,bsizen

The sizes, in bytes, of the buffers to support each file. The size of a buffer for a particular file depends on the device supporting the file if the standard block buffer size is used. Standard block sizes for devices are established at system generation time. The override block buffer size (ovbs) parameter can be used in the FDBF$x macro call to increase buffer size, as described in Section 2.2.1.6; these increases must be considered when you explicitly specify block buffer pool requirements.

mbc1,mbc2,...,mbcn

The multiple buffer counts (refer to Section 2.3.1.6) specified for the respective files.

The total value expressed by the bufsiz parameters must always represent the worst case buffer pool requirements among all combinations of simultaneously open record I/O files. The number of files (or buffers) representing the worst case is expressed as the first parameter of the macro call.

NOTE

If you use RSX-11D, you must not allocate an FSR block buffer less than 512(decimal) bytes in length for spooled output to a record-oriented device (such as a line printer).

psect

The name of the program section (PSECT) to which control returns after FSRSZ$ completes processing. If no name is specified, control returns to the blank PSECT.

2.7.2 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:

Format

```
     label:  FINIT$
```

Parameter

label

> An optional symbol, which you specify, 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.

The FINIT$ macro call should be issued in the program's initialization code. 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). However, 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 shown in the third paragraph following. 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 you, the OPEN$R macro call invokes the FINIT$ 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 as shown here:

```
     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 reinitialization of the FSR, whether or not it has been done in a previous execution of the program through an OPEN$x macro call.

It is important to realize 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 each 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.

2.8 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 Sections 2.8.1 and 2.8.2.

2.8.1 FSR Extension Procedures for MACRO-11 Programs

Increase the size of the FSR for a MACRO-11 program by following either of these procedures:

- Modify the parameters in the FSRSZ$ macro call to redefine the buffer pool requirement of files open simultaneously for record I/O processing. Reassemble the program.
- Use the EXTSCT (extend program section) command at task-build time to define the new size of the FSR. To invoke this option, specify the command in the following form:

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

  Parameter

  $$FSR1

  The symbolic name of the program section within the FSR that is reserved as the block buffer pool **length.** 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 length of the FSR, you can use either of the formulas:

- Length = (S.BFHD*fbufs)+bufsiz
- Length = fbufs*(S.BFHD+512.)

  Length Argument

  S.BFHD

  A symbol that defines the number of bytes required for each block buffer header. You can define this symbol locally in your program by issuing the following macro call:

  ```
  BDOFF$  DEF$L
  ```

  fbufs

  A numeric value representing either the maximum number of files open simultaneously for record I/O (when single buffering only is used) or the maximum number of buffers ever in use simultaneously among all files open concurrently for record I/O (when multiple buffering is used). Refer also to the description of this parameter in the FSRSZ$ macro call in Section 2.7.1.

bufsiz

A numeric value defining the total block buffer pool space (in bytes) needed to support the maximum number of files that can be open simultaneously for record I/O. Refer to the description of this parameter in the FSRSZ$ macro call in Section 2.7.1.

512.

The standard default buffer size.

The EXTSCT option is described in detail in the RSX-11M/M-PLUS and Micro/RSX Task Builder Manual.

2.8.2 FSR Extension Procedures for FORTRAN Programs

For a FORTRAN program, if an explicit ACTFIL option is not issued to the Task Builder, an ACTFIL statement with a default value of 4 is generated during task build. You may extend the size of the FSR at task-build time by issuing the command:

```
ACTFIL = files
```

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

This command, like the EXTSCT command described previously, 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 previously, 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 option is described in detail in the RSX-11M/M-PLUS Task Builder Reference Manual.

2.9 COORDINATING I/O OPERATIONS

Your programs perform all I/O operations by issuing GET$ or PUT$ and READ$ or WRITE$ macro calls. (See Chapter 3 for a complete discussion of these file-processing macro calls.) 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 macro called Queue I/O (QIO$, QIO$C, or QIO$S) 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 RSX-11 systems. Although transparent to your task, the QUEUE I/O directive is used for all FCS file access operations.

When invoked, the QIO$ macro 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 previously, two separate and distinct processes are involved in accomplishing a specified I/O transfer:

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

2. The successful completion of the requested data transfer operation

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

2.9.1 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. The system executes operations that manipulate, test, and 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 you have 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(decimal) may be defined for local use by the task. If you have not specified an event flag, FCS uses event flag 32(decimal) by default to signal the completion of I/O transfers.

Specific FDB-initialization and I/O-initiating macro calls in FCS enable you to specify event flags, if desired, that are unique to a particular task and that 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 2.3.1.6 or 2.3.2, 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 2.3.1.4 or 2.3.2, 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 event flag's state is transparent to your task, which 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 your control.

Also, a code indicating the success or failure of the QIO$ macro request resulting from the READ$ or WRITE$ macro call is returned to the Directive Status Word ($DSW). If desired, symbolic location $DSW may be tested to determine the status of the I/O request. The success/failure codes for the QIO$ macros are listed in the RSX-11M/M-PLUS I/O Drivers Reference Manual and the Micro/RSX I/O Drivers Reference Manual.

Event flag directives are described in the RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual. The relationship of event flags to specific devices is described in the RSX-11M/M-PLUS I/O Drivers Reference Manual and the Micro/RSX I/O Drivers Reference Manual.

2.9.2 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 in the READ$ and WRITE$ macro calls, that enables the system to return status information to your 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 previously. When this parameter is supplied, the system returns status information to a 2-word block reserved in your program. Although the I/O status block is used principally as a QIO$ macro housekeeping mechanism for containing certain device-dependent information, this area also contains information of particular interest to you.

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 you are performing READ$ operations, it is good practice to use the value returned to the second word of the I/O status block as the number of bytes actually read, rather than to assume 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 WRITE$ operations, the specified number of bytes is 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 that reflects the final status of the READ$ or 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 RSX-11M/M-PLUS I/O Drivers Reference Manual or the Micro/RSX I/O Drivers Reference Manual.

If the address of the I/O status block is not made available to FCS (and hence to the QIO$ macro) through any of the macro calls noted previously, 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 and makes normal error reporting easier.

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

```
     IOSTAT: .BLKW   2
```

where IOSTAT is a symbol, which you define, 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 is an integral part of issuing the desired I/O request.

2.9.3 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, you may specify the address of an AST service routine that is to be entered upon completion of a block I/O transfer. Because an AST is a trap action, it constitutes an 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 2.3.1.4 or 2.3.2, 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 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 your task 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 (for example, another I/O transfer).

If an AST service routine is not provided by you, 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, you 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 3.17) 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.

Implementing an AST service routine in your program is application dependent and must be coded specifically to meet your task's particular I/O-processing requirements. A detailed discussion of asynchronous system traps is beyond the scope of this document. Refer to the RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual for discussions of trap-associated system directives.

CAUTION

Do not execute any FCS routines while in an AST service routine. FCS maintains an impure data area that it uses as a Directive Parameter Block and as a scratch area for directives. An AST could interrupt an FCS operation that is altering this impure area. Executing an FCS routine in AST state could alter the impure area and cause unpredictable results when task execution resumes.

2.9.4 Block Locking

Block locking selectively controls access to blocks within a file while that file is being read from or written to by one or more users. Block locking can be used from FCS or RMS-11, or by issuing QIO$ macros.

You can enable block locking only when the file is opened. Once block locking is enabled, you can establish "locks," which are structures allocated from system dynamic storage that control access to specific blocks in the file.

When your task reads or writes a block, the Executive creates a lock that subsequently restricts other users from writing to or reading from that block. When your task has a file open on a LUN with block locking enabled and locks are created, your locks do not restrict your task from reading or writing blocks if you use the same LUN. Locks may be selectively eliminated by issuing a QIO$ macro with the IO.ULK (unlock) function code. You can only eliminate those blocks that you have created. When your task closes the file, all your locks on that file are released to system dynamic storage.

Block locking operates in the following ways when using FCS:

1. **Opening the File:**

   To enable block locking when opening a file from FCS, you must change two fields in the File Descriptor Block. The value FA.SHR in byte F.FACC must be set to allow shared write access to the file. Additionally, the values FA.LKL, FA.EXL, and FA.ENB must be set in word F.ACTL. Setting FA.SHR causes FCS to clear AC.LCK in the DPB. For example:

   ```
   FDOP$R  #FDB,,,,#FA.SHR,#FA.LKL!FA.EXL!FA.ENB
   OPEN$R  R0,,,,,,,,ERRSUB    ;OPEN SHARED FOR READ WITH LOCKS
   ```

2. **Writing or Reading Blocks:**

   A one-block read or write operation locks a block for exclusive access. A write or read operation of more than one block similarly locks all blocks operated on in this QIO$ macro. A file open for block mode may invoke READ$ and WRITE$ macros in the usual manner.

Note that, in general, FCS operates as follows on sequential read access:

a. OPEN$R positions the file to record 1

b. GET$ returns record 1 and positions the file to record 2

c. GET$ returns record 2 and positions the file to record 3

d. GET$ returns record 3 and positions the file to record 4

Be aware that successive GET$ macros scan across the file sequentially.

However, if you have files open for record mode operations, the following special considerations may exist:

- A number of tasks are updating records in a single file
- One of these tasks is reading records sequentially

For example, if the GET$ macro for record 2 in the task that reads blocks sequentially fails because record 2 is contained in a block previously locked by another task, FCS loses its position in the file. The next GET$ macro yields undefined results; it obtains neither record 2 nor record 3.

After this kind of error occurs, FCS must reposition its pointer to the records in the file. This can happen in one of the following ways:

- Operating in random mode on fixed-length records, FCS repositions its record pointer to the first record for each GET$ or PUT$ operation.
- FCS repositions the FCS pointer in a file of variable-length records by calling the FCS .POINT routine. You can reposition the pointer either to a location noted by a previous .MARK call, or to the beginning of the file.
- FCS closes and reopens the file to reposition the pointer to the beginning.

3. **Unlocking Blocks:**

To unlock blocks without closing the file, you must execute a QIO$ macro with the function code IO.ULK. You can use IO.ULK to unlock one block, a series of blocks, or all the blocks in an open file.

To unlock one or more blocks in a series, specify the block count in device-dependent parameter Word 2, specify the high 8 bits of the starting VBN in the low byte of parameter Word 4, and specify the low 16 bits of the starting VBN in parameter Word 5. For example, to unlock previously locked VBNs 5, 6, and 7, use the following code:

```
MOV         #3,R0             ;UNLOCK 3 BLOCKS
MOV         #5,R1             ;STARTING AT VBN 5
QIOW$S      #IO.ULK,#MYLUN,#1,,#IOSB,,<,R0,,,R1>
```

To unlock all blocks you have locked on this LUN, issue the QIO$ macro with no parameters beyond the device-independent part of the DPB, as follows:

```
QIOW$S  #IO.ULK,#MYLUN,#1,,#IOSB   ;UNLOCK ALL BLOCKS
```

Also, you can use FCS to execute the QIO$ macros for you by calling the .XIOU routine.

To use the .XIOU routine to unlock all blocks that you have locked on this LUN and file, call .XQIO with no option parameters, that is, with R2=0 as follows:

```
MOV     #FDB,R0             ;GET FDB ADDRESS
MOV     #IO.ULK             ;UNLOCK BLOCK FUNCTION
CLR     R2                  ;UNLOCK ALL BLOCKS
CALL    .XQIO               ;EXECUTE QIO
BCS     ERROUT              ;IF CS ERROR IS IN F.ERR(R0)
```

To use .XIOU to unlock one or more blocks in a series, you must set up a 5-word parameter block. Specify the count of blocks in Word 2, specify the high 8 bits of the starting VBN in the low byte of parameter Word 4, and specify the low 16 bits of the starting VBN in parameter Word 5. For example, to unlock the previously locked VBNs 5, 6, and 7, use the following code:

```
PRMBK: .WORD    0            ;PARAMETER BLOCK FOR UNLOCK QIO
       .WORD    0            ;COUNT OF BLOCKS TO UNLOCK
       .WORD    0            ;
       .WORD    0            ;HIGH 8 BITS OF START VBN
       .WORD    0            ;LOW 16 BITS OF START VBN
       MOV      #FDB,R0      ;GET FDB ADDRESS
       MOV      #IO.ULK      ;UNLOCK BLOCK FUNCTION
       MOV      #5,R2        ;FIVE PARAMETERS
       MOV      #PRMBK,R3    ;ADDRESS OF PARAMETER BLOCK
       MOV      #3,2(R3)     ;UNLOCK 3 BLOCKS
       MOV      #5,8.(R3)    ;STARTING AT VBN 5
       CALL     .XQIO        ;EXECUTE QIO
       BCS      ERROUT       ;IF CS ERROR IS IN F.ERR(R0)
```

4. **Closing the Files**

   Closing the file in the ordinary manner will release all blocks that have been established on that file for the specific task and LUN.

2.9.5 Error Codes Related to Shared Files and Block Locking

Error codes relating to file sharing and block locking may be returned in the following circumstances:

1. **Opening the File**

   IE.WAC

   Indicates that you have requested that other users be denied write access (no FCS FA.SHR or AC.LCK=1), but someone else has already opened the file to write to it.

Solution: Do not attempt to open the file until all others writing to the file have closed it.

IE.LCK

Indicates that one of the following conditions is true:

- You want to write to the file and have allowed shared write access (set FCS FA.SHR or AC.LCK=0), but someone else has already opened the file denying others write access.

  Solution: Do not attempt to open the file until all accessors without shared write access have closed the file.

- You want to write to the file and have allowed shared write access (set FCS FA.SHR or AC.LCK=0) without enabling block locking but someone else has already opened the file with block locking enabled.

  Solution: Open the file with block locking enabled.

- F11ACP cannot perform a directory operation because the directory is locked or being written to.

  Solution: The solution depends on what you anticipate as normal activity on your system. If it is legitimate for a task to access a directory, then consider attempting the operation again.

IE.ULK

Indicates that the Executive does not support block locking. This error can only be returned on an RSX-11M system that has been generated without block locking support.

Solution: Open the file without enabling block locking.

2. **Writing or Reading Blocks:**

   IE.ULK

   Returned by the Executive when any read or write error occurs that relates to block locking. It generally means that another task has locked the block.

   Solution: The solution depends on the application. Wait and retry the operation or report the error and stop processing.

3. **Unlocking Blocks:**

   IE.IFC

   Returned when the Executive does not support block locking.

   Solution: Do not attempt to unlock blocks on a system that does not support block locking.

   IE.LCK

Returned upon the occurrence of any other error. For example, IE.LCK is returned if another task has locked the blocks.

Solution: Unlock only those blocks that you have previously locked for that file.

4. **Closing the File:**

   No block locking error can occur when closing a file.

CHAPTER 3

FILE-PROCESSING MACROS

You can manipulate files through a set of file-processing macro calls. The assembler invokes and expands these macros at assembly time and the operating system executes the resulting code at run time. This chapter describes these run-time macro calls, which allow you to manipulate files and to perform the following I/O operations:

OPEN$	- Opens and prepares a file for processing
OPNS$	- Opens and prepares a file for processing and to allow shared access to that file (depending on the mode of access)
OPNT$	- Creates and opens a temporary file for processing
OFID$	- Opens an existing file using file identification information in the filename block
OFNB$	- Opens a file using file name information in the filename block
CLOSE$	- Terminates file processing in an orderly manner
GET$	- Reads logical data records from a file
GET$R	- Reads fixed-length records from a file in random mode
GET$S	- Reads records from a file in sequential mode
PUT$	- Writes logical data records to a file
PUT$R	- Writes fixed-length records to a file in random mode
PUT$S	- Writes records to a file in sequential mode
READ$	- Reads virtual data blocks from a file
WRITE$	- Writes virtual data blocks to a file
DELET$	- Removes a named file from the associated volume directory and deallocates the space occupied by the file
WAIT$	- Suspends 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 File Descriptor Block (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 macros is the symbolic address of an optional, user-defined 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 CARRY BIT IN PROCESSOR STATUS
          WORD.
          JSR     PC,ERRLOC  ;INITIATES ERROR-HANDLING ROUTINE
                             ;AT "ERRLOC" ADDRESS.
     nn$:                    ;CONTINUES NORMAL PROGRAM EXECUTION.
```

where nn$ represents a generated local symbol. If the operation is completed successfully, the Carry bit 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.

However, if an error condition is detected during the execution of the file-processing routine, the Carry 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 you must explicitly test the Carry bit in the Processor Status Word to ascertain the status of the requested operation.

Note that executing the FCS file-processing routines causes all your task's general registers to be saved except R0. FCS uses R0 by convention to contain the address of the FDB associated with the file being processed.

3.1 OPEN$X - GENERALIZED OPEN MACRO

Before any file can be processed by your task or system program, it must first be opened. An alphabetic suffix accompanying the macro name indicates to FCS the action you intend to perform on a file. For example, in issuing the following generalized macro:

```
     OPEN$x
```

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

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

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

NOTE

You can issue the generalized OPEN$x macro without an alphabetic suffix. In this case, the action to be performed on the file is indicated to FCS through an additional parameter in the macro. 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 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 here:

1. If R is specified (for reading an existing file), that file cannot also be written; that is, 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; that is, concurrent GET$ and PUT$ or READ$ and WRITE$ operations can 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 can be read, written, or extended.

The program that issues the OPEN$x macro must have appropriate access privileges for the specified action. Table 3-1 summarizes the access privileges for the various forms of the OPEN$x macro. 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

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 your task issues any form of the OPEN$x macro, FCS first fills in the filename block with file name 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 file name information specified in your task, is created in the program by issuing the NMBLK$ macro (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 can be supplied to FCS through the FDOP$A macro, the FDOP$R macro call, or the OPEN$x macro. FCS requires access to the Dataset Descriptor or the default filename block in retrieving file name information used in opening files.

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

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

2. Initializes the file attribute section of the file header block. The file header block is a file system structure maintained on the volume containing the file. Each file on a volume has an associated file header block that describes the attributes of that file. FCS obtains attribute information for a new file from the FDB associated with the file. The format and content of a 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 your task explicitly issues a particular type of macro that specifies that the file not be superseded, the old file is not deleted and an error code is returned. 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$ or PUT$) operations are processing the file.

If an existing file is to be opened, any one of the following macros 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 the default filename block, or both. FCS then searches the UFD by file name 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$ or PUT$) operations are processing the file.

NOTE

As described in Section 2.6, you allocate buffers through the FSRSZ$ macro. The number of buffers allocated is dependent upon the number of files that you intend to open simultaneously for record I/O operations.

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

3.1.1 Format of Generalized OPEN$x Macro

The OPEN$x macro takes the following general form:

Format

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

Parameters

x

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

A symbolic value of the address of the associated FDB.

lun

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. If so specified, the lun parameter need not be present in the OPEN$x macro. Each FDB must have a unique LUN.

dspt

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. If so specified, this parameter need not be present in the OPEN$x macro.

This parameter specifies the address of the manually created dataset descriptor (see Section 2.4.1). If the Command String Interpreter (CSI) interprets command lines dynamically, this parameter specifies the address of the dataset descriptor within the CSI control block (see offset location C.DSDS in Section 6.2.2).

racc

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 associated FDB. You can specify any combination of the following parameters by separating them with exclamation points:

- FD.RWM - Requests that block I/O (READ$ or WRITE$) operations are to process the file. If you do not specify this parameter, FCS assumes by default that record I/O (GET$ or PUT$) operations are to process the file.
- FD.RAN - Requests random access to the file for record I/O (GET$ or PUT$) operations. The file is opened and the first record is pointed to. With this parameter, a PUT$ operation in the file, without exception, does not truncate the file. If this parameter is not specified, FCS uses sequential access by default. Refer to Section 1.5 for a description of random access mode.
- FD.PLC - Requests locate mode (see Section 1.6.2) for record I/O (GET$ or PUT$) operations. If this parameter is not specified, FCS uses move mode (see Section 1.6.1) by default.
- FD.INS - Requests that a PUT$ 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.

Specifying 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 that of the record being replaced.

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, the racc parameters need not be present in the OPEN$x macro.

urba

The symbolic address of your task's record buffer. This parameter initializes FDB offset location F.URBD+2.

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

urbs

A numeric value defining the size of your task's record buffer (in bytes). This parameter initializes FDB offset location F.URBD.

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

err

The symbolic address of an optional user-coded error-handling routine.

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

The following examples show sample uses of the OPEN$x macro.

A macro 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 that the FDB address is assumed to be present in R0. The third parameter, that is, 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$ or PUT$ operations.

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

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

This macro 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, respectively, of your task's record buffer.

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

```
OPEN$A  #OUTFDB
```

This macro 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.

Note in all three preceding examples that the error parameter is not specified, requiring that you explicitly test the Carry bit in the Processor Status Word to ascertain the success of the specified operation.

NOTE

You can use R0 only to pass the FDB address parameter. Any other use of R0 when you issue the OPEN$A macro will fail.

3.1.2 FDB Requirements for Generalized OPEN$x Macro

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

- The assembly-time macros described in Section 2.2.1
- The NMBLK$ macro described in Section 2.4.2

- The run-time macros described in Section 2.2.2
- The various macros described in this chapter for opening files

Using any particular combination of macros to define and initialize the FDB is a matter of choice, as indicated previously. Of far greater significance is the fact that certain information must be present in the FDB before you can open the associated file. 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$W macro, the following information must first be supplied to the FDB. You can specify this information through the FDAT$A macro (see Section 2.2.1.2) or the FDAT$R macro (see Section 2.2.2):

   - The record type must be established for record I/O operations.

     The record type cannot be supplied to the FDB through any of the various macros used to create or open files (for example, OPEN$W, OPEN$R, and so forth). Furthermore, this information is required when opening an existing file, because FCS obtains such information from the first 14 bytes of your task's file attribute section of the file header block (see Appendix F).

     To establish the record type, you must initialize byte offset location F.RTYP with the following symbolic values:

     - R.FIX - Requests that fixed-length records are to be written into the file.
     - R.VAR - Requests that variable-length records are to be written into the file.
     - R.SEQ - Requests that sequenced records are to be written into the file.

   - The desired record attributes must be specified for record I/O operations.

     The record attributes cannot be supplied to the FDB through any of the various macros used to create or open files (for example, OPEN$W, OPEN$R, and so forth). Furthermore, the record attributes are required when opening an existing file, because FCS obtains such information from the first 14 bytes of your task's file attribute section of the file header block (see Appendix F).

     The record attributes are defined by initializing byte offset location F.RATT with the appropriate value(s), as follows:

     - FD.FTN - Requests that the first byte of each record contain a FORTRAN carriage-control character.
     - FD.CR - Requests that a line-feed (<LF>) character precede each record and that a carriage-return (<CR>) character follow the record when that record is output

to a device requiring carriage control information (for example, 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 - Requests that records be prevented from crossing block boundaries.

- FD.PRN - Requests that the record be preceded by a word containing carriage-control information. Files with this attribute must also be sequenced files; that is, files with the bit R.SEQ set in the byte F.RTYP in the FDB. For more information about FD.PRN as a record attribute, see Chapter 2, Section 2.2.1.2.

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

  The record size cannot be supplied to the FDB through any of the various macros used to create and open files, or both (for example, OPEN$W, OPEN$R, and so forth). Furthermore, the record size is required when opening an existing file, because FCS obtains such information from the first 14 bytes of your task's file attribute section of the file 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:

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

     - FD.RWM - Requests that READ$ or WRITE$ (block I/O) operations process the file. If this parameter is not specified, GET$ or PUT$ (record I/O) operations result by default.

     - FD.RAN - Requests that random access mode (GET$ or PUT$ record I/O) process the file. The file is opened and the first record pointed to. If this parameter is not specified, sequential access mode results by default. Refer to Section 1.5 for a description of random access mode.

     - FD.PLC - Requests that locate mode (GET$ or PUT$ record I/O) process the file. If this parameter is not specified, move mode results by default.

     - FD.INS - Requests that a PUT$ operation in sequential mode in the body of a file shall not truncate the file. If this parameter is not specified, a PUT$ 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.

- Your task's record buffer descriptors (that is, 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 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.

- You must specify the logical unit number to initialize FDB offset location F.LUN. Initializing this cell can be accomplished with the lun parameter of the FDOP$A, the FDOP$R, or the generalized OPEN$x macro. Each FDB must have a unique logical unit number.

- 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 file name information for use in opening the file. These address values may be specified in either the FDOP$A macro (see Section 2.1.1.5) or the FDOP$R macro (see Section 2.2.2). The generalized OPEN$x macro (see Section 3.1) may also be used to specify the dataset descriptor pointer.

- 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$A macro (see Section 2.2.1.6) or the FDBF$R macro (see Section 2.2.2). If not specified, FCS uses event flag number 32(decimal) by default in synchronizing all record I/O activity.

3. **To Specify Desired File Options**

   If certain options are desired for a given file, they must be specified before that file is opened. Because this information is needed only in opening the file, it is zeroed when the file is closed, thus ensuring that the FDB is properly reinitialized for subsequent use. The options that may be specified for a given file are described here:

   - The override block size (ovbs parameter) must be specified in either the FDBF$A or the FDBF$R macro 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 or if the big-buffering or multiple-buffering versions of FCS are in use. The override block size is specified to improve I/O system performance with record I/O, and with record-oriented devices (such as line printers) and sequential devices (such as magnetic tape units). (See Section 2.2.1.6.)

   - The multiple buffer count (mbct parameter) must be specified in either the FDBF$A or the FDBF$R macro to initialize FDB offset location F.MBCT. If multiple-buffered record I/O operations are to be used, this parameter must be greater than 1, and it must agree with the desired number of buffers to be used. This parameter is neither overlaid nor zeroed when the file is closed.

If the multiple buffer count is not established as described previously, multiple-buffered operations can still be invoked by changing the default buffer count in the FSR. A default buffer count of 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.

In addition, if your task uses multiple buffering, you must specify the appropriate control flag as the mbfg parameter in either the FDBF$A or the FDBF$R macro to appropriately initialize FDB offset location F.MBFG. Either of two symbolic values may be specified for this purpose, as follows:

- FD.RAH - Requests that read-ahead operations are to process the file.

- FD.WBH - Requests that write-behind operations are to process 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 when that file is being used basically for sequential read operations, but scattered updating is also being performed.

- 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. 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.

- The address of the 5-word statistics block in your program must be moved manually into FDB offset location F.STBK. This address value specifies an area in your task 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. You can define such an area in a program with coding logically equivalent to

  ```
  STBLK:  .BLKW   5
  ```

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

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

  where STBLK is the symbolic address of the statistics block, which you define, 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$x macro opens a file for shared access. This macro has the same format, that is, takes the same alphabetic suffixes and run-time parameters, as the generalized OPEN$x macro. The shared access conditions that result from the use of this macro are summarized in Section 1.8.

3.3 OPNT$W - CREATE AND OPEN TEMPORARY FILE

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

The OPNT$W macro creates a file but does not enter a file name for that file into any associated user directory file.

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

Although the OPNT$W macro takes the same format and parameters as those of the generalized OPEN$x macro, 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 (for example, 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$W macro. Perform any desired operations on that file. If the file is to be used only for a single OPNT$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 (see Section 3.8). Continue other processing in the program, as desired.

4. In anticipation of reopening the temporary file, restore the filename block to the FDB by reversing step 2.

5. Reopen the file by issuing any of the FCS macros that 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 following, to mark the file for deletion:

   ```
   CALL    .MRKDL
   ```

   The .MRKDL routine is described in Section 4.15.1.

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

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

Thus, not saving the filename block before closing a temporary file results in a "lost" file, because 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 you. This program is described in detail in the RSX-11M/M-PLUS Utilities Manual.

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

The OPNT$D macro creates and opens a temporary file. This macro is a convenient way to perform the previously shown steps 1 and 6. A file marked for deletion cannot be opened by another program. Furthermore, when the file is closed, it is 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 is that the file thus created is to be used only once. This is a particularly desirable way to open a temporary file, because the file will be deleted even if the program terminates abnormally without closing the file.

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

NOTE

> If the OPNT$D macro is used for a temporary file containing sensitive information, it is recommended that you zero the file before closing it, or reformat the disk to destroy the sensitive information. (Although a temporary file is deleted after use, the information physically remains on the volume until written over with another file and could be analyzed by unauthorized users.)

3.5 OFID$X - OPEN FILE BY FILE ID

You issue the OFID$x macro to open an existing file using information stored in the file identification field (offset location N.FID) of the filename block in the FDB (not in your default filename block). Thus, issuing this macro invokes an FCS routine that opens a file only by file ID (see Section 2.5). The OFID$x macro, which has the same format and takes the same parameters as those of the generalized OPEN$x macro (see Section 3.1), is for use with overlaid programs.

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

- 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. Saving and restoring the filename block are discussed in detail in Section 2.5.1.

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

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

1. Call the .PARSE routine (see Section 4.7.1). This routine takes information from a specified dataset descriptor or default filename block, or both, and initializes and fills in the specified filename block.

2. Call the .FIND routine (see Section 4.8.1). This routine locates an appropriate directory entry for the file (by file name) and stores the file identification information found there 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$x macro.

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

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

Opening an existing file by file ID for write access is a special case. Because it is intended to rewrite the existing file, the following occur:

- Any initial allocation (F.CNTG) is ignored.

- File access byte (F.FACC) value NA.NSP (do not supersede file) is ignored.

- File access byte (F.FACC) value FA.CRE (create new file) is set even though the file is being rewritten rather than created.

- This operation may not be performed on ANSI magnetic tape. The data in the file header labels is not changed when the file is written. See Section 5.2 for information on positioning file on tape to rewrite a file in a particular position.

The OFID$W macro is equivalent to the OFID$U macro. Invoking either OFID$W or OFID$U opens an existing file by file ID number for update and extension.

3.6 OFNB$X OPEN FILE BY FILENAME BLOCK

The OFNB$x macro either opens an existing file or creates and opens a new file using file name information in the filename block. Like the OFID$x macro previously described, the OFNB$x call is for use with overlaid programs. However, the OFNB$x macro 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.

The OFNB$x call has the same format and takes the same parameters as those of the generalized OPEN$x macro (as described in Section 3.1.1):

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

The OFNB$x macro also uses the same suffixes that are available to the OPEN$x macro: OFNB$R, OFNB$W, OFNB$M, OFNB$U, OFNB$A. The suffixes have the same meaning as they do for OPEN$x (see Table 3-1).

In describing the functions of the OFNB$x macro, the same assumptions outlined for OFID$x apply, namely, that the filename block has been saved and restored in anticipation of issuing the OFNB$x macro, or that the file is being opened for the first time. Because 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 in the following text.

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

1. The file name (offset location N.FNAM)

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

3. The file version number (offset location N.FVER)

4. The directory ID (offset location N.DID)

5. The device name (offset location N.DVNM)

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

In providing the information to the filename block, you can use either of two general procedures, as described in the following sections.

3.6.1 Dataset Descriptor or Default Filename Block

If the dataset descriptor contains all the required information listed previously, follow these procedures:

1. Call the .PARSE routine (see Section 4.7.1). This routine takes information from a specified dataset descriptor and default filename block, or both, and fills in the appropriate offsets of a specified filename block.

2. Issue the OFNB$x macro.

3.6.2 Default Filename Block Only

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

1. Issue the NMBLK$ macro (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:

   - Call the .GTDIR routine (see Section 4.9.1) to retrieve the directory ID from the specified dataset descriptor and to store the directory ID in the default filename block.

   - Call the .GTDID routine (see Section 4.9.2) to retrieve the default UIC from $$FSR2 and to store the directory ID in the default filename block.

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

4. Issue the OFNB$x macro.

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

Issuing the OFNB$x macro is usually done under the premise that the filename block contains the requisite information, as described previously. However, if the file identification field (offset location N.FID) in the filename block contains a nonzero value when the call to OFNB$x is issued, the file is unconditionally opened by file ID.

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

3.7 OPEN$ - GENERALIZED OPEN FOR SPECIFYING FILE ACCESS

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

By issuing the OPEN$ macro without an alphabetic suffix, and by specifying two additional parameters, you can inhibit the superseding 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 are identical to those specified for the generalized OPEN$x macro (see Section 3.1), with the exception of the facc parameter and the dfnb parameter. These additional parameters are described in this section.

Format

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

Parameter

facc

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

- FO.RD - Requests that an existing file is to be opened for reading only.
- FO.WRT - Requests that a new file is to be created and opened for writing.
- FO.APD - Requests that an existing file is to be opened and appended.
- FO.MFY - Requests that an existing file is to be opened and modified.
- FO.UPD - Requests that an existing file is to be opened, updated, and, if necessary, extended.
- FA.NSP - Requests, in combination with FO.WRT, that the old file having the same file specification is not to be superseded by the new file.
- FA.TMP - Requests, in combination with FO.WRT, that the file is to be a temporary file.
- FA.SHR - Requests that the file is to be opened for shared access.

dfnb

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.

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

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

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

The following macro creates a temporary file for shared access:

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

NOTE

You can use R0 only to pass the FDB address parameter. Any other use of R0 when you issue the OPEN$ macro will fail.

3.8 CLOSE$ - CLOSE SPECIFIED FILE

When the processing of a file is completed, you must close it by issuing the CLOSE$ macro. 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, which contains data for an output file, is completely written if it is partially filled (record I/O only)
3. By default, truncates the file being closed
4. Deaccesses the file
5. Releases the FSR block buffer(s) allocated for the file (record I/O only)
6. Prepares the FDB for subsequent use by clearing appropriate FDB offset locations
7. Calls an optional user-coded and user-specified error-handling routine if an error condition is detected during the CLOSE$ operation

Note that I/O does occur in items 1 and 2. Therefore, your program should include error processing for CLOSE$ calls as it would for calls to PUT$.

Issuing a CLOSE$ when a file is already closed results in a success status code. Closing a file that is already closed is not an error.

3.8.1 Format of CLOSE$ Macro

Format

```
CLOSE$  fdb,err
```

Parameter

fdb

A symbolic value of the address of the associated FDB.

err

The symbolic address of a user-coded, optional error-handling routine.

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

```
CLOSE$  #FDBIN,CLSERR

CLOSE$  ,CLSERR

CLOSE$  R0
```

The first example shows an explicit declaration for the relevant FDB, and the symbolic address of a user-coded 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 reads logical records from a file. After a GET$ operation, the next record buffer descriptors in the FDB always identify the record just read; that is, 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 your task's 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 your task's record buffer, and the address of your task's 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, regardless of whether 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.

If the record read was a sequenced record, the sequence number is stored in F.SEQN regardless of whether the GET$ was in move mode or locate mode.

GET$ operations are fully synchronous; that is, record I/O operations are completed before control is returned to your program.

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

3.9.1 Format of GET$ Macro

Format

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

Parameter

fdb

A symbolic value of the address of the associated FDB.

urba

The symbolic address of your task's record buffer that is 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

A numeric value defining the size (in bytes) of your task's record buffer. This parameter determines the largest record that can be placed in your task's record buffer in move or locate mode. When specified, this parameter initializes offset location F.URBD in the associated FDB.

err

The symbolic address of an optional error-handling routine, which you coded.

If neither the urba nor the urbs parameter is specified in the GET$ macro, FCS assumes that these requisite values have been supplied previously through the FDRC$A, the FDRC$R, or the generalized OPEN$x macro. Any resulting nonzero values in offset locations F.URBD+2 and F.URBD are used as the address and the length, respectively, of your task's 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:

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

In both move and locate mode, only data up to the amount specified in F.URBD is placed in your task's buffer. The next GET$ begins reading at the beginning of the next record.

The following statements show how the GET$ macro may be used in a program:

```
GET$    R0,,,ERROR
```

In this example, the address of the desired FDB is assumed to be present in R0. Note that the next two parameters, that is, your task's record buffer address (urba) and your task's 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. The final parameter in the string is the symbolic address of a user-coded error-handling routine.

```
GET$    ,#RECBUF,#25.,ERROR
```

This example also assumes that R0 contains the address of the desired FDB. Explicit parameters then define the address and the size, respectively, of your task's record buffer and an error handler, which you coded.

```
GET$    #INFDB
```

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

NOTE

You can use R0 only to pass the FDB address. Any other use of R0 when you issue the GET$ macro will fail.

3.9.2 The FDB 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 following text focuses mainly on whether your task's record buffer is required under certain conditions. In this regard, you should recall that your task's record buffer descriptors, that is, the urba and the urbs parameters, may be specified in the FDRC$A, the FDRC$R, or the generalized OPEN$x macro, as well as the I/O-initiating GET$ macro. These parameters must be present in the GET$ macro (to appropriately initialize the FDB) only if they were not previously supplied through other available means.

If operating in random access mode, 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. FCS increments this value after each GET$ or GET$R operation to point to the next record in the FSR block buffer.

Thus, unless your task alters the values in locations F.RCNM and F.RCNM+2 before each issuance of the GET$ or GET$R macro call, the next record in sequence is read. Your specified record buffer size (that is, the urbs parameter) always determines the largest record that can be read during a GET$ operation.

3.9.2.1 **GET$ Operations in Move Mode** - With respect to GET$ operations in move mode (refer to Section 1.7.1.1 for information on move mode), the following generalization applies. If records are always moved to the same record buffer in your task, the urba and urbs parameters need be specified only in the initial GET$ macro. Alternatively, these values may be specified beforehand through any available means identified previously, for initializing your task's 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 that involve the same record buffer in your task.

3.9.2.2 **GET$ Operations in Locate Mode** - In performing GET$ operations in locate mode (refer to Section 1.7.1.2 for information on locate mode), you should take into account the following:

NOTE

> In the following text, reference is made to the FSR block buffer. By default, the blocksize that FCS uses is equivalent to the buffer size of the device on which the file is opened. If big buffering is enabled (that is, an ovbs parameter value is specified in the FDBF$x macro as described in Section 2.2.1.6), the FSR block buffer will be more than one block long. As a result, it may not be necessary to move a record even though it crosses block boundaries, because both blocks are currently within the FSR block buffer space. Thus, moves are only necessary when the record crosses a buffer boundary, which is not necessarily the same as a block boundary in a big-buffered file.

- If fixed-length records are to be processed, and if they fit evenly within the FSR block buffer, your task's record buffer descriptors need not be present in the associated FDB.
- If fixed-length records that do not fit evenly within the FSR block buffer are to be processed, or if variable-length records are to be processed, your task's 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 the time the file is created, the records in the resulting file are not allowed to cross buffer boundaries.

  For an existing file, the 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 records within that file are allowed to cross block boundaries.

  The design of FCS requires you to use your task's record buffer only in the event that records (either fixed or variable in length) cross buffer boundaries.
- 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 the associated FDB in offset locations F.NRBD+2 and F.NRBD, respectively. However, if that record crosses buffer boundaries, it is moved to your task's record buffer. In this case, the address of your task's 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 buffer boundaries during GET$ operations in locate mode, then your task's 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 reads fixed-length records from a file in random mode. Thus, by definition, issuing this macro requires that you be familiar with the structure of the file to be read and, furthermore, that you be able to specify precisely the number of the record to be read.

The GET$ and GET$R macros are identical, except that the parameter list of GET$R includes 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$ may be used. 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$x macro, then neither GET$ nor GET$R will read the desired record.

The GET$R macro takes two more parameters in addition to those specified in the GET$ macro.

Format

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

Parameter

lrcnm

> 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(decimal) 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

> 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 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 0.
>
> If you use F.RCNM to specify a desired record number for any given GET$R operation, this cell must be cleared before issuing a subsequent GET$R macro that requires 16 bits or less to express the desired record number; otherwise, any residual value in F.RCNM yields an incorrect record number.

If the lrcnm and hrcnm parameters are not specified in a subsequent GET$R macro, the next sequential record is read because the record number in offset locations F.RCNM+2 and F.RCNM is increased by one with each GET$ operation. In the case of the first GET$R, after opening the file, record number 1 is read because the record number has been initialized to 0 by the OPEN. If a record other than the next sequential record is to be read, you must explicitly specify the number of the desired record.

The following statements represent the use of the GET$R macro:

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

In this example the number of the desired record to be read, that is, 1040(decimal), 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.

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

This 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.

NOTE

> R0 can be used only to pass the FDB address parameter. Any other use of R0 when issuing the GET$R macro will fail.

3.11 GET$S - READ LOGICAL RECORD IN SEQUENTIAL MODE

The GET$S macro reads logical records from a file in sequential mode. Although the routine invoked by the GET$S macro 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 is specifically for use in an overlaid environment in which the amount of memory available to the program is limited and files are to be read in strictly sequential mode.

If both GET$S and PUT$S are to be used by the program, note that the savings in memory usage over GET$ and PUT$ can 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 writes logical records to a file. If operating in random access mode, the number of the record to be written is maintained by FCS in offset locations F.RCNM and F.RCNM+2 of the associated FDB. FCS increases this value by one after each PUT$ or PUT$R operation to point to the next sequential record position. Thus, unless your program alters this value before issuing another PUT$ or PUT$R operation, the next record in sequence is written.

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 your choice. This buffer is not necessarily your task's record buffer previously described in the context of record I/O operations. In other words, you can build records in an area of a program apart from that normally defined by your task's record buffer descriptors in the FDB (F.URBD+2 and F.URBD). In this case, you specify the address of the record buffer so used and the size of the record in the PUT$ macro, 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. Then, if the record so built is not already in the FSR block buffer, it is moved there as the PUT$ operation is performed.

If the records in the file are sequenced records, the field F.SEQN in the FDB contains the sequence value, which you can modify.

PUT$ operations are fully synchronous; that is, record I/O operations are completed before control is returned to your task's 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.

3.12.1 Format of PUT$ Macro

Format

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

Parameter

fdb

A symbolic value of the address of the associated FDB.

nrba

The symbolic address of the next record buffer, that is, the address of the record to be PUT$. This parameter initializes FDB offset location F.NRBD+2.

nrbs

A numeric value specifying the size of the next record buffer, that is, the length of the record to be PUT$. This parameter initializes FDB offset location F.NRBD.

err

The symbolic address of an optional error-handling routine, which you coded.

The following examples show how the PUT$ macro may be used in a program:

```
PUT$    #FDBADR,,,ERRRT
```

In this example 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.

```
PUT$    ,,#160.,ERRRT
```

This 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.

```
PUT$    R0
```

This example specifies only the address of the FDB; all other parameter fields are null.

NOTE

R0 can only be used to pass the FDB address parameter as shown in the previous example; it cannot be used to pass any other parameter in the PUT$ macro.

3.12.2 The FDB Relevant to PUT$ Operations

This subsection highlights aspects of PUT$ operations in move and locate mode that have a bearing on the associated FDB.

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

If your task's record buffer is required, the specified size of that buffer (that is, 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, that is, 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 cannot fit in the space remaining in the FSR block buffer, one of two possible operations is performed:

- 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.

- 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 (see Section 2.2) is basically driven by specifying in each PUT$ macro 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:

- Your task's 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. The values so specified dynamically update offset locations F.NRBD+2 and F.NRBD in the associated FDB.

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

- If variable-length records are being PUT$, the size of each record must be specified as the nrbs parameter in each PUT$ 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, your task's record buffer is required for PUT$ operations in locate mode (see Section 2.2) only when the potential exists for records to cross buffer boundaries. In other words, if there is insufficient space in the FSR block buffer to accommodate the building of the next record, you must provide a buffer in your task's memory space 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 your record buffer is required in locate mode.

In this regard, the following generalizations apply:

NOTE

In the following discussion, reference is made to the FSR block buffer. By default, the block size that FCS uses is equivalent to the buffer size of the device on which the file is opened. If big buffering is enabled (that is, an ovbs parameter value is specified in the FDBF$x macro, as described in Section 2.2.1.6) the FSR block buffer will be more than one block long. As a result, it may not be necessary to move a record even though it crosses block boundaries, because both blocks are currently within the FSR block buffer space. Thus, moves are only necessary when the record crosses a buffer boundary, which is not necessarily the same as a block boundary in a big-buffered file.

- If fixed-length records are being PUT$ and they fit evenly within the FSR block buffer, your task's record buffer is not required.

- If a fixed-length record crosses block boundaries, your task's 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 cannot fit in the FSR block buffer, FCS sets offset location F.NRBD+2 to point to your task's record buffer. Then, when the record is PUT$, it is moved from your record buffer to the FSR block buffer.

- If a variable-length record is being PUT$, the potential task's record buffer descriptors must be present in offset exists for crossing block boundaries. In this case, your task's record buffer descriptors must be present in offset locations F.URBD+2 and F.URBD of the associated FDB. Moreover, the size of each variable-length record must be specified as the nrbs parameter in each PUT$ macro.

  Determining if FCS points offset location F.NRBD+2 to the FSR block buffer for the PUT$ operation or to your task's 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 is PUT$, as defined by the size of your task's record (F.URBD). Thus, if a record of this defined size cannot fit in the space remaining in the FSR block buffer, FCS sets offset location F.NRBD+2 to point to your task's 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 your task's record buffer to the FSR block buffer. For example, assume that your task 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 continues to point to your task's record buffer for PUT$ operations, even if you continue to PUT$ short (10- or 20-byte) records. Thus, some unavoidable movement of records takes place in locate mode.

If the largest record that you intend 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$). Aside from having to allocate a smaller record buffer in your task, PUT$ operations in locate mode are more efficient if this precaution is observed. Exercising care in this regard reduces the tendency to move records from your task's 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$R macro writes fixed-length records to a file in random mode. As noted in Section 3.10 in discussing the GET$R macro, operations in random access mode require you to be very familiar with the contents of such files. The PUT$R macro also relies entirely on you to specify the number of the record before a specified PUT$ operation can be performed. Because the usual purpose of a PUT$R operation is to update known records in a file, it is assumed that you also know the number of such records within the file.

The PUT$ and PUT$R macros are identical, except that PUT$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 macros, then neither PUT$ nor PUT$R will write the desired record.

The PUT$R macro takes two more parameters in addition to those specified in the PUT$ macro.

Format

```
     PUT$R    fdb,nrba,nrbs,lrcnm,hrcnm,err
```

Parameter

lrcnm

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 (see Section 3.10), except that it identifies the record to be written.

hrcnm

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, except that it identifies the record to be written.

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

If F.RCNM is used in expressing a desired record number for any given PUT$R operation, you must clear this cell before issuing a subsequent PUT$R macro 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$R macro, the next sequential record is written, because FCS increases the record number by one in these cells after each PUT$ operation. In the case of the first PUT$R after opening the file, record number 1 is written. Note that this is true even if the file has been opened for an append (OPEN$A). If a record other than the next sequential record is to be written, you must explicitly specify the number of the desired record.

NOTE

A random mode PUT$R operation executed in locate mode must be preceded by a call to .POSRC. Because locate mode allows you to store data directly into the block buffer, the file must be positioned so that the desired record position is in fact in the block buffer. See Section 4.10.2 for further details.

Examples of how the PUT$R macro may be used in a program follow:

```
     PUT$R    #OUTFDB,#RECBUF,,#12040.,,ERRLOC
```

In the first example, the presence of RECBUF as the next record buffer address (nrba) parameter merely indicates that you are specifying the address of the record. Although specifying this address repeatedly is unnecessary, it is not invalid. Normally, a buffer address is specified dynamically, because 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. Note also that the next record buffer size (nrbs) parameter is null, because 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.

```
     PUT$R    #FDBADR,#RECBUF,,R4

     PUT$R    #FDBADR,#RECBUF,,LRN
```

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 you have preset the desired record number in the referenced location.

NOTE

> You can use R0 only to pass the FDB address. Any other use of R0 when you issue the PUT$R macro will fail.

3.14 PUT$S - WRITE LOGICAL RECORD IN SEQUENTIAL MODE

The PUT$S macro writes logical records to a file in sequential mode. Although the routine invoked by the PUT$S macro 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 is specifically for use in an overlaid environment in which the amount of memory available to the program is limited and files are to be written in strictly sequential mode.

If both GET$S and PUT$S are to be used by the program, the savings in memory utilization over GET$ and PUT$ are 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 reads a virtual block of data from a block-oriented device (for example, a magnetic tape, a disk, or DECtape). In addition, if certain optional parameters are specified in the READ$ macro, status information is returned to the I/O status block (see Section 2.8.2), or the program traps to an AST service routine, which you coded, at the completion of block I/O operations (see Section 2.8.3).

In issuing the READ$ (or WRITE$) macro, you are responsible for synchronizing all block I/O operations. For this reason, the WAIT$ macro is provided (see Section 3.17), allowing you to suspend program execution until a specified READ$ or WRITE$ operation has been completed. It is important, however, that you test the contents of F.ERR in the FDB for error codes immediately after issuing the READ$ or WRITE$ call as well as on return from the WAIT$ call. When errors occur during multiple-block transfers, the second word of the I/O status block will contain the number of bytes transferred before the error occurred. The READ$ or WRITE operations can return error codes distinct from those that can be present on completing a WAIT$ operation. For example, IE.EOF will be returned upon completing the READ$ operation, but not upon completing WAIT$.

When your task issues the WAIT$ macro with a READ$ (or WRITE$) macro, you must ensure that the event flag number and the I/O status block address specified in both macro calls are the same.

When the WTSE$ macro waits for I/O completion, the issuing task must check I/O errors by examining the I/O status block (defined by the task). (The I/O status block is described in Section 2.8.2.) When WTSE$ is used, FCS will not return a completion code to offset F.ERR in the FDB.

3.15.1 Format of READ$ Macro

From the format following, note that the parameters of the READ$ macro are identical to those of the FDBK$A or the FDBK$R macro, 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. 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.

Format

```
    READ$   fdb,bkda,bkds,bkvb,bkef,bkst,bkdn,err
```

Parameter

fdb

A symbolic value of the address of the associated FDB.

bkda

The symbolic address of the block I/O buffer in your 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.

bkds

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. In any case, the maximum block size that may be specified for file-structured devices is 32256 bytes.

bkvb

The symbolic address of a 2-word block in your 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$A macro described in Section 2.2.1.4, assembly-time initialization of the virtual block number in the FDB is ineffective, because the generalized OPEN$x macro sets the virtual block number in the FDB to 1.

The virtual block number can be made available to FCS only through the FDBK$R macro or the I/O-initiating READ$ (or WRITE$) macro 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 nonexistent block, that is, a block beyond the end of the file. If the virtual block number references nonexistent data, an end-of-file (IE.EOF) error indication is returned to offset location F.ERR of the associated FDB; otherwise, the READ$ operation proceeds normally. If the total number of bytes goes beyond the end of the file, then as many blocks as exist are read and the byte count of the shortened transfer is returned in I/O STATUS+2. No error condition occurs, so you must check the count on each READ. An end-of-file indication is returned only if no blocks can be read.

If the virtual block number is not specified through any of the available means identified already, sequential operation results by default, beginning with virtual block number 1. The virtual block number is incremented by the number of blocks read after each READ$ operation is performed.

bkef

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, initializes FDB offset location F.BKEF; if so specified, this parameter need not be included in the READ$ (or WRITE$) macro.

If this optional parameter is not specified through any available means, event flag 32(decimal) is used by default. The function of an event flag is discussed in further detail in Section 2.8.1.

bkst

The symbolic address of the I/O status block in your task (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 or 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. If the address of this 2-word structure is not supplied to FCS through any of the available means, status information cannot be returned to your program. Regardless, the event flag specified through the bkef parameter is set to indicate block I/O completion, but, without an I/O status block, your program must assume that the operation (for example, READ$ or WRITE$) was successful.

bkdn

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$R macro, and, if so specified, need not be present in the READ$ (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

The symbolic address of an optional error-handling routine, which you coded.

The following examples represent READ$ macros that may be issued to accomplish a variety of operations:

```
READ$   R0
```

This 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.

```
     READ$    #INFDB,,,,,,ERRLOC
```

This example shows an explicit declaration of the associate FDB and includes the symbolic address of an error-handling routine, which you coded.

```
     READ$    R0,#INBUF,#BUFSIZ,,#22.,#IOSADR,#ASTADR,ERRLOC
```

In this 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 your 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.

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

This example reflects, as the last parameter in the string, the symbolic address of the 2-word block in your program containing the virtual block number.

NOTE

> You can use R0 only to pass the FDB address. Any other use of R0 when you issue the READ$ macro will fail.

3.15.2 The FDB Relevant to READ$ Operations

The READ$ macro requires that the associated FDB be initialized with certain values before it can be issued. You can specify these values through either the FDBK$A or the FDBK$R macro, or they may be made available to the FDB through the various parameters of the READ$ macro. 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)

3. The virtual block number (in offset locations F.BKVB+2 and F.BKVB)

NOTE

> When either READ$ or WRITE$ operations are performed, FCS maintains the end-of-file block number field (F.EFBK) and clears the first free byte in the last block field (F.FFBY) in the FDB. During a READ$ operation, end-of-file is determined by the end-of-file block number field in F.EFBK. If desired, you can modify F.FFBY before closing the file by using the CLOSE$ macro call.

3.16 WRITE$ - WRITE VIRTUAL BLOCK

The WRITE$ macro is issued to write a virtual block of data to a block-oriented device (for example, magnetic tape, disk, DECtape, or DECtape II). Like the READ$ macro, if certain optional parameters are specified in the WRITE$ macro, 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 you supply the address of an AST service routine and an event flag number, or both, you are responsible for synchronizing all block I/O processing. The WAIT$ macro can be issued with the WRITE$ macro to suspend program execution until a program-dependent I/O transfer has been completed. When the WAIT$ macro is used for this purpose, the event flag number and the I/O status block address in both macros must be the same. Again, as with READ$ operations, you should check for an error code immediately following the WRITE$ macro as well as on return from the WAIT$ macro.

3.16.1 Format of WRITE$ Macro

The WRITE$ macro takes the same parameters as the READ$ macro. The bkvb parameter represents the symbolic address of a 2-word block containing the number of the virtual block to be written. The virtual block number is incremented after each WRITE$ operation is performed.

Format

```
     WRITE$  fdb,bkda,bkds,bkvb,bkef,bkst,bkdn,err
```

Parameter

fdb

A symbolic value of the address of the associated FDB.

bkda

The symbolic address of the block I/O buffer in your 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.

bkds

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. In any case, the maximum block size that may be specified for file-structured devices is 32256 bytes.

bkvb

The symbolic address of a 2-word block in your 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$A macro described in Section 2.2.1.4, assembly-time initialization of the virtual block number in the FDB is ineffective, because the generalized OPEN$x macro sets the virtual block number in the FDB to 1.

The virtual block number can be made available to FCS only through the FDBK$R macro or the I/O-initiating READ$ (or WRITE$) macro after the file has been opened. The virtual block number is created as described in item 4 of Section 2.2.2.1.

The WRITE$ function checks the specified virtual block number to ensure that it does not reference a nonexistent block, that is, a block beyond the end of the file. If the virtual block number references nonexistent data, an end-of-file (IE.EOF) error indication is returned to offset location F.ERR of the associated FDB; otherwise, the WRITE$ operation proceeds normally. If the total number of bytes goes beyond the end of the file, then as many blocks as exist are read and the byte count of the shortened transfer is returned in I/O STATUS+2. No error condition occurs, so you must check the count on each WRITE. An end-of-file indication is returned only if no blocks can be read.

If the virtual block number is not specified through any of the available means identified already, sequential operation results by default, beginning with virtual block number 1. The virtual block number is incremented by the number of blocks read after each WRITE$ operation is performed.

bkef

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, initializes FDB offset location F.BKEF; if so specified, this parameter need not be included in the WRITE$ macro.

If this optional parameter is not specified through any available means, event flag 32(decimal) is used by default. The function of an event flag is discussed in further detail in Section 2.8.1.

bkst

The symbolic address of the I/O status block in your task (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 or 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. If the address of this 2-word structure is not supplied to FCS through any of the available means, status information cannot be returned to your program. Regardless, the event flag specified through the bkef parameter is set to indicate block I/O completion, but, without an I/O status block, your program must assume that the WRITE$ operation was successful.

bkdn

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 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, and, if so specified, need not be present in the 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

The symbolic address of an optional error-handling routine, which you coded.

When this macro is issued, the virtual block number (that is, 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.

The following examples show how the WRITE$ macro may be used in a program:

```
WRITE$  R0
```

This example specifies only the FDB address and assumes that all other required values are present in the FDB.

```
WRITE$  #OUTFDB,#OUTBUF,#BUFSIZ,#VBNADR,#22.
```

This 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 signalling block I/O completion.

```
WRITE$  R0,,,,#22.,#IOSADR,#ASTADR,ERRLOC
```

This 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, it specifies the address of an error-handling routine, which you coded.

NOTE

You can use R0 only to pass the FDB address. Any other use of R0 when you issue the WRITE$ macro will fail.

3.16.2 The FDB Relevant to WRITE$ Operations

WRITE$ operations require the presence of the same information in the FDB as READ$ operations (see Section 3.15.2).

3.17 WAIT$ - WAIT FOR BLOCK I/O COMPLETION

The WAIT$ macro, which is issued only with READ$ and WRITE$ operations, suspends program execution until the requested block I/O transfer is completed. This macro may be used to synchronize a block I/O operation that depends on the successful completion of a previous block I/O transfer.

As noted in Section 3.15 in connection with the READ$ macro, you can specify an event flag number through the bkef parameter. This event flag number is used during READ$ (or WRITE$) operations to indicate the completion of the requested transfer. If desired, you can issue a WAIT$ macro (specifying the same event flag number and I/O status block address) following the READ$ (or WRITE$) macro.

In this case, the READ$ (or WRITE$) operation is initiated in the usual manner, but the Executive suspends program execution until the specified event flag is set, indicating that the I/O transfer has been completed. The system then returns information to the I/O status block, indicating the success or failure of the operation. FCS then moves the I/O status block success or failure indicator into offset location F.ERR of the associated FDB, and returns with the carry condition code in the Processor Status Word cleared if the operation is successful, or set if the operation is not successful. Task execution then continues with the instruction immediately following the WAIT$ macro.

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.

3.17.1 Format of WAIT$ Macro

Format

```
    WAIT$   fdb,bkef,bkst,err
```

Parameter

fdb

A symbolic value of the address of the associated FDB.

bkef

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$ or WRITE$ macro.

If this parameter is not specified, either in the WAIT$ macro call or the associated READ$ or WRITE macro, 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.

bkst

The symbolic address of the I/O status block in your program (see Section 2.8.2). Although this parameter is optional, if it is specified, it must agree with the corresponding (bkst) parameter in the associated READ$ or WRITE$ macro.

If this parameter is not specified, either in the WAIT$ macro call or the associated READ$ or WRITE$ macro, 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. If F.BKST has not been initialized, no information is returned to the I/O status block.

err

The symbolic address of an optional error-handling routine, which you coded.

The following examples show how the WAIT$ macro may be used in a program.

```
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, because no flag number (bkef parameter) is specified, offset location F.BKEF is assumed to contain the desired event flag number. If this cell in the FDB contains 0, event flag number 32(decimal) is used by default.

The second example shows an explicit specification of the FDB address and specifies 25(decimal) 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.

The fourth example contains a null specification for the event flag number, and, in addition, specifies the address of an error-handling routine, which you coded.

Please note that the WAIT$ macro associated with a given READ$ or WRITE$ operation need not be issued immediately following the macro to which it applies. For example, the following sequence is typical:

1. Issue the desired READ$ or WRITE$ macro.
2. Perform other processing that is not dependent on the completion of the requested block I/O transfer.
3. Issue the WAIT$ macro.
4. Perform the processing that is dependent on the completion of the requested block I/O transfer.

When performing several asynchronous transfers in the same general sequence as described previously, a separate buffer, I/O status block, and event flag must be maintained for each operation. If you intend 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.

NOTE

> You can use R0 only to pass the FDB address. Any other use of R0 when you issue the WAIT$ macro will fail.

3.18 DELET$ - DELETE SPECIFIED FILE

The DELET$ macro causes the directory information for the file associated with the specified FDB to be deleted from the appropriate UFD. The space occupied by the file is then deallocated and returned for reallocation to the pool of available storage on the volume.

This macro 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 (including 0 and -1).

NOTE

> If the DELET$ macro is issued for use with a file containing sensitive information, it is recommended that you zero the file before closing it, or reformat the disk to destroy the sensitive information. (Although DELET$ logically removes a file, the information physically remains on the volume until written over with another file, and could be analyzed by unauthorized user tasks.)

3.18.1 Format of DELET$ Macro

Format

```
DELET$  fdb,err
```

Parameter

fdb

A symbolic value of the address of the associated FDB.

err

The symbolic address of an optional error-handling routine, which you coded.

The following examples show how the DELET$ macro may be used in a program.

```
     DELET$  R0

     DELET$  #OUTFDB,ERRLOC

     DELET$  R0,ERRLOC
```

CHAPTER 4

FILE CONTROL ROUTINES

This chapter describes a set of file control routines that you can invoke in MACRO-11 programs to perform the following functions:

- Read or write default directory string descriptors in program section $$FSR2.
- Read or write the default UIC word in program section $$FSR2.
- Read or write the default file protection word in program section $$FSR2.
- Read or write the file owner word in program section $$FSR2.
- Convert a directory string from ASCII to binary, or binary to ASCII.
- Fill in all or part of a filename block from a dataset descriptor or default filename block.
- 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.
- Truncate a file.
- Mark a temporary file for deletion.
- Delete a file by filename block.
- Perform device-specific control functions.

4.1 CALLING FILE CONTROL ROUTINES

The CALL macro invokes file control routines (JSR PC, dst). The Task Builder includes these routines from the system object library ([1,1]SYSLIB.OLB) at task-build time and incorporates them into your task. Your task calls the following file control routines:

```
CALL    .RDFDR

CALL    .EXTND
```

Before your task issues the CALL macro, certain file control routines require that specific registers be preset with requisite information. The descriptions of the respective routines identify these requirements. Upon return to your task, all task registers are preserved except for those that have been explicitly specified as changed.

If a file control routine detects an error, it sets the Carry bit indication to FDB offset location F.ERR. However, certain file control routines do not return error indications even if one is present. The following file control routines are listed according to whether they return error indications.

Normal Error Return (Carry bit and F.ERR)	**No Error Return**
.ASCPP	.RDFDR
.PARSE	.WDFDR
.PRSDV	.RDFUI
.PRSDI	.WDFUI
.PRSDV	.RDFFP
.ASLUN	.WDFFP
.FIND	.RFOWN
.ENTER	.WFOWN
.REMOV	.PPASC
.GTDIR	.MARK
.GTDID	
.POINT	
.POSRC	
.POSIT	
.XQIO	
.RENAM	
.EXTND	
.TRNCL	
.MRKDL	
.DLFNB	
.CTRL	

Appendix I lists the error codes that the routines listed in the opening of Chapter 4 return in FDB offset location F.ERR.

4.2 DEFAULT DIRECTORY STRING ROUTINES

The .RDFDR and .WDFDR routines read and write directory string descriptors.

4.2.1 .RDFDR - Read $$FSR2 Default Directory String Descriptor

Your task calls the .RDFDR routine to read default directory string descriptor words previously written by the .WDFDR routine into program section $$FSR2 of the FSR. These descriptor words define the address and the length of an ASCII string that contains the default directory string. This directory string is the default directory that FCS uses when one is not specified in a dataset descriptor.

If you have not established default directory string descriptor words in program section $$FSR2 by using the .WDFDR routine described in the following text, the descriptor words in program section $$FSR2 are null. FCS uses a default directory (when one is not specified in a dataset descriptor) corresponding to the User Identification Code (UIC) under which the task is running.

When called, the .RDFDR routine returns values in the following registers:

R1 contains the size (in bytes) of the default directory string "in program section $$FSR2.

R2 contains the address of the default directory string in program section $$FSR2. If no default directory string descriptor words have been written by .WDFDR, R2 equals 0.

4.2.2 .WDFDR - Write New $$FSR2 Default Directory String Descriptor

Your task calls the .WDFDR routine to create default directory string descriptor words in program section $$FSR2. For example, if your program is to operate on files in the directory [220,220], regardless of the UIC under which the program runs, you can establish default directory string descriptor cells in program section $$FSR2 to point to this alternate directory string [220,220] created elsewhere in the program. To do this, first create the desired directory string through an .ASCII directive. Then, by calling the .WDFDR routine, you can initialize the default directory string descriptor cells in program section $$FSR2 to point to the new directory string.

Assume that the task is currently running under default UIC [200,200]. You define a new directory string by issuing the following MACRO-11 directive:

```
NEWDDS:    .ASCII /[220,220]/
```

By calling the .WDFDR routine, you initialize string descriptor cells in program section $$FSR2 to point to the new directory string.

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

Establishing default directory string descriptor words in program section $$FSR2 does not change the default UIC in program section $$FSR2 or the task's privileges.

4.3 DEFAULT UIC ROUTINES

The .RDFUI and .WDFUI routines read and write the default UIC maintained in program section $$FSR2 of the file storage region (FSR). Unlike the default directory string descriptor that describes an ASCII string, the default UIC is maintained as a binary value with the following format:

```
Bit 15              8 7                0
    +----------------+-----------------+
    |     GROUP      |     MEMBER      |
    +----------------+-----------------+
```

The default UIC in program section $$FSR2 provides directory identification information for a file being accessed. FCS uses the default UIC only when all other sources of such information have failed to specify a directory (refer to Section 4.7.1.2). FCS never uses it to establish file ownership or file access privileges.

Unless you change the default UIC through the .WDFUI routine described in the following text, the default UIC in program section $$FSR2 always corresponds to the UIC under which the task is running.

4.3.1 .RDFUI - Read Default UIC

Your task calls the .RDFUI routine to return the default UIC as follows:

R1 contains the binary encoded default UIC as maintained in program section $$FSR2.

4.3.2 .WDFUI - Write Default UIC

Your task calls the .WDFUI routine to create a new default UIC in program section $$FSR2.

The following register must be preset before calling the .WDFUI routine:

R1 must contain the binary representation of a UIC.

NOTE

The .WDFUI routine overrides any default UIC descriptor in program section $$FSR2 that was previously created by the .WDFDR routine.

4.4 DEFAULT FILE PROTECTION WORD ROUTINES

The .RDFFP and .WDFFP routines described in the following text read and write the default file protection word in a location in program section $$FSR2 of the file storage region (FSR). FCS uses this word only when a file is created (for example, 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 -1, it indicates that the volume default file protection value is to be used for the new file.

The default file protection word has four file protection categories: world, group, owner, and system. It has the following format:

Bit	15 12	11 8	7 4	3 0
	WORLD	GROUP	OWNER	SYSTEM

Each of these four file protection categories has four bits; each bit represents the kind of access allowed to a file, as shown in the following example:

Bit	3	2	1	0
	DELETE	EXTEND	WRITE	READ

A bit value of 0 indicates that the corresponding file access is to be allowed; a bit value of 1 indicates that the access is to be denied.

4.4.1 .RDFFP - Read $$FSR2 Default File Protection Word

You call 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 program section $$FSR2.

4.4.2 .WDFFP - Write New $$FSR2 Default File Protection Word

You use the .WDFFP routine to write a new default file protection word into program section $$FSR2.

The following register must be preset before calling the .WDFFP routine:

R1 must contain the new default file protection word to be written into program section $$FSR2. If this register is set to -1, the default file protection values established through the appropriate operating system command will be used in creating all subsequent new files.

4.5 FILE OWNER WORD ROUTINES

The file owner word, like the default file protection word, is a location in program section $$FSR2 of the FSR. Its contents are specified by the current program through the .WFOWN routine. If not so specified, the file owner word contains 0.

For nonprivileged users, the owner of a new file corresponds to the default UIC specification, as follows:

- If the volume on which the new file is created is private (allocated), the owner UIC is the same as the UIC of the task creating the file.
- If the volume on which the new file is created is a system volume, the owner UIC is the same as the task's login UIC.

For privileged users, the owner UIC is always the same as the UIC of the task creating the file.

Note that for files created by privileged or nonprivileged tasks that are started by a time-scheduled request, the owner UIC is set to the UIC specified at task-build time.

A specific UIC value can be stored in the file owner word by the .WFOWN routine (see Section 4.5.2). All new files then created and closed by your task will contain the specified UIC value.

The format of the file owner word is shown next:

```
Bit 15        8 7         0
    ----------------------
    |  GROUP   |  MEMBER  |
    ----------------------
```

The routines for reading and writing the file owner word are described in Section 4.5.1.

NOTE

> The UIC and the file protection word for the file (see Section 4.4) must not be set such that the UIC under which the task is running does not have access to the file. This condition results in a privilege violation.

When a file is created, the owner UIC is always set to either the UIC of the task creating the file or the task's login UIC, as previously described. However, when closing the file, you can change the owner UIC by using the .WFOWN routine. If the file is not closed properly, the owner UIC will not change.

4.5.1 .RFOWN - Read $$FSR2 File Owner Word

You use the .RFOWN routine to read the contents of the file owner word in program section $$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). If the current program has not previously established the contents of the file owner word through the .WFOWN routine, R1 contains 0.

4.5.2 .WFOWN - Write New $$FSR2 File Owner Word

You use the .WFOWN routine to initialize the file owner word in program section $$FSR2.

The following register must be preset before calling this routine:

R1 must contain a file owner word to be written into $$FSR2.

4.6 ASCII/BINARY UIC CONVERSION ROUTINES

Your task calls the .ASCPP and .PPASC routines to convert a directory string from ASCII to binary, or binary to ASCII.

4.6.1 .ASCPP - Convert ASCII Directory String to Equivalent Binary UIC

Your task calls the .ASCPP routine 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 your program (see Section 2.4.1) for the string to be converted.

R3 must contain the address of a word location in your 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.

4.6.2 .PPASC - Convert UIC to ASCII Directory String

Your task calls the .PPASC routine 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 your program into which to place the ASCII string. The resultant string can be up to nine bytes in length, for example, [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 (for example, 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 (square brackets and commas) in the directory string (for example, [10,20]). Bit 1 is set to 1 to suppress separators (for example, 1020).

The .PPASC routine adds to the contents of R2, allowing R2 to point to the byte immediately following the last byte in the converted directory string.

4.7 FILENAME BLOCK ROUTINES

FCS provides the .PARSE, .PRSDV, .PRSDI, .PRSFN, and .ASLUN routines, which perform functions related to a specified filename block. These routines are described in the following sections. FCS provides the main support for logical name translation in these routines.

4.7.1 Logical Name Translation

When FCS obtains a device name or a file specification, it examines the leftmost component to check for the presence of a logical name. It does this by checking whether the leftmost character string, which may consist only of alphanumeric characters, dollar signs ($), or underscores (_), ends in either a colon (:) or a space ( ). If it does, FCS recognizes the character string as a logical name and makes an attempt to translate the entire character string. If FCS finds an equivalence name, the original information that the logical name represents, the equivalence name is merged with the file specification without the logical name. If the leftmost character string ends in any character other than a space or colon, FCS uses the string as a file specification.

In the following example, the first line shows how to specify a file specification. The second line shows the use of a logical name to specify the location of a file.

```
     $ TYPE ALPHA
     $ TYPE DISK:ALPHA
```

In the first line, FCS obtains the file specification ALPHA and checks to see if ALPHA is a logical name because ALPHA is the leftmost (and in this example, the only) component of the file specification. In the second line, FCS checks to see if DISK, the leftmost component, is a logical name. FCS does not check ALPHA.

The system stores logical names in four logical name tables. When FCS translates logical names, it searches first the task table, then the session table, then the group table, and finally the system table, and uses the first match it finds.

4.7.1.1 **Iterative Translation** - When FCS translates logical names in file specifications, the logical name translation can be iterative. That is, after FCS translates a logical name in the file specification, it continues to translate and repeats the process of translating the file specification if it finds the presence of a logical name.

NOTE

> Use the ASSIGN command for all file-specification arguments. ASSIGN performs several checks to insure that the logical names are syntactically correct.

As an example of iterative translation, consider logical name table entries made with ASSIGN commands as follows:

```
     $ ASSIGN DB1: DISK
     $ ASSIGN REPORT DISK:[HUMIDO]WEATHER.SUM
```

The first ASSIGN command equates the logical name DISK to the device DB1:. The second ASSIGN command equates the logical name REPORT to the file specification DISK:[HUMIDO]WEATHER.SUM. In subsequent commands, or in programs you execute, you can refer to the logical name REPORT. For example,

```
     $ TYPE REPORT
```

When FCS attempts to translate the logical name REPORT, it checks the logical name tables and finds the equivalence name DISK:[HUMIDO]WEATHER.SUM. It then checks the file specification for the presence of a logical name; if it finds a match in one of the logical name tables (DISK in this example), it translates that logical name also. When the logical name translation is complete, the following translated file specification results:

```
DB1:[HUMIDO]WEATHER.SUM
```

FCS limits logical name translation to ten levels. If you define a logical name to more than ten levels or create a circular definition, an error occurs when you use the logical name.

If a device name or file specification is preceded by an underscore (_), the logical translation process stops. In the example,

```
TYPE _REPORT
```

FCS would look for the file REPORT. in your default device and directory. Note also that if an extension or version are included with the file name, it is not considered for logical translation. In the example,

```
TYPE REPORT.TXT
```

FCS would look for the file REPORT.TXT in your default device and directory.

4.7.1.2 **Logical Translation Process** - The following three basic operations occur during the logical translation process:

1. **Parsing the file specification** - FCS determines the location and length of the various parts of the primary file specification. It sets flags indicating which parts were present and which parts contain wild cards. The length and location of the string following the file specification is also determined.

2. **Expanding the string** - FCS takes the first portion of the string and attempts to translate it. The first portion could be a logical name. A logical name can be a device name or a file name. If there is a logical name string to equivalent string translation, FCS uses the equivalent string as the secondary string in a merge. The primary string for this merge is the original file specification less the logical name. This operation is reiterative until one of the following occurs:

   - There is no logical name to translate.
   - The translation fails.
   - The result of the translation is not a file specification.
   - The recursion limit of more than ten levels of translation has occurred.
   - The first character is an underscore (_).

3. **Merging the specifications** - FCS generates a merged string from the primary and the default string. The merged string consists of the primary string with any missing components filled in from the default string.

4.7.2 .PARSE - Fill in All File Name Information

FCS includes the main support for logical name translation in the FCS .PARSE directive. When the .PARSE routine receives a string, the .PARSE routine performs any necessary logical expansion and parses the resultant string.

If you use logical name parsing, logical translation is performed first. If you have not requested logical name translation (as when it has already been done by CSI$4), the FL.AEX bit should be set in the F.FLG byte in the FDB. Setting this bit disables logical name translation.

When called, the .PARSE routine first zeros the filename block pointed to by R1 and then stores the following information in the filename block:

- The ASCII device name (N.DVNM)
- The binary unit number (N.UNIT)
- The directory ID (N.DID)
- The Radix-50 file name (N.FNAM)
- The Radix-50 file type or extension (N.FTYP)
- The binary file version number (N.FVER)

For ANSI magnetic tape file names, the following information is stored in the filename block:

- The ASCII device name (N.DVNM)
- The binary unit number (N.UNIT)
- The file name as 17 ASCII bytes (N.ANM1 and N.ANM2)
- The binary file version number (N.FVER)

In addition, the .PARSE routine calls the .ASLUN routine to assign the LUN associated with the FDB to the device and unit currently specified in the filename block.

Both formats for filename blocks are shown in detail in Appendix B.

Before the .PARSE routine can be called, the FINIT$ macro (see Section 2.6) must be invoked explicitly in your program, or it must be invoked implicitly through a prior OPEN$x macro call. Note, however, that your task can issue the FINIT$ call only once in the initialization section of the program; that is, 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.

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 (that is, R0 + F.FNB).

R2 must contain the address of the desired dataset descriptor if .PARSE is to access a dataset descriptor in filling in the specified filename block. This structure is usually, but not necessarily, the same as that associated with the FDB specified in R0 (that is, the dataset descriptor pointed to by the address value in F.DSPT).

If R2 contains 0, a dataset descriptor has not been defined; therefore, the dataset descriptor logic of the .PARSE routine is bypassed.

R3 must contain the address of the desired default filename block for the .PARSE routine to access a default filename block in filling in the specified filename block. This default filename block is usually, but not necessarily, the same as the one associated with the FDB specified in R0 (that is, the default filename block pointed to by the address value in F.DFNB).

If R3 contains 0, a default filename block has not been defined; therefore, the default filename block logic of the .PARSE routine 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 filling in the specified filename block.

The .PARSE routine fills in the specified filename block in the order described in the following sections.

4.7.2.1 **Device and Unit Information** - The .PARSE routine first tries to fill in the filename block with device (N.DVNM) and unit (N.UNIT) information. The following operations are performed until the required information is obtained from the specified data structures:

1. If the address of a dataset descriptor is specified in R2 and the dataset descriptor contains a device string, the .PARSE routine moves the device and unit information from the dataset descriptor into the specified filename block.

2. If step 1 fails, and if the address of a default filename block is specified in R3, and the default filename block contains a nonzero value in the device name field, the .PARSE routine moves the device and unit information from the device name field into the specified filename block.

3. If step 2 fails, the .PARSE routine uses the device and unit currently assigned to the logical unit number in offset location F.LUN of the specified FDB to fill in the filename block.

   This feature allows a program to use preassigned logical units that are assigned through either the device assignment (ASG) option of the Task Builder or one of the following commands: ASSIGN in DCL or ASN in MCR. In this case, you simply avoid 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, the .PARSE routine assigns this number to the system device (SY0:).

The .PARSE routine first determines the device and unit, assigns the logical unit number, and then invokes the GLUN$ directive to obtain necessary device information. The required information obtained by GLUN$ is placed by the .PARSE routine into the following offsets in the filename block pointed to by R1:

N.DVNM - Device Name Field, which contains the redirected device name.

N.UNIT - Unit Number Field, which contains the redirected unit number.

Additionally, the .PARSE routine places the information returned by GLUN$ into the following offsets in the FDB, which R0 points to:

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, you 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) of the device associated with the assigned LUN.

The GLUN$ directive is described in detail in the RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual.

4.7.2.2 Directory Identification Information

4.7.2.2 Directory Identification Information - The N.DID field in the filename block contains the following information:

Word	**Meaning**
1	File ID
2	File Sequence Number
3	Reserved

The .PARSE routine moves these three words from the Master File Directory (MFD) to the N.DID field in the filename block. The file ID is the header number of the header (in the index file) for a User File Directory. The .FIND routine uses the file ID to locate and search a UFD and fill in the N.FID field in the filename block. The N.FID has the same format as the N.DID field except that it identifies the header number of the header for a user data file. The file sequence number is incremented each time a file header is reused for a new file.

Following the operations described in the preceding section, .PARSE attempts to fill in the filename block with directory identification (N.DID) information. The methods for obtaining this information are as follows:

1. If your task specifies the address of a dataset descriptor in R2 and the dataset descriptor contains a directory string, FCS uses that directory string to find the associated UFD in the MFD. The resulting file ID is then moved into the directory-ID field of the specified filename block.

2. If step 1 fails, and your task specifies the address of a default filename block in R3, and the default filename block contains a nonzero directory ID, the contents of the default filename block are moved into the specified filename block.

   Because 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, your task must initialize these cells manually. Or your task can call the .GTDIR routine (see Section 4.9.1) or the .GTDID routine (see Section 4.9.2). Note that these routines can also initialize a specified filename block directly with required directory information.

3. If neither step 1 nor step 2 yields the required directory string, the .PARSE routine examines the default directory string words in program section $$FSR2. If your program has previously initialized these words through use of the .WDFDR routine, FCS uses the string described as the default directory.

4. If steps 1 through 3 fail to produce directory information, FCS uses the binary value stored in the default UIC word in program section $$FSR2 as the directory identifier. Unless changed by you through the .WDFUI routine, this word contains the UIC under which the task is running.

NOTE

The .PARSE routine does not accept UICs that contain wildcards. Additionally, the .PARSE routine does not set filename block status word (N.STAT) bits NB.SD1 or NB.SD2 (group and owner wildcard specifications, respectively).

4.7.2.3 **File Name, File Type, and File Version Information** - After completing the operations described in the preceding section, the .PARSE routine attempts to obtain file name information (N.FNAM, N.FTYP, and N.FVER), as follows:

1. If your task specifies the address of a dataset descriptor in R2 and this structure contains a filename string, the file name information therein is moved into the specified filename block.

2. If your task specifies the address of a default filename block in R3, and one or more of the file name, file type, and file version number fields of the dataset descriptor that you specified in R2 are null, the corresponding fields of the default filename block fill in the specified filename block.

3. If neither step 1 nor step 2 yields the requisite file name information, any specific fields not available from either source remain null.

NOTE

If a period (.) appears in the filename string without an accompanying file type designation (for example, TEST. or TEST.;3), FCS interprets the file type 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 (for example, TEST.DAT;), FCS also interprets the file version number as being null; again, the default file version number is not used. This information concerning semicolons in filename strings does not apply to the 17-byte ASCII filename strings supported for ANSI magnetic tape.

4.7.2.4 **Using the FDB Extension for Logical Names** - FCS uses the FDB extension to obtain the correct directory string. The extension has the following format:

```
    .BYTE     Extension length
    .BYTE     Unused
    .BYTE     Length of the directory string buffer
    .BYTE     Length of the directory string (filled in by .PRSDI)
    .WORD     Address of the directory string buffer
```

The FDB extension block and the directory string buffer are allocated in your task's address space. You fill in the address, the length of the buffer, and the length of the extension into the appropriate locations in the FDB extension block. You then place the address of the extension block in the offset F.EXT in the FDB. When the directory parsing code detects that F.EXT has a value, it uses the value as an address and moves the directory string into the buffer. It also puts the length of the actual directory string into the appropriate byte of the extension. This directory string is always filled in unless FCS obtains the directory from the default name block, because the default name block does not contain the directory string. If FCS obtains the directory from the default name block, FCS sets the directory length to zero.

4.7.2.5 **Other Filename Block Information** - After performing all the previously described operations, the .PARSE routine also fills in the 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 that in this table the directory, device, file name, file type, or file version number specification pertains to ASCII data supplied through the dataset descriptor pointed to by R2.

In addition, the .PARSE routine zeros offset location N.NEXT in the filename block pointed to by R1. This action has implications for wildcard operations, as described in Section 4.8.1.

4.7.2.6 **.EXPLG Module (Expand Logical)** - The .EXPLG module expands a logical name and returns a pointer to the task that points to the expanded string. The module has the following inputs and outputs:

INPUTS:	R2 - Pointer to the dataset descriptor of the string to be expanded.
OUTPUTS:	R2 - Pointer to the dataset descriptor of the expanded string. All other registers are preserved.

This routine expands the string into the same buffer that the .PARSE routine and CSI$4 use for input files; therefore, caution is advised if you use this method. In addition, the call only accepts logical names that expand into a correct FCS file specification. The inclusion of a node specifier or other non-FCS characters results in an error being returned.

4.7.3 .PRSDV - Fill in Device and Unit Information Only

The .PRSDV routine is identical to the .PARSE routine, except that it performs only those operations associated with requisite device and unit information (see Section 4.7.1.1). The .PRSDV routine zeros the filename block pointed to by R1, calls the .PARSE routine to operate on the device and unit fields in the specified dataset descriptor or default filename block, and assigns the logical unit number contained in offset location F.LUN of the specified FDB.

After the logical device translation is performed, .PRSDV fills the filename block with the required device and unit information. If the device is LB:, the actual physical device name and unit are placed in the filename block. If the logical device expands to contain anything other than a device specification, for example, a directory or a filename, the remainder is ignored. Setting the FL.AEX bit (see Chapter 6) disables logical expansion for the device and unit information.

4.7.4 .PRSDI - Fill in Directory Identification Information Only

The .PRSDI routine is identical to the .PARSE routine, except that it performs only those operations associated with requisite directory identification information (see Section 4.7.1.2). The .PRSDI routine performs a .PARSE operation on the directory identification information (N.DID) field in the specified dataset descriptor or default filename block. The .PRSDI routine does not perform any logical name expansion.

4.7.5 .PRSFN - Fill in File Name, File Type, and File Version Only

The .PRSFN routine is identical to the .PARSE routine, except that it performs only those operations associated with requisite file name, file type, and file version information (see Section 4.7.1.3). This routine performs a .PARSE operation on the file name, file type, and file version information fields (N.FNAM, N.FTYP, N.FVER) in the specified dataset descriptor or default filename block. The .PRSFN routine does not perform any logical name expansion.

4.7.6 .ASLUN - Assign Logical Unit Number

The .ASLUN routine assigns a logical unit number to a specified device and unit and returns 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 where the desired device and unit information are located. This filename block is usually, but not necessarily, within the FDB specified by the address in R0.

If the device name field (offset location N.DVNM) of the filename block pointed to by R1 contains a nonzero 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 offset location N.DVNM in the filename block contains 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 specific device, the .ASLUN routine assigns it to the system device (SY0:) by default.

The information returned to the specified filename block and the specified FDB is identical to that returned by the device and unit logic of the .PARSE routine (see Section 4.7.1.1).

4.8 DIRECTORY ENTRY ROUTINES

The .FIND, .ENTER, and .REMOV routines find, insert, and delete directory entries. The term "directory entry" refers to entries in both the Master File Directory (MFD) and the User File Directory (UFD).

4.8.1 .FIND - Locate Directory Entry

You call the .FIND routine to locate a directory entry by file name and 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, within the FDB specified by the address 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 file name, file type, and file version number. 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 file name, 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 matches that corresponding field. Thus, if the file name and file version number fields contain wildcard specifications (*), and the file type field is specified as .OBJ (that is, *.OBJ;*), the first directory entry encountered that contains .OBJ in the file type field matches.

When a wildcard match is found, the complete file name, file type, and file version number fields of the matching entry are returned to the filename block, along with the file-ID field (N.FID). 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. FCS uses this information in subsequent .FIND operations to locate the next matching entry in the directory file.

For example, the .FIND routine often opens 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.

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 in the following text tests 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$  R0
```

   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 directory file search for a matching entry.

4. Perform the desired operations on the file.

NOTE

This procedure applies only to the following types of wildcard file specifications:

```
TEST.DAT;*
TEST.*;*
*.DAT;*
TEST.*;5
*.DAT;3
```

This 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 file name 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. In the latter case, however, the version number cannot be 0, for the latest version of the file, or -1, for the oldest version of the file.

When your task sets NB.ANS, the .FIND operation compares the file name against the full 17-character ANSI filename string that is stored in the filename block (see Appendix B). When NB.ANS is clear, the file name is converted to Radix-50, as described in Appendix G.

ANSI magnetic tape file names in the following formats can be converted to Radix-50:

- Up to nine Radix-50 characters followed by spaces
- Up to nine Radix-50 characters followed by a period, followed by spaces or by a 3-character file type

Note that unless NB.ANS is set before the call to .FIND, some file names may be incorrectly matched. For example, the names "ABC" and "ABC." are considered the same when compared with the name ABC in Radix-50.

When a wildcard operation is performed, the name returned in the filename block is normally converted to Radix-50. However, if NB.ANS is set, the ANSI filename string is returned as up-to-17 ASCII bytes. The first twelve bytes are returned at offset N.ANM1 in the ANSI filename block. The remainder are returned at offset N.ANM2.

It is incorrect to set NB.ANS before a wildcard .FIND operation unless both file name and file type are wild, or neither filename nor file type are wild.

To delete a file whose file descriptor entry in the FDB contains wildcards, you must save the values in the fields N.STAT and N.NEXT in the FDB, then zero these fields in the FDB. A DELETE call then uses the information returned from the last .FIND to delete the file. Once the file is deleted, the saved values of N.STAT and N.NEXT must be restored in the FDB.

4.8.2 .ENTER - Insert Directory Entry

You use the .ENTER routine to insert an entry by file name 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 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 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.

4.8.3 .REMOV - Delete Directory Entry

You use the .REMOV routine to delete an entry from a directory by file name. 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 those defined for the .FIND routine described in Section 4.8.1. The file version number for .REMOV operations must be an explicit number (including 0 and -1) or a wildcard. Each .REMOV operation deletes the next directory entry that has the specified file name, file type, and file version number.

4.9 FILENAME BLOCK ROUTINES

The .GTDIR and .GTDID routines insert directory information in a specified filename block. Sections 4.9.1 and 4.9.2, following, describe the use and operation of these routines.

4.9.1 .GTDIR - Insert Directory Information in Filename Block

You call the .GTDIR routine to insert directory information 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 the filename block in which the directory information is to be placed. This filename block is usually, but not necessarily, within the FDB specified by the address in R0.

R2 must contain the address of the 2-word directory string descriptor in your program. This string descriptor defines the size and the address of the desired directory string.

This routine performs a .FIND operation for the specified UFD in the 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 clears the rest of the filename block.

You can also use the .GTDIR routine with the NMBLK$ macro call (see Section 2.4.2) to insert directory information into a specified default filename block.

4.9.2 .GTDID - Insert Default Directory Information in Filename Block

The .GTDID routine provides an alternative means for inserting directory information into a specified filename block. Instead of allowing the specification of the directory string, as does the .GTDIR routine, this routine uses the binary value found in the default UIC word maintained in program section $$FSR2 as the desired UFD.

Before calling the .GTDID routine, you must preset the following registers:

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, within the FDB specified by the address in R0.

When called, the .GTDID routine takes the default UIC from its 1-word location in program section $$FSR2 and performs a .FIND operation for the associated UFD in the MFD. The resulting directory ID is returned to the three words of the specified filename block, starting at offset location N.DID. As does the .GTDIR routine, .GTDID preserves offset locations N.FNAM, N.FTYP, N.FVER, N.DVNM, and N.UNIT in the filename block, but clears the rest of the filename block.

The .GTDID routine uses considerably less code than the .GTDIR routine. Its input is the binary representation of a UIC rather than an ASCII string descriptor. Therefore, it does not invoke the .PARSE logic; furthermore, .GTDID is specifically for use in programs that open files by the OFNB$ macro call (see Section 3.6). Such a program does not invoke the .PARSE logic because all necessary file name information is provided to the program in filename block format.

Like the .GTDIR routine described in Section 4.9.1, the .GTDID routine can be used with the NMBLK$ macro call (see Section 2.4.2) to insert directory information (N.DID) into a specified default filename block. You also have the option to initialize offset location N.DID manually with the required directory information.

The .GTDID routine returns file-ID 177777,177777,0 for nondirectory devices such as terminals.

4.10 FILE POINTER ROUTINES

The .POINT, .POSRC, .MARK, and .POSIT routines point to a byte or a record within a specified file. Sections 4.10.1 through 4.10.3, following, briefly describe the use of these routines and their operation.

4.10.1 .POINT - Position File to Specified Byte

You call the .POINT routine to position a file pointer 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.

NOTE

> Use of the .POINT routine is restricted to files accessed with GET$ or PUT$ macros. For files accessed with READ$ or WRITE$ macros, use the FDBK$R macro to initialize the block access section of the FDB.

The .POINT routine is used often with the .MARK routine and achieves a limited degree of random access with variable-length records. The .MARK routine saves the positional information of a file, permitting you to temporarily close that file and reopen it later at the same position; this procedure is outlined in the following steps:

1. Call the .MARK routine to save the current positional context of the file.
2. Close the file.
3. Reopen the file when desired.
4. Load the information returned by the .MARK routine into R1, R2, and R3.
5. Call the .POINT routine. The .POINT routine may be called to rewind a file on disk or ANSII magnetic tape to its start. For this case, R1 and R3 must be set to 0, and R2 must be set to 1. The .POINT routine may also be called to rewind a file that is open on a terminal. Doing so clears the terminal end-of-file condition.
6. Resume processing of the file.

4.10.2 .POSRC - Position File to Specified Record

The .POSRC routine sets up the position information for 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, you 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.

The register R0 must be preset before calling the .POSRC routine:

R0 must contain the address of the associated FDB.

You use the .POSRC routine when performing random access PUT$ operations in locate mode. Normally, PUT$ operations in locate mode are sequential; however, when you use random access mode, you must follow the next procedure 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.

4.10.3 .MARK - Save Position Information Context of File

The .MARK routine allows you to save the current position information of a file for later use; you can save the current position information of a file, close that file, and later reopen it to the same position. The .MARK routine also allows you to alter records within a file; you can save the file position, retrieve information elsewhere in that file, and return to the saved position of the file to alter the desired record. This sequence may be repeated to update any number of desired records in the file. The following register must be preset before calling the .MARK routine:

R0 must contain the address of the associated FDB before calling this routine.

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.

4.10.4 .POSIT - Return Specified Record Position Information

The .POSIT routine calculates the virtual block number and the byte number locating 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, which sets up the position information of the file to the specified record, .POSIT calculates the positional information of a specified record so that a .POINT operation can be later performed to position to the desired record.

The .POSIT routine returns register values identical to those described previously for the .MARK routine.

4.11 QUEUE I/O FUNCTION ROUTINE (.XQIO)

The Queue I/O Function Routine (.XQIO) executes a specified QIO$ function and waits 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 QIO$ macro function code. Refer to the RSX-11M/M-PLUS I/O Drivers Reference Manual or Micro/RSX I/O Drivers Reference Manual for the desired QIO$ macro function codes.

R2 must contain the number of optional parameters, if any, to be included in the QIO$ directive.

R3 must contain the beginning address of the list of optional QIO$ directive parameters, if R2 contains a nonzero value. Refer to the RSX-11M/M-PLUS I/O Drivers Reference Manual or Micro/RSX Drivers Reference Manual for the parameter list.

4.12 RENAME FILE ROUTINE (.RENAM)

The .RENAM routine is called to change the name of a file in its associated directory. To rename a file, you must specify the address of an FDB containing file name information, a LUN, and an event flag number.

If the file to be renamed is open when the call to .RENAM is issued, that file is closed before the renaming operation is attempted.

The following registers must be preset before calling this routine:

R0 must contain the address of the FDB associated with the file with the original name.

R1 must contain the address of the FDB containing the desired file name information, LUN assignment, and event flag.

If the renaming operation is successful, a new directory entry is created and the original entry is deleted. If the operation is unsuccessful, the file is closed under its original name, and the associated directory is not affected.

The .RENAM routine uses the absence of a value in location F.FNB + N.FID to indicate that .PARSE must be called to parse a file specification (an open file always has a nonzero value in F.FNB + N.FID). If neither a dataset descriptor nor a default filename block is present, .PARSE returns a null file name. The rename operation then produces a new file name of version ".;1". If a wildcard (*) is part of the input file specification, wildcard processing like that described for the .FIND routine occurs. Wildcards are not allowed in an output file specification.

NOTE

The renaming process is merely a directory operation that replaces an old entry with a new entry. The file name stored in the file header block is not altered.

4.13 FILE EXTENSION ROUTINE (.EXTND)

The .EXTND routine extends either contiguous or noncontiguous files. The file to be extended can be either open or closed. A call to the .EXTND routine disables file truncation. You must explicitly call .TRNCL to truncate a file after you call .EXTND.

The following registers must be preset before calling the .EXTEND 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 with the 16 bits of R1 to define the number of blocks to be added to the file (see NOTE 1, which follows).

NOTE

1. FCS uses the contents of R1 and the high-order byte of R2 (Bits 8 through 15) to perform 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.

2. If a file previously had space allocated to it, a contiguous file extension by the .EXTND routine cannot be done. You can create a contiguous file by opening a new file with a zero allocation and calling .EXTND to allocate the desired number of blocks.

3. When writing a new file using QIO$ macros, the task must explicitly issue .EXTND calls as necessary to reserve enough blocks for the file, or the file must be initially created with sufficient blocks. In addition, the task must put an appropriate value in the FDB for the end-of-file block number (F.EFBK) before closing the file or rewinding and reading it.

4. If R2 contains a zero, FCS defaults to noncontiguous allocation.

In general, when FCS implicitly extends a file, it activates file truncation. See Section 4.14 for information on how to turn off file truncation. When your program explicitly allocates space to a file, either with an OPEN$ or .EXTND, FCS turns off truncation. To turn off file truncation and close the file, call the following routines:

1. Call the .EXTND routine. Set both R1 and R2 to 0.

2. Issue the CLOSE$ macro.

4.14 FILE TRUNCATION ROUTINE (.TRNCL)

The .TRNCL routine truncates a file to the logical end of the file, deallocates any space beyond this point, and closes the file.

The following register must be preset before calling this routine:

R0 must contain the address of the associated FDB.

The file must have been opened with both write and extend access privileges. Otherwise, the truncation will fail.

The close operation will be attempted even if the truncation operation fails. If errors occur in both operations, the error code from the close operation will be returned.

FCS turns on truncation when it extends a file. However, when your program explicitly calls the .EXTND routine, FCS turns off truncation.

4.15 FILE DELETION ROUTINES

FCS provides the .MRKDL and .DLFNB routines for deleting files.

NOTE

If you use the .MRKDL or .DLFNB routine to delete a file containing sensitive information, you should clear the file before closing it or reformat the disk to destroy the sensitive information. (Although the file is marked for deletion, the information physically remains on the volume until written over with another file, and could be analyzed by unauthorized users.)

4.15.1 .MRKDL - Mark Temporary File for Deletion

You use the .MRKDL routine to mark a temporary file for deletion; that is, 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 when the file is closed.

Table 4-1
R2 Control Bits for .EXTND Routine

Value in Low-Order Byte of R2	Meaning
0	Indicates that the file extent is to be noncontiguous.
200	Indicates that the file extent is to be noncontiguous. This clears the contiguous file attribute.
201	Indicates that the contiguous area is to be added to the file. This clears the contiguous file attribute.
203	Indicates that the largest available contiguous area is to be added to the file if the desired file extent space is not available. This clears the contiguous file attribute.
205	Indicates that this is the initial extent of the file. The file is to be contiguous.
207	Indicates that the largest contiguous area up to the specified extend size is to be added to the file. The file is to be contiguous.
210	Indicates that the file is to be extended by the default extend size for the volume. The extend is to be noncontiguous.
211	Indicates that the file is to be extended by the default extend size for the volume. The extend is to be contiguous, whereas the file is to be noncontiguous.

Before calling the .MRKDL routine, you must preset the following register:

R0 must contain the address of the associated FDB. This FDB is assumed to contain the file identification, device name, and unit information of the file to be deleted.

If the .MRKDL routine is invoked while the temporary file is open, as is normally done, the file is deleted unconditionally when it is closed. This occurs even if the calling task terminates abnormally without closing the file.

4.15.2 .DLFNB - Delete File by Filename Block

You use this routine to delete a file by filename block. The .DLFNB routine assumes that the filename block is completely filled; 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 DELET$ macro call (see Section 3.18), but .DLFNB does not require any of the .PARSE logic and thus requires less memory than the normal DELET$ function.

Like the DELET$ operation, however, the .DLFNB operation fails if the file to be deleted is not open, and if an explicit file version number is missing from offset location N.FVER of the associated filename block.

4.16 DEVICE CONTROL ROUTINE (.CTRL)

You call the .CTRL routine to perform device-specific control functions. The following are examples of .CTRL device-specific functions:

- Rewind a magnetic tape volume set.
- Position to the logical end of a magnetic tape volume set.
- Close the current magnetic tape volume and continue file operations on the next volume.
- Space forward or backward n blocks on a magnetic tape.
- Rewind a file on a magnetic tape or a terminal (record-oriented device).
- Clear the terminal end-of-file.

The following registers must be preset before calling this routine to perform the first three bulleted items listed previously in this section.

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 must be set to 0.

R3 must be set to 0.

When using .CTRL to space forward or backward, you must ensure that registers R0, R1, R2, and R3 contain the following values:

R0 must contain the address of the associated FDB.

R1 must contain the value FF.SPC.

R2 must contain the number of blocks to space forward or backward. A positive number means space forward; a negative number means space backward.

R3 must contain 0.

When using .CTRL to rewind a file, you must ensure that register R1 contains the value FF.RWF and that registers R2 and R3 contain 0.

See Chapter 5 for an explanation of using .CTRL to accomplish magnetic tape device-specific functions.

4.17 BUFFER FLUSH ROUTINE (.FLUSH)

The buffer flush routine (.FLUSH) writes the block buffer to the file being written in record mode. The .FLUSH routine also writes file attributes (including F.EFBK and F.HIBK, the end-of-file and high-allocation block numbers) each time the routine is called.

Closing the file guarantees that the block buffer is flushed and that the file attributes will be written back to the file header. However, closing and opening a file frequently, solely to write the block buffer, causes high system overhead and unnecessary disk accesses.

4.17.1 Purpose of the .FLUSH Routine

When FCS executes a PUT$ macro to a disk file, the PUT$ macro puts a record into the block buffer. When the block buffer is full, or the file is closed, FCS writes the block buffer to the file. You cannot predict when FCS will actually write the block buffer to the file.

Some applications may require that a record be written to a file immediately. As an example, a task that handles a laboratory device may write small amounts of data to a file every few minutes. If the system crashes, the contents of the block buffer may not have been written to the file. This data may be lost unless a PUT$ is immediately followed by a call to the .FLUSH routine. As another example, the .FLUSH routine should be called by an originating task to write data immediately if another task must then read data written by that originating task. In these examples, the tasks need not close the file to ensure that the data is written to the file.

4.17.2 When .FLUSH Should Be Used

Your task should call .FLUSH whenever data should be immediately written to a file.

You need not call the .FLUSH routine for block mode (WRITE$) or record mode (PUT$) write operations to a record-oriented device; the block buffer is always written in these cases. Nothing happens if you call .FLUSH when a file is open under these circumstances except the return of a cleared Carry bit and status +1 (success) in FDB byte F.ERR.

4.17.3 Performance Considerations Using .FLUSH

Calling the .FLUSH routine after every PUT$ macro can greatly increase I/O activity compared to using solely the PUT$ macro. One alternative is to call the .FLUSH routine after certain intervals have passed or after a certain number of calls to PUT$.

4.17.4 Using the .FLUSH Routine

The following register must be preset before calling this routine:

R0 must contain the address of the associated FDB.

During output, all registers are preserved, the C-bit is clear or set to indicate success or failure, and the FDB F.ERR byte contains the success or failure code.

CHAPTER 5

FILE STRUCTURES

This chapter describes the structure of files supported by the RSX-11M systems. It specifically covers the identical file structure on disk, DECtape, and DECtape II and the ANSI file structure on magnetic tape supported by the Micro/RSX and RSX-11M/M-PLUS systems.

The disk, DECtape, and DECtape II file structure is called Files-11; the magnetic tape file structure is ANSI standard.

The Files-11 structure is a file-organization system, which primarily determines the way that files and their associated control files are arranged on a disk or DECtape. Associated with the Files-11 structure is a system of data structures in memory, the most important of which include the Volume Control Block, the File Control Block, and the Device Control Block. Files-11 structure includes not only the physical file and its associated control files, but the necessary information in these files that determines the file's size, location, content, and various attributes.

The ANSI standard describes a way of organizing sequential files on a magnetic tape that allows the tape to be used on any computer system. The standard includes file structure, labeling, and physical characteristics such as end-of-tape length.

5.1 DISK AND DECTAPE FILE STRUCTURE (FILES-11)

Disk and DECtape volumes (defined by and associated with a Volume Control Block) contain both user files and system files. Disks and DECtapes initialized through the DCL command INITIALIZE or MCR command INITVOLUME have the standard Files-11 structure built for them. The standard system files created by these commands include:

- Index file
- Storage allocation file
- Bad block file
- Master File Directory (MFD)
- Checkpoint file

Each Files-11 volume has all of these files. A volume may have more than one directory file; the system uses these files, created by the MCR UFD command or the DCL CREATE/DIRECTORY command for RSX-11M/M-PLUS and Micro/RSX systems, to locate user files on the volume.

5.1.1 User File Structure

Data files on disk and DECtape consist of ordered sets of virtual blocks; these blocks constitute the virtual structure of the data files as they appear to you. Virtual blocks can be read and written directly by issuing READ$ and WRITE$ macro calls (see Sections 3.15 and 3.16, respectively). The first block in the file is virtual block 1; subsequent virtual blocks are numbered in ascending order.

The virtual blocks of a file are stored on the volume as logical blocks. Because virtual blocks and logical blocks are equal in size, and the logical block size of all volumes is 256 words, 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 file name and its associated file number and file sequence number. The number of required directory files depends on the number of users of the volume. For single-user volumes, only a Master File Directory (MFD) is needed; for multiuser volumes, an MFD is required, and one User File Directory (UFD) is required for each user of the volume.

The MFD contains a list of all the UFDs on the volume, and each UFD contains a list of all that user's files. UFDs are identified by User Identification Codes (UICs). An MCR UFD command or a DCL CREATE/DIRECTORY command creates the User File Directory. These commands are described in detail in the _RSX-11M/M-PLUS MCR Operations Manual_ and the _RSX-11M/M-PLUS Command Language Manual._

Figures 5-1 and 5-2 illustrate the directory structure for single-user and multiuser volumes, respectively.

5.1.3 Index File

You create the index file for the operating system to use when you initialize a volume. During initialization, the information required by the system to manage the file is placed in the index file. The index file contains volume information and user file header blocks, which the system file control primitives (FCP) use to manage the file. The file header blocks (see Section 5.1.4) are stored in the index file so that they can be located quickly. Furthermore, because a file header block is 256 words in length, it can be read into memory with a single access. Appendix E contains a detailed description of the format and content of an index file.

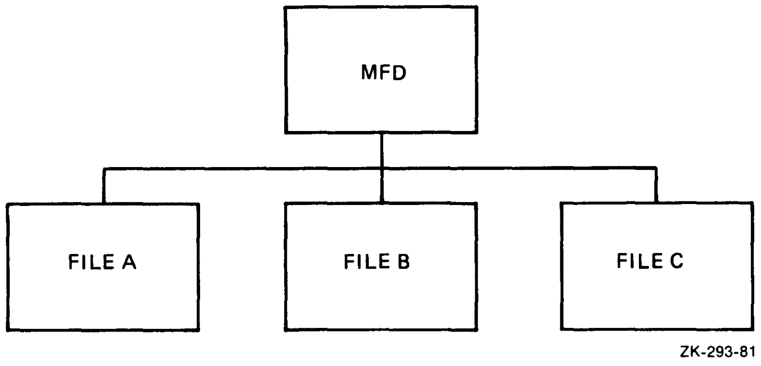


Figure 5-1 Directory Structure for Single-User Volumes

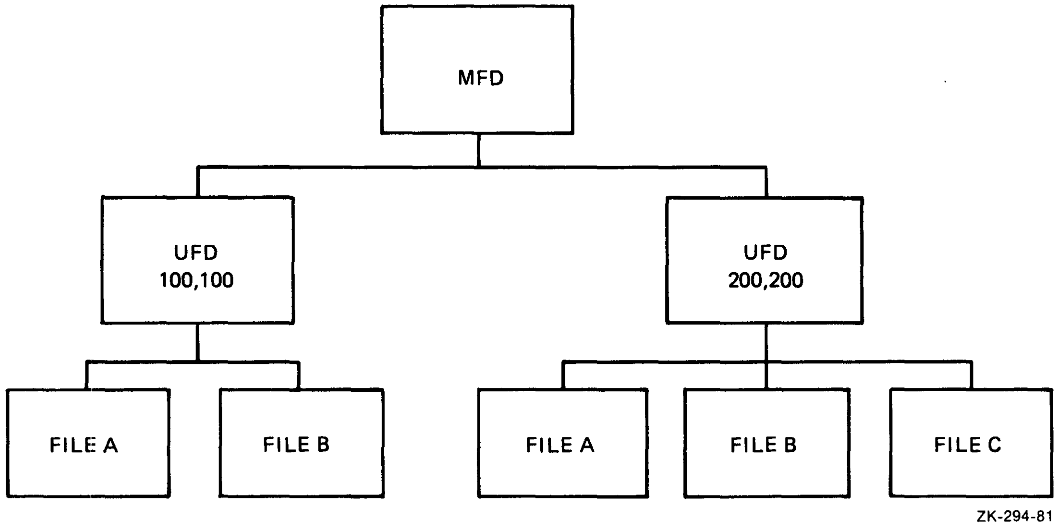


Figure 5-2 Directory Structure for Multiuser Volumes

5.1.4 File Header Block

Each file has a file header block that contains a description of the file. File header blocks are stored in the index file. Each file header block is 256 words long and contains 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. FCS uses the first word of the file ID (that is, the file number) to calculate the virtual block number (VBN) of that file's header block in the index file. (This calculation is done as follows: the VBN is the file number + 2 + the number of index file bit map blocks.) The operating system uses the second word (that is, the file sequence number) to verify that the header block is really the header for the desired file.

When you request file access, both the file number and the file sequence number are specified. The system denies a request for access if the file sequence number does not match the corresponding field in the file header block that is associated with the specified file number.

When you delete a file, its file header block space becomes available for storing a newly created file's sequence number. If you attempt to access a file that has been deleted (for example, by referencing an obsolete directory entry), this updated file sequence number ensures the rejection of the request for access, even if the same file header block is reused for a different file.

The identification area specifies the file's creation name 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, 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 is transferred correctly. Appendix F contains a detailed description of the format and content of the file header block.

5.2 MAGNETIC TAPE FILE PROCESSING

RSX-11M/M-PLUS and Micro/RSX systems support the standard ANSI magnetic tape structure as described in "Magnetic Tape Labels and File Structure for Information Interchange," ANSI X3.27-1978. Any of the following file/volume combinations can be used:

- Single file on a single volume
- Single file on more than one volume
- Multiple files on a single volume
- Multiple files on more than one volume

The second and fourth file and volume combinations constitute a volume set.

The record format on magnetic tape differs from that on disk. When a file containing variable-length records or fixed-length records that cross block boundaries is copied to magnetic tape, it occupies more blocks on the magnetic tape than it did on the disk. This is so because magnetic tape record counts are larger disk record counts, and there is unused space at the end of the blocks. In addition, a bit is set in the file's FDB that indicates the file cannot cross block boundaries.

Appendix G defines the sequence in which volume and file labels are used and the format of each label type.

NOTE

> The ANSI file header label contains no place for the creation time or the length of the file. Consequently, the creation time of a file on ANSI magnetic tape is listed as 0. If a contiguous file is copied to ANSI magnetic tape and then transferred back to disk, the resulting disk file is not marked as contiguous even if you use the /CO switch, because the system cannot know how much space to allocate for the output file when it reads from magnetic tape.

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. Only one file in a volume set can be open at a time. The system protects access by volume set rather than by file. On volumes produced by DIGITAL systems, the contents of the owner identification file determine user access rights as described in Section G.1.1.1. Volumes produced by non-DIGITAL systems are restricted to read-only access unless the access is overridden explicitly at MOUNT time.

5.2.2 Rewinding Volume Sets

You can rewind a magnetic tape volume set by using either the FDOP$R macro before an OPEN$ or CLOSE$ macro or by using the .CTRL file control routine. Regardless of the method you use, FCS performs these procedures:

1. All mounted volumes are rewound to the beginning-of-tape (BOT).

2. If the first volume in the set is not mounted, the device-unit to be used is placed off line.

3. If the volume is not already mounted and if the rewind was requested with an OPEN$ macro or by a .CTRL routine call, a request to mount the first volume appears on the operator's console.

4. If the rewind was requested with a CLOSE$ macro, no mount message is issued until the next volume is needed.

5.2.3 Positioning to the Next File Position

The standard procedure for writing a new file onto a magnetic tape is to begin writing the file following the end of the volume set's last file. However, you can use the FDOP$R macro to indicate that the new file is to be written immediately after the labels at the end of the most recently closed file.

NOTE

The next file position option causes the loss of any files physically following this most recently closed file in the volume set.

If, in addition to the next file position option, the rewind option also is specified, the file is created after the VOL1 label on the first volume of the set. All files previously contained in the entire 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 itself at the logical end of the volume set to create the file.

When you use the default, the file system makes no check 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.

Directory device file processors ignore the next file position option. However, programs written mainly for directory devices can specify the next file position option in open commands for output and, therefore, override a process of positioning the file system to the logical end of file normally used with ANSI magnetic tape.

5.2.4 Single-File Operations

You perform single-file operations by specifying the rewind option with the FDOP$R macro before the open and before the close. Using this approach, you can perform operations on temporary tapes or work tapes (scratch tapes) as follows:

1. Open the first file with the rewind option specified.
2. Write the data records and close the file with rewind.
3. Open the first file again for input (rewinding 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

You create a multiple-file volume by first opening, writing, and then closing a series of files without specifying the rewind option. You can process files sequentially on the volume by closing without rewind and opening the next file without rewind.

Opening a file for extend with the OPEN$A macro is legal only for the last file on the volume set.

Perform the following tape operations to create a multiple-file tape volume:

1. Open a file for output with the rewind option.
2. Write data records and close the file.
3. Open the next file without rewinding.
4. Write the data records and close the file.
5. Repeat for as many files as desired.

You can open files on tape in a nonsequential order, but doing so increases processing and tape-positioning time. Nonsequential access of files in a multivolume set is not recommended.

5.2.6 Using .CTRL

You can call the .CTRL file control routine to override normal FCS defaults for magnetic tape. This routine might be used to:

- Continue processing a file on the next volume of a volume set before the end of the current volume is reached.
- Position to the logical end-of-volume set.
- Rewind a volume at other times than when opening or closing the file.
- Space forward or backward any number of records.
- Rewind a file.

When FCS uses the .CTRL routine 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 these volumes constitutes a file section. For input files, the following .CTRL routine processing occurs:

1. If the current volume is the last volume in the set (that is, there is no next volume), the end-of-file is reported to you.
2. If another file section exists, the current volume is rewound and the next volume is mounted. A request to mount the next volume appears on the operator's console.
3. The header label (HDR1) of the next 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 .CTRL routine 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 a write operation. 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 the .CTRL routine positions the tape to the logical end of the volume, the file system positions the tape between the two tape marks at the logical end of the last volume in the set.

When the .CTRL routine spaces forward or backward across blocks on magnetic tape, spacing crosses volumes for multivolume files.

5.2.7 Examples of Magnetic Tape Processing

The following sections contain examples of FCS statements that process magnetic tape. Macro parameters not related to magnetic tape handling are omitted from these statements.

5.2.7.1 Examples of OPEN$W Macro-11 Statements 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$R  R0,,,,,#0                ;DISABLE USE OF AND RESET
        BR      OPNOUT                   ;F.ACTL TO ZERO
;
; OPEN FILE WITH CURRENT USER ACCESS CONTROL
;
OPOURO:
        BIS     #FA.ENB,F.ACTL(R0)       ;ENABLE USE OF F.ACTL
OPNOUT: FDBF$R  R0,,#2048.               ;OVERRIDE BLOCK SIZE FOR TAPE
        OPEN$W  R0
        RETURN
```

5.2.7.2 Examples of OPEN$R Macro-11 Statements 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
```

```
;
; OPEN USING USER ACCESS CONTROL
;
OPDFLI: BIS     #FA.ENB,F.ACTL(R0)      ;ENABLE USE OF F.ACTL
OPNIN:  FDBF$R  R0,,#2048.              ;OVERRIDE BLOCK SIZE FOR TAPE
        OPEN$R  R0
        RETURN
```

5.2.7.3 **Examples of CLOSE$ Macro-11 Statements** - 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$ Macro-11 Statements** - The following examples call routines shown in previous examples in Section 5.2.7.1. By combining various magnetic tape operations, you 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
5       RETURN
;
; TO APPEND TO LAST FILE
;
        OPEN$A  #FDBOUT
```

CHAPTER 6

COMMAND LINE PROCESSING

This chapter describes two object library routines that are available from the system object library, [1,1]SYSLIB.OLB. These routines may be linked with your task to provide the logical capabilities necessary to process terminal command line input as follows:

- **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 you of the burden of manually coding command line input operations.

- **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 that FCS requires for opening files.

The Task Builder links these routines with your program when the task is being built. GCML and CSI are often used together in system or application programs as a standardized interface for obtaining and interpreting dynamic command line input. Figure 6-1 shows the flow of data during command line processing.

Although this chapter assumes the joint use of these routines to process command line input, GCML and CSI may be used independently. Using one without the other, however, requires that you manually code the functions normally performed by the missing component.

Invoking GCML and CSI functions requires that certain initialization be done when you write the source code. 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 that invoke GCML and CSI execution-time functions must be included in the source code at appropriate logical points to effect the dynamic processing of command lines.

GCML and CSI macro calls observe the same register conventions as FCS. All registers except R0 are preserved exactly as those in FCS macro calls. R0 contains 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 be listed as an argument in an .MCALL directive (see Section 2.1) before you insert them in your program.

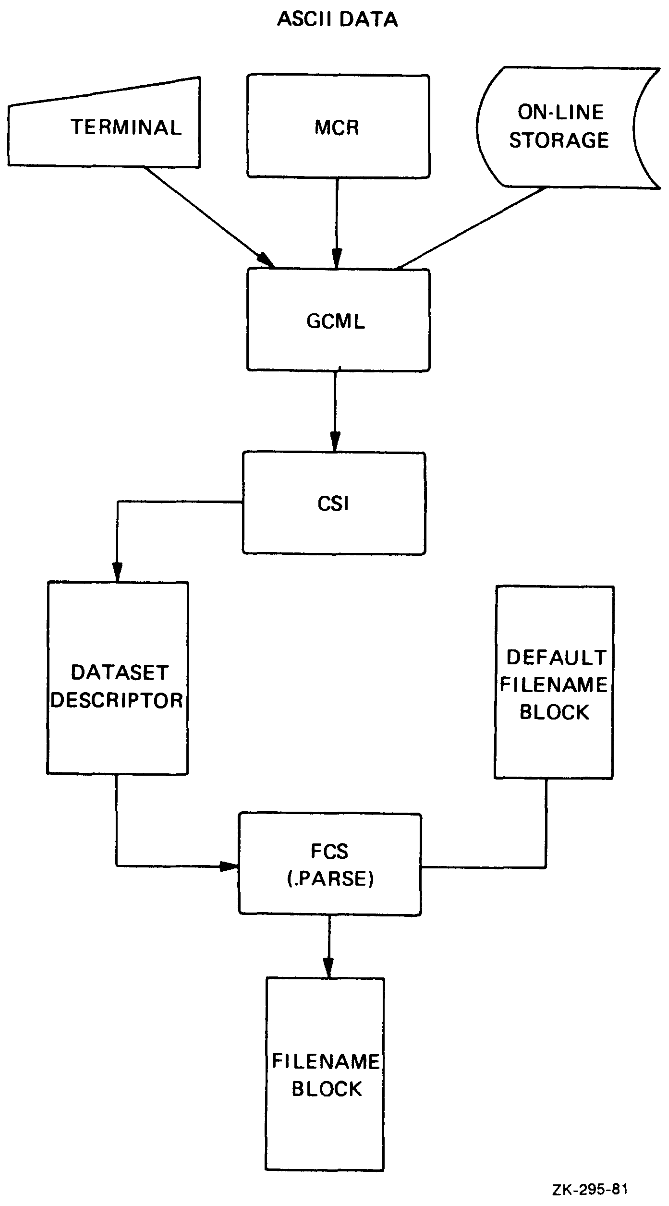


Figure 6-1 Data Flow During Command Line Processing

6.1 GET COMMAND LINE (GCML) ROUTINE

The Get Command Line (GCML) routine contains all the logical capabilities necessary to enter command lines dynamically during program execution. GCML accepts input from a terminal or an indirect command file that contains predefined command lines. If your program allocates sufficient buffer space in the file storage region (see Section 2.6), GCML accepts commands that are longer than one line of terminal input. The appearance of a hyphen as the last printing character of a command line permits the continuation of commands from one line to the next.

All GCML functions require you to create and initialize a GCML control block. See Section 6.1.1 for a description of this macro call. The GCML run-time macro calls that your task may issue dynamically are described in Section 6.1.3.

6.1.1 GCMLB$ - Allocate and Initialize GCML Control Block

This section describes the GCMLB$ macro. This macro is a necessary part of the code needed to dynamically obtain and execute command lines. During the assembly of your program the GCMLB$ macro:

- Reserves storage for and initializes a GCML control block within your program.
- Creates and initializes an FDB for the indirect command file in the first part of the GCML control block.
- Creates and initializes a default filename block within the GCML control block.
- Defines the symbolic offsets for the GCML control block and initializes certain offsets to required values by invoking the GCMLD$ macro. These offsets are described in detail in Section 6.1.2.

FCS uses the FDB to open an indirect command file. Your program may open and read a command file, which can use a terminal or a file-structured device such as a disk. GMCL and FCS initialize and maintain this FDB.

FCS uses the default filename block for an indirect command file. If you do not specify an explicit filename string for an indirect command file, the values CMI for the file name and .CMD for the file type are assumed by default. There is no default designation for the device name.

The GCMLB$ macro has the following format:

Format

```
label:  GCMLB$  maxd,prmpt,ubuf,lun,pdl,size
```

Parameters

label

> 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 that require access to this structure (see Section 6.1.3).

maxd

> 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 can access in obtaining command line input.
>
> An indirect command file, which often resides on disk, contains well-defined, nonvarying command sequences, which may be read directly by GCML to control such highly repetitive operations as Task Builder activities.
>
> If you do not specify this parameter, the default nesting level depth is 0, which effectively eliminates an indirect command file as a source of command line input.

prmpt

A 3-character ASCII prompting sequence that you specify. The GCML routine displays this default prompt string at your terminal to solicit command line input.

Construct the ASCII prompting sequence as the following 6-byte string:

- A carriage return (<CR>) and a line-feed (<LF>)
- The three ASCII characters that you specify
- A right angle bracket (>)

The ASCII prompting sequence initializes GCML control block offset location G.DPRM (see Section 6.1.2).

If you do not specify this parameter, GCML uses the right angle bracket (>), preceded by three blanks, as the default prompting sequence.

ubuf

The address of a buffer that the GCML routine uses for temporary storage of command line input. If you do not specify this parameter, a buffer is reserved in the GCML control block for command line input. The size parameter determines the length of the buffer. If you specify neither this parameter nor the size parameter, a 41-word buffer is reserved by default in the GCML control block.

lun

A logical unit number. The GCML routine uses the device assigned to this logical unit number as the command input device. If you do not specify this parameter, GMCL uses a logical unit number of 1 by default.

pdl

The address of an area reserved in your program as a push-down list. Indirect command file processing uses this area for working storage. Normally, you do not specify the pdl parameter unless you want to increase the storage for the push-down list.

Statements logically equivalent to the following create the push-down list:

```
         .EVEN
label:   .BLKB    G.LPDL
```

The label that you supply specifies 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. You can increase the storage in the control block for the push-down list by calculating the value according to the following algorithm:

1. Add 1 to the maximum nesting level depth declared with the maxd parameter described previously.

2. Multiply the sum of step 1 by 16(decimal) to find the number of bytes that must be reserved for the push-down list.

For example, if you specify 4 as the maxd parameter, you determine the length of the push-down list as follows:

```
(4+1)*16. = 80.  bytes
```

From the previous mathematical statement, note that 16(decimal) bytes of storage are required for each indirect command file (4), plus another 16(decimal) bytes as general overhead.

size

The size, in bytes, of the buffer reserved for command line input. The size must always include two extra bytes that are used internally by GCML. The default size value is 82 (that is, 80 bytes for command line input and 2 bytes GCML overhead).

If you want GCML to accept continuation lines, the specified value for the size parameter must be greater than 82. When the size is greater than 82, the bit value GE.CON is set in the status and mode control byte (offset G.MODE) of the GCML command block. This value indicates that the continuation mechanism is in effect.

Examples of how a GCMLB$ macro call may be used in a program follow:

```
GCLBLK: GCMLB$  4.,GCM,BUFADR,1.
GCLBLK: GCMLB$  ,,BUFADR
GCLBLK: GCMLB$  DEPTH,GCM,BUFADR,CMILUN,PDLIST,BUFSIZ
```

6.1.2 GCMLD$ - Define GCML Control Block Offsets and Bit Values

The GCMLD$ macro, which the GCMLB$ macro call invokes, locally defines the GCML control block offsets and bit values within the current module. A description of these offsets and their bit values follows:

Symbolic Offset Name	Description
G.ERR	**Error Return Code Byte** This field initially contains 0. If any error conditions that GCML recognizes occur during the processing of a command line, an appropriate error code is returned to offset location G.ERR in the control block. Descriptions of these error bits follow: **GE.IOR** - I/O error occurred during the input of a command line. **GE.OPR** - GCML unable to open or reopen the specified command file. **GE.BIF** - Syntax error detected in the name of the indirect command file. **GE.MDE** - Attempt made to exceed the maximum permissible nesting-level depth for an indirect command file (see the description of the maxd parameter in Section 6.1.1).

Symbolic Offset Name	Description
	GE.RBG - Command line input buffer was too small for the total command. This condition can occur when multiple lines have been entered using the continuation mechanism. The input buffer contains as much of the command as possible.
	GE.EOF - End-of-file (EOF) on the top-level command file detected.

NOTE

For GE.IOR and GE.OPR, additional information concerning the error is available by examining the FCS error code at offset F.ERR from the start of the GCML block.

The error code is set along with command file input. When the first call is issued for input, GCML attempts to retrieve an MCR command line. Command level 0 is set for the first line obtained, whether it is an MCR command or a terminal command. If the name of an indirect command file is then entered, the command input level is increased to 1. Each indirect file name 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 1, restoring the count to the previous level and reopening the associated command file. The next command line from that file is then read.

If an MCR command has already been read at level 0, entering another MCR 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 command line can be read at level 0. If input thus fails at MCR level 0, then GCML continues to prompt for input until you type CTRL/Z 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 command; if the MCR 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, that is, from the command input file.

G.MODE **Status and Mode Control Byte**

This field is initialized at assembly time with bit definitions to specify certain default actions for GCML during the retrieval of a command line.

Symbolic Offset Name	Description

At run time, you can reset default status and mode control bits by issuing a Bit Clear Byte (BICB) instruction that takes the symbolic name of the bit to be cleared as the source operand. In the case of the GE.LC value (see the following text), the BISB instruction can override the default action.

Descriptions of the symbolic names of the bits defined in the status and mode control byte follow:

GE.IND - (Default) A command line that begins with a leading at sign (@) is an explicit indirect command file specification. If you reset the GE.IND bit to 0, a command line beginning with an at sign (@) is returned to the calling program.

GE.CLO - (Default) The command file currently being read is closed after each GCML$ macro call is issued. If you reset the GE.CLO bit to 0, GCML keeps the current command file open between calls for input. In this case, the File Storage Region (FSR) described in Section 2.6.1 must include one additional 512(decimal)-byte buffer for command line input. This requirement adds 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 renders 512(decimal) bytes of FSR block buffer space unavailable for other purposes, because the command file remains open between calls for command line input.

GE.COM - (Default) A command line that begins with a leading semicolon (;) is a comment. Such lines are not returned to the calling program. If you reset this bit to 0, a command line beginning with a leading semicolon (;) is returned to the calling program.

GE.CON - If the value of the size parameter of the GCMLB$ macro is greater than 82, the continuation mechanism is in effect by default. You must not attempt to set this bit in the mode byte without providing a buffer larger than 82 bytes.

GE.LC - If this bit is set to 1 in the GCML control block at run time, lowercase characters in the command line are passed unaltered to your program. If this bit is not set, lowercase characters are changed to upppercase before being passed to your program.

Symbolic Offset Name	Description
G.PSDS	**Prompt String Descriptor** This 2-word field is initialized at assembly time to 0 by issuing the GCMLB$ macro call (see Section 6.1.1). When you issue the GCML$ macro call to request command line input (see Section 6.1.3.1), the address and the length of a prompting sequence are 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 to solicit command line input. (See the description of G.DPRM in the following text.) If you want to define an alternate prompt string elsewhere in the program, you 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. (See the description of these parameters and how they affect alternate prompt strings in the following text.) 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. GCML then types out the alternate prompt string to solicit command line input, thereby overriding the default prompt string previously established through the GCMLB$ macro call. If you do not specify the adpr and lnpr parameters in a subsequent GCML$ macro call, offset location G.PSDS in the control block is reset to 0, causing GCML to revert to the use of the default prompt string contained in offset location G.DPRM.
G.CMLD	**Command Line Descriptor** GCML initializes this 2-word field after retrieving a command line. The address of this command line 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 that enable CSI to retrieve file specifications 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

Symbolic Offset Name	Description
	for pointers, flags, counters, and so forth, along with input from an indirect command file. In normal usage, you need not be concerned with this symbol. The space between the FDB and the default prompt string (see G.DPRM in the following text) is the impure area of the GCML control block. The value of the symbol S.FDB defines the size of the FDB. Thus, the size of the impure area is equal to G.DPRM minus S.FDB (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), GMCL types out this default prompt string to solicit terminal input.

You can reference the GCML control block offsets and bit values in another module by establishing the appropriate symbolic definitions within that module through one of the following statements:

```
GCMLD$                  ;DEFAULT LOCAL DEFINITION

GCMLD$  DEF$L           ;LOCAL DEFINITION

GCMLD$  DEF$G           ;GLOBAL DEFINITION
```

6.1.3 GCML Routine Run-Time Macros

GCML provides the following three run-time macro calls to perform specific functions:

GCML$	- Retrieves a command line
RCML$	- Resets the indirect command file scan to the first (unnested) level
CCML$	- Closes the current command file

These routines are described in the following sections.

6.1.3.1 **GCML$ - Get Command Line Macro** - GCML$ serves as your program interface for retrieving command lines from a terminal or an indirect command file. You can issue this macro call at any logical point in the program to solicit command line input.

This macro call has the following format:

Format

```
GCML$   gclblk,adpr,lnpr
```

Parameters

gclblk

> 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 you do not specify this parameter, R0 is assumed to contain the address of the GCML control block.

adpr

> The address of your program location containing an alternate prompt string. When this optional parameter and the lnpr parameter following are present in the GCML$ macro call, the alternate prompt string appears on your 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

> The length (in bytes) of the optional, alternate prompt string. If you do not specify this parameter, offset location G.PSDS in the GCML control block (see Section 6.1.2) is cleared.
>
> If you specify this parameter, but do not specify the adpr parameter described previously, an .ERROR directive is generated during assembly that 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 you specify this parameter, as well as the adpr parameter, the default prompt string is used.

If you do not specify the adpr and lnpr parameters in a subsequent GCML$ macro call, offset location G.PSDS in the GCML control block is reset to 0. Consequently, GCML reverts to using the default prompt string contained in offset location G.DPRM (see Section 6.1.2).

When you issue the GCML$ macro call, the following occurs:

1. R0 is loaded with the address of the GCML control block. If you do not specify the gclblk parameter, R0 is assumed to contain the address of the GCML control block. If it does not contain that address, you must first manually initialize R0 with the address of the control block before you issue the GCML$ macro call.

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 that calls GCML to transfer a command line to the command line input buffer. If the last character of an input line is a hyphen, and if the value GE.CON is present in the status and mode control byte, GCML transfers commands that are longer than one line. The continuation lines obtained are concatenated in the input buffer with the continuation hyphen(s) removed.

When your task first issues the GCML$ macro call, GCML$ tries to retrieve an MCR command line. If this attempt fails, or if the MCR command line is null, GCML uses the FDB within the GCML control block to open a file for command line input. If the command input device is a terminal, a prompt string appears on your terminal to solicit input. Any appropriate command input may then be entered. If the continuation mechanism is in effect, the prompt string reappears to solicit subsequent portions of the continued command line.

If appropriate, you may enter an at sign (@) as the first character in the command line, followed by the name of an indirect command file. This file name identifies an explicit indirect command file from which input is to be read. GCML then opens this file and retrieves the first command line. On successive GCML calls, this file is read until one of the following occurs:

- 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 reduced by 1, and the previous command file is reopened. If the command input level count is already 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).

- An indirect file specification is encountered in a command line. In this case, the current indirect command file is closed (if not already closed), the new indirect file is opened, and the first command line is read.

- An RCML$ macro call is issued in the program (see Section 6.1.3.2). In this case, the current indirect command file is closed, and the command input count reverts to level 0; that is, the top-level command file is again used for input.

You may also enter a semicolon (;) as the first character in the command line. If GE.COM is set, such a line is treated as a comment and is not returned to the calling program. If GE.COM is clear, the line is 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 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 (see Section 6.2.3.1).

Successful retrieval of a command line causes the Carry bit in the Processor Status Word to be cleared. Any error condition that occurs during the retrieval of a command line, however, causes the Carry 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.

Examples of how you may use the GCML$ macro in a program 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 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 you have defined within the program. GCML uses this alternate prompt string 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 Macro** - If you must close the current indirect command file and return to the top-level file, that is, to the top-level (unnested) file, you may do so by issuing the RCML$ macro.

You specify the RCML$ macro in the following format:

Format

```
     RCML$   gclblk
```

Parameter

gclblk

The address of the GCML control block. If you do not specify this parameter, R0 is assumed to contain the address of the GCML control block.

When you issue this macro, the current indirect command file is closed, returning control to the top-level (unnested) file. A subsequent GCML$ macro then retrieves the next command line from the 0-level command file. Note, however, that a second MCR 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 how you may use the RCML$ macro in a program 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 Macro** - You may want to close the current command file between calls for input to free FSR block buffer space for some other use. FCS normally closes the command file 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 1 at assembly time. If you reset this bit to 0, the current command file remains open between calls for input.

For a program that frequently reads command files, this may be a desirable operational mode, because keeping the file open between calls for input reduces total file access time. However, should you want to close such a file to free FSR block buffer space, you may do so by issuing the CCML$ macro call.

The CCML$ macro call has the following format:

Format

```
     CCML$   gclblk
```

Parameter

gclblk

> The address of the GCML control block. If you do not specify this parameter, R0 is assumed to contain the address of the GCML control block.

Issuing this statement closes the current command file, effectively releasing 512(decimal) bytes of FSR block buffer space for some other use between calls for input. If the command file is already closed when your task issues the CCML$ macro call, control is returned to your task. A subsequent GCML$ macro call then causes the command file to be reopened and the next command line in the file to be returned to the calling program.

Examples of how the CCML$ macro may be used in a program follow:

```
     CCML$   #GCLBLK

     CCML$   R0
```

As in the RCML$ macro call described previously, this macro call takes a single parameter, specifically, 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 first part of the GCML control block. Although ordinarily you need not manipulate this FDB (because it is under GCML and FCS control), you can perform the following operations on this FDB:

1. In an unrecoverable error situation, you can issue a CLOSE$ macro call (see Section 3.8) with the address of this FDB before issuing the system EXIT$ macro call.

2. You can test the FD.TTY bit in the device characteristics byte (offset location F.RCTL) of the FDB to determine whether the command line just obtained was retrieved from a terminal.

3. In the event that error code GE.IOR or GE.OPR is returned to control block offset location G.ERR (indicating that an I/O error has occurred during the retrieval of a command line), you can test offset location F.ERR of the associated FDB for a more complete error analysis. This FDB cell also contains an error code that may be helpful in determining the nature of the error condition.

At task-build time, the Task Builder device assignment (ASG) option 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 option 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 option is described in further detail in the RSX-11M/M-PLUS and Micro/RSX Task Builder Reference Manual.

As covered in the Section 2.6.1 discussion on FSRSZ$, at any given time there must be an FSR block buffer available for each file currently open for record I/O operations. You must consider the buffer requirements of the command file when issuing the FSRSZ$ macro (FSRSZ$ must be issued with a nonzero first parameter).

6.2 COMMAND STRING INTERPRETER (CSI) ROUTINE

The Command String Interpreter (CSI) routine analyzes command lines and parses them into their component device name, directory, and filename strings. You should be aware that CSI processes command lines in the following formats only:

- dev:[g,m]outputfilespec/switch

  More than one file specification can be specified by separating them with commas.

- dev:[g,m]outputfilespec/switch,...= dev:[g,m] inputfilespec/switch,...

A file specification may be either of the following:

```
filename.type;version
```

or

```
"ANSI name string";version
```

CSI maintains a dataset descriptor within the CSI control block (see Section 6.2.1), which FCS may use in opening files. The run-time routines that analyze and parse command lines for your calling program are described in Section 6.2.3.

Using 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 Macro

Following is the only initialization coding required for CSI at assembly time:

```
        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 does not generate any executable code. The CSI control block resulting from the .BLKB directive allows communication between CSI and the calling program. The symbol C.SIZE specifies the length of the control block. C.SIZE is defined during the expansion of the CSI$ macro. Expanding this macro also causes a local definition of the symbolic offsets and bit values within the CSI control block.

You can 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 follows:

```
     CSI$    DEF$G
```

6.2.2 CSI$ Macro Control Block Offset and Bit Value Definitions

The CSI$ macro locally defines the following symbolic offsets and bit values within the CSI control block:

Symbolic Offset Name	Description
C.TYPR	**Command String Request Type** This byte field indicates which type of file specification is being requested. Depending on whether an input or output file specification is being requested (see the io parameter in the CSI$2 macro call described in 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 specification is being requested. CS.OUT - Indicates that an output file specification 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 according to the following bit definitions: 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 specifications. Once CS.EQU is set, CSI1 and CSI2 processing preserve the value of CS.EQU. CS.NMF - Indicates that the current file specification contains a filename string. Accordingly, control block offset locations C.FILD+2 and C.FILD (see the entry for C.FILD) are initialized with the address and the length

Symbolic Offset Name	Description
	(in bytes), respectively, of the command line segment that contains 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 specification contains a directory string. Thus, control block offset locations C.DIRD+2 and C.DIRD (see the description following for C.DIRD) are initialized with the address and the length (in bytes), respectively, of the command line segment that contains the directory string. If no directory string is present, this bit is not set. In this case, any residual nonzero values in the directory string descriptor cells that pertain to a previous command string request of similar type are used by default (see the description of C.TYPR). Thus, FCS uses the last directory string encountered in a file specification.
	CS.DVF - Indicates that the current file specification contains a device name string. Similarly, control block offset locations C.DEVD+2 and C.DEVD (see the description of C.DEVD) 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. Like CS.DIF (see the previous description of CS.DIF), any residual nonzero values in the device name descriptor cells that pertain to a previous command string request of similar type are used by default. Thus, the last device name string encountered in a file specification is used.
	CS.WLD - Indicates that the current file specification contains an asterisk (*), signaling the presence of a wildcard specification.
	CS.MOR - Indicates that the current file specification is terminated by a comma (,), which indicates that more file specifications are to follow. If this bit is not set, it signifies that the end of the input or output file specification has been reached.
C.CMLD	**Command Line Descriptor** This 2-word field is initialized with the length (in bytes) and the address, respectively, of the compressed command line. In other words, the values returned to these cells are the CSI output after it scans a file specification and removes all nonsignificant characters from the string (that is, nulls, unquoted blanks and tabs, and RUBOUTs).

Symbolic Offset Name	Description
	CSI uses the values contained in these cells as the descriptors of the compressed command line to be parsed (see CSI$2 macro call in Section 6.2.3.2).
C.DSDS	**Dataset Descriptor Pointer** This pointer defines the address of the 6-word dataset descriptor in the CSI control block. This structure is functionally identical to the manually created dataset descriptor detailed in Section 2.4.1. You can use this symbol to initialize offset location F.DSPT in the FDB associated with the file to be processed. Thus, FCS is able to retrieve the ASCII information from this structure that it needs to open files. Assembly-time initialization of F.DSPT in the associated FDB may be accomplished as follows: `FDOP$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 the following entries for offset names) identifying the requisite ASCII filename information. Run-time initialization of F.DSPT in the associated FDB may also be accomplished by using 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 those with the same request type) encountered in a file specification. Note that the colon that follows the device name is not included in the device name string.
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 those with the same request type) encountered in a file specification.
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 specification.

Symbolic Offset Name	Description
	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), 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 switches present in the current file specification after each invocation of the CSI$2 macro call. The switch mask for each of the defined switches encountered in a file specification between delimiting commas is inserted into this location by a logical OR operation. The mask for a switch is specified in the CSI$SW macro call (see Section 6.2.4.1). When a switch is encountered in a file specification for which a defined mask exists, the corresponding bits in C.MKW1 are set. By testing C.MKW1, you can determine the particular combination of defined switches present in the current file specification.
C.MKW2	**CSI Mask Word 2** This word provides you with an indication of switch polarity. When a switch is present in a file specification and you do not negate that switch, the defined mask for that switch is inserted into C.MKW2 by a logical OR operation in the same manner as described previously for C.MKW1. Conversely, when a switch is present in a file specifier and you do negate that switch, the corresponding bits in C.MKW2 are cleared. Thus, you can check the polarity of each switch that C.MKW1 indicates is present 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.
C.EXPS	**User Task Expansion Buffer Size** This symbol is the constant for your task's expansion buffer size (for logical name expansion). It is currently set to 48(decimal). See the description of the CSI$4 routine in Section 6.2.3.3 for more information.

6.2.3 CSI Run-Time Macros

Three run-time macro calls in CSI invoke routines that 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 nonsignificant characters from the line, and checks it for syntactic validity. This macro also initializes 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 specification 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. In addition, certain cells in the CSI control block are initialized with the appropriate string descriptors for subsequent use by FCS in opening the specified file.

CSI$4 - Expands a file specification and returns a dataset descriptor and flag word that refer to this expanded file specification. The file specification string is not expanded into the original buffer but into a separate buffer in the file storage region. The use of CSI$4 is recommended when parsing logical file specification strings.

6.2.3.1 **CSI$1 - Command Syntax Analyzer** - The CSI$1 macro invokes 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 correct syntax. In addition, it compresses the file specifications in the command line by removing all nonsignificant characters (that is, null, RUBOUT, and unquoted tabs and blanks). 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 specification in the command line is then parsed into its component device name, directory, and filename strings during each successive time the CSI$2 macro call is issued (see Section 6.2.3.2).

The CSI$1 macro call has the following format:

Format

```
CSI$1   csiblk,buff,len
```

Parameters

csiblk

The address of the CSI control block. If you do not specify this parameter, R0 is assumed to contain the address of the CSI control block.

buff

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 you do not specify this parameter, you 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. The following statement shows one way to manually initialize this location:

```
MOV GCLBLK+G.CMLD+2,CSIBLK+C.CMLD+2
```

len

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 that enables CSI to retrieve the current command line from the input buffer.

As with the buff parameter described previously, if you do not specify this parameter, you 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. The following statement shows one way to manually initialize this location:

```
MOV GCLBLK+G.CMLD,CSIBLK+C.CMLD
```

The combination of the buff and len parameters described previously 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 syntactical error is detected during the validation of the command line, the Carry 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 that define the address and the length, respectively, of the command line segment in error.

Examples of how the CSI$1 macro call may be used in a program follow:

```
CSI$1     #CSIBLK,#BUFF,#LEN

CSI$1     R0,GCLBLK+G.CMLD+2,GCLBLK+G.CMLD

CSI$1     #CSIBLK
```

The first example shows symbols that 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.

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 Macro** - The CSI$2 macro invokes 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 routine then parses the referenced line into its component device name, directory, and filename strings. The equal sign (=) in the command line indicates that the string that follows is an input file specification. 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 your program (see Section 2.4.1).

The parser also decodes any switches and associated switch values present in a file specification, provided that the address of the appropriate switch descriptor table has been specified in the CSI$2 macro call (see the following text). The CSI switch definition macro calls are described in detail in Section 6.2.4.

The CSI$2 macro call has the following format:

Format

```
     CSI$2   csiblk,io,swtab
```

Parameters

csiblk

The address of the CSI control block. If you do not specify this parameter, R0 is assumed to contain the address of the CSI control block.

io

A symbol that identifies the type of file specification to be parsed. You may specify either of the following two symbolic arguments in this parameter field:

INPUT - The next input file specification in the command line is to be parsed.

OUTPUT - The next output file specification in the command line is to be parsed.

You must initialize offset location C.TYPR in the CSI control block (see Section 6.2.2), either manually or through the CSI$2 macro call, with the type of file specification being requested. If arguments other than the symbolic arguments defined previously are specified in the CSI$2 macro call, an .ERROR directive is generated during assembly that causes the error message INCORRECT REQUEST TO .CSI2 to be printed in the assembly listing. This diagnostic message alerts you to the presence of an invalid io parameter in the CSI$2 macro call.

swtab

The address of the associated switch descriptor table. You specify this optional parameter only if you suspect that the file specification contains a switch to be decoded. For you to specify this parameter, the program must already contain a switch descriptor table, which you created with the CSI$SW macro (see Section 6.2.4.1). In addition, if the switch to be decoded has any associated switch values, the program must already contain an associated switch value descriptor table, which you create with the CSI$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 you do not specify this parameter, FCS uses any residual nonzero value in this cell by default as the switch descriptor table address.

You can also initialize offset location C.SWAD 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. (The switch table must be aligned on an even address.)

If an error condition occurs during the parsing of the file specification, the Carry 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:

- The request type is invalid; that is, offset location C.TYPR in the CSI control block (see Section 6.2.2) is incorrectly initialized.
- The file specification contains a switch, but the address of the switch descriptor table is not specified in the CSI$2 macro call, or the switch descriptor table does not contain a corresponding entry for the switch.
- The file specification contains an invalid switch value.
- The number of values accompanying a given switch in the file specification are greater than the number of corresponding entries in the switch value descriptor table for decoding those values.
- The file specification contains a negative switch, but the corresponding entry in the switch descriptor table prevents you from negating the switch (see the nflag parameter of the CSI$SW macro call in Section 6.2.3.3).

Examples of how the CSI$2 macro may be used in a program follow:

```
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 specification, which may include an associated switch. The second example, which assumes that R0 presently contains the address of the CSI control block, parses an output file specification, which also may include a switch. The last example is a request to parse an input file specification and to disallow any accompanying switches.

6.2.3.3 **CSI$4 - Command Semantic Parser Macro**

Use CSI$4 in the same way that you use CSI$2. However, CSI$4 is the preferred method of parsing logical names. Using CSI$4 causes the same function as CSI$2 except that CSI$4 allows all tasks to use logical names correctly, except under the following two conditions:

1. The task saves the dataset descriptor for one call to CSI$2 and then calls CSI$2 again, expecting to be able to use the first dataset descriptor at a later time.
2. The task assumes that the dataset descriptor points into the original string.

CSI$4 cannot be called repeatedly using the previous Dataset Descriptor. The reason is that whenever a parse is done, whether by the .PARSE routine (see Section 4.7.1), or the CSI$4 macro, or the .EXPLG routine, the expanded string is always put into the FSR area of the task. If your task repeatedly calls CSI$4, each call overwrites the previous call. CSI$4 uses this common buffer for input file specifications, but uses a separate expansion buffer for output file specifications. This permits CSI$4 to process input and output file specifications simultaneously.

To allow multiple calls to CSI$4, another parameter is added to the CSI macro, so that:

```
CSI$2    #CSIBLK,INPUT,#CSISWT
```

is changed to:

```
CSI$4    #CSIBLK,INPUT,#CSISWT,#DSCBLK
```

The DSCBLK parameter is a pointer to a 2-word descriptor block that specifies the expanded command line buffer. The descriptor block contains the following two words:

WORD 0: Contains the size of the expanded command line buffer

WORD 1: Contains the address of the expanded command line buffer

The descriptor block and the command line expansion buffer are allocated in your task's address space. The size of the expansion buffer should be as large as any possible single file specification. Therefore, C.EXPS, currently set to 48(decimal), should be used. If the buffer is too small, an error is returned. If either value in the descriptor block is 0, the expansion area in the FSR is used.

Logical name expansion requires a relatively large amount of system overhead. These steps should be followed to reduce the overhead demand:

1. Do not change CSI$2 macros to CSI$4 macros when you want your task to only scan the line for switches or files without opening any files found. In this case, changing CSI$2 to CSI$4 only adds to the extra translation required.

2. Include the following line before any call to .PARSE:

   ```
   BISB   #FL.AEX,F.FLG(R0)   ;R0 is the FDB address
   ```

   This code line informs .PARSE that the string has already been expanded and .PARSE need not repeat the expansion.

6.2.4 CSI Switch Definition Macros

The following macro calls create the requisite switch descriptor tables in your program for processing switches that appear in a file specification:

CSI$SW - Creates an entry in the switch descriptor table for a particular switch that you expect to encounter in a file specification.

CSI$SV - Creates a matching entry in the switch value descriptor table for the switch defined through the CSI$SW macro.

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 in the following sections.

6.2.4.1 **CSI$SW - Create Switch Descriptor Table Entry Macro** - You must define a matching entry in the switch descriptor table for each switch that you expect your task to encounter in a file specification. If no switch descriptor table is specified or no corresponding entry exists, the presence of a switch in the command line causes an error. When your task issues a CSI$2 macro (see Section 6.2.3.2) and the address of a switch descriptor table is specified, the following processing occurs:

1. For each switch encountered in a file specification, CSI searches the switch descriptor table for a matching entry. If either the switch descriptor table address is not specified, or a matching switch entry is not found in the table, that switch is considered invalid. As a result, the Carry bit in the Processor Status Word is set, any remaining switches in the file specification 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 you do not negate the switch, 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 you negate the switch but negation is not allowed, the switch is considered invalid. In this case, the error sequence described in step 1 would occur. However, if you negate the switch, and negation is allowed, 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 (described in the following text).

4. If the optional mask word address is not present in the corresponding switch descriptor table entry, that is, if you did not specify the mkw parameter in the associated CSI$SW macro, switch processing continues with step 7. If, however, you did specify the optional mask word address, 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, you negate the switch, the corresponding bits in the optional mask word are cleared.

   You specify the clear/set flag as the csflg parameter in the CSI$SW macro.

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 you negate the switch, the corresponding bits in the optional mask word are set.

7. If a switch value accompanies a switch in a file specification, FCS uses the associated switch value descriptor table created through the CSI$SV macro call (see Section 6.2.4.2) 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 specification. If the switch value descriptor table is incomplete, or an invalid switch value is encountered, or the address of the switch value descriptor table is not present in the associated switch descriptor table, the switch is invalid, and the error sequence described in step 1 would occur.

   You specify the address of the switch value descriptor table as the vtab parameter in the CSI$SW macro call.

The CSI$SW macro call has the following format:

Format

```
label:  CSI$SW  sw,mk,mkw,csflg,nflg,vtab,compflg
```

Parameters

label

An optional symbol that names the resulting switch descriptor table entry and defines its address. 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

The switch name to be stored as an entry in the switch descriptor table. This name may comprise any number of alphabetic characters. CSI compares the name entered on the command line with this switch name as entered in the switch descriptor table. This is a required parameter; if you omit it, the Assembler generates an .ERROR directive during assembly that causes the error message MISSING SWITCH NAME to be printed in the assembly listing.

mk

A mask that you define for the switch specified through the sw parameter. To enable CSI to indicate the presence of a given switch in a file specification, you must define a mask value for the switch, as follows:

```
ASMSK = 1
NUMSK = 2
  .
  .
  .
VWMSK = 40000
XYMSK = 100000
```

where the octal value that you assign to each symbol defines a unique bit configuration. This configuration is to be set in CSI mask word 1 (C.MKW1) of the control block when a defined switch is encountered in a file specification.

When you specify 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 you can determine the particular combination of switches present in a file specification. For every matching entry found in the switch descriptor table, the corresponding bits are set in C.MKW1.

mkw

The address in your program storage of a mask word that CSI changes each time it changes C.MKW1. CSI stores the same value into this mask word that it stores into C.MKW1. This mask word can be manipulated, that is, changed or tested by the SET and CLEAR functions or by instructions in your program. You set the SET and CLEAR functions using the csflg parameter.

Such an optional word may be reserved through a statement logically equivalent to the following:

```
MASKX:  .WORD   0
```

csflg

A symbolic argument that specifies the clear/set flag for a given switch. This parameter is optional; if you do not specify it, SET is assumed. You may specify either one of two symbolic arguments for this parameter, as follows:

CLEAR - Indicates that the bits in the optional mask word corresponding to the switch mask are to be cleared, provided that you did not negate the switch. (If you negate the switch, the bits are set.)

SET - Indicates, conversely, that the bits in the optional mask word in your task corresponding to the switch mask are to be set, provided that you did not negate the switch. (If you negate the switch, the bits are cleared.)

If you specify other than SET or CLEAR, the Assembler generates an .ERROR directive that causes the error message INVALID SET/CLEAR SPEC to be printed in the assembly listing.

nflg

Specifies an optional negation flag for the switch. If you specify this parameter, it indicates that the switch can be negated, for example, /-LI or /NOLI.

If you specify this parameter as other than NEG, the Assembler generates an .ERROR directive that causes the error message INVALID NEGATE SPEC to be printed in the assembly listing. If you do not specify this parameter, the assumption is that switch negation is not allowed.

vtab

The address of the switch value descriptor table associated with this switch. If you specify this optional parameter, it allows CSI to decode any switch values accompanying the switch, provided that you have defined an associated switch descriptor table entry for that switch. The CSI$SV macro defines the switch value descriptor table. (If you specify the vtab parameter in the CSI$SV macro, you need not specify it in the CSI$SW macro call.)

compfl g

Defines the method CSI uses to compare the switch name entered on the command line with the value entered in the switch descriptor table by the sw parameter. Either LONG or EXACT may be specified. The default value is entered if you do not specify a value.

Default - If you do not code the parameter, only the first two characters of the switch name (specified by sw) are entered into the switch descriptor table and only these two characters are compared when the command line is parsed. Additional characters in the command line switch name are ignored.

LONG - All characters specified by the sw parameter are entered in the switch descriptor table. During compare processing, the first characters of the switch name on the command line must exactly match the value for the switch in the switch descriptor table. Additional characters in the command line switch name are ignored.

EXACT - All characters specified by the sw parameter are entered in the switch descriptor table. During compare processing, all the characters of the switch name on the command line must exactly match the value in the switch descriptor table. Extra characters in either the command line or the table are treated as an error.

The switch table must be aligned on an even address. The format of the switch descriptor table entry created by the CSI$SW macro is shown in Figure 6-2.

The switch name characters precede the control information in the table. The sign bit of each word indicates whether the following word contains more switch name characters. A sign bit set to 1 indicates that the next word contains more switch name characters, whereas a sign bit set to 0 indicates that this is the last word containing switch name characters.

If the number of characters in the switch name is odd, the high-order byte of the last word contains zeros, and CSI ignores it.

The sign bit of the first byte of the last word of the switch name is the EXACT match bit. If this bit is set to 1, additional characters in the switch name on the command line are treated as an error by CSI; if this bit is set to 0, additional characters are ignored.

The switch name characters are followed by entry control information consisting of the CSI mask word, the address of the area task of a mask word corresponding to the CSI mask word, and the address of the switch value table.

A bit setting of 1 in the low-order bit of the address of your mask word indicates the CLEAR function; a bit setting of 0 indicates the SET function.

The last word of the switch descriptor table entry contains the address of the switch value table. A bit setting of 1 in the low-order bit of this word indicates that the switch may be negated.

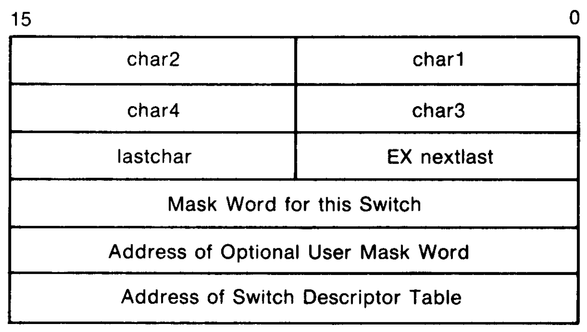

15
0
char2
char1
char4
char3
lastchar
EX nextlast
Mask Word for this Switch
Address of Optional User Mask Word
Address of Switch Descriptor Table
ZK-296-81

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 parameter in the first statement creates an entry in the switch descriptor table for the switch /AS. The second parameter is an equated symbol that defines the switch mask, and the third parameter (MASKX) is the address of an optional mask word in your task (see the description of the mkw parameter). The fourth parameter indicates that the bits in MASKX that correspond to the switch mask are to be set. The fifth parameter (the negation flag) is null. The last parameter is the address of the associated switch value descriptor table.

The second statement creates 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 mask word (MASKX) in your task that 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 switch value descriptor table associated with this switch.

Note that the switch descriptor table entry macros are terminated with the CSI$ND macro (see Section 6.2.4.3).

6.2.4.2 CSI$SV - Create Switch Value Descriptor Table Entry Macro -

CSI$SV defines a switch value descriptor table entry. For every switch value that you expect your task to find with a given switch in a file specification, a corresponding switch value descriptor table entry must be defined in your program so that the switch value can be decoded. This macro creates a 2-word entry in the switch value descriptor table. The format of this table is shown in Figure 6-3.

The CSI$SV macro call has the following format:

Format

```
CSI$SV type,adr,len
```

Parameters

type

A parameter that specifies the conversion type for the switch value. Any one of four symbolic values may be specified. The possible conversion types include the following:

ASCII - Indicates that the switch value is to be treated as an ASCII string. If you quote the string, the quotes are returned in the buffer as part of the string. If a quote appears anywhere in the switch value, all characters following it, up to the end of the line or another quote, are included in the string.

NUMERIC - Indicates a numeric switch value is to be converted to binary using octal as a default conversion radix.

OCTAL - Indicates a numeric switch value is to be converted to binary using octal as a default conversion radix.

DECIMAL - Indicates a numeric switch value is to be converted to binary using decimal as a default conversion radix.

If any parameter is specified other than these, an .ERROR directive is generated during assembly that causes the error message INVALID CONVERSION TYPE to be printed in the assembly listing. If you do not specify any of the previously described parameters, ASCII is assumed by default.

adr

The address of your program location that 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 that causes the error message VALUE ADDRESS MISSING to be printed in the assembly listing.

len

A numeric value that defines the length (in bytes) of the area that is to receive the switch value that results from switch processing. This parameter is also required; if not specified, an .ERROR directive is generated during assembly that causes the error message LENGTH MISSING to be printed in the assembly listing.

The format of a switch value descriptor table entry created by a CSI$SV macro is shown in Figure 6-3.

The low-order byte of the first word in the switch value descriptor table indicates whether the conversion type is ASCII or numeric. The low-order byte of this word is set to 1 if ASCII is specified; it is set to 2 if NUMERIC or OCTAL is specified, and is set to 3 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 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 padded with nulls.

If the conversion type is numeric, the length of the resulting binary value is either two bytes or four bytes. If the size field is less than 4, 2 bytes are stored. If the size field is greater than 4, 4 bytes are stored. You must align the buffer on a word boundary.

If you specify the default conversion type for a switch value on numeric conversions, you can override it with a pound sign (#) or a period (.). Preceding a numeric value by a pound sign (for example, #10) forces the conversion type to octal; a numeric value followed by a period (for example, 10.) forces the conversion type to decimal. Note also that you may precede a numeric switch value with a plus sign (+) or a minus sign (-). The plus sign is the default assumption. If you specify an explicit octal switch value by using the pound sign (#), the arithmetic sign indicator (+ or -), if included, must precede the pound sign (for example, -#10).

If the conversion type is decimal, the switch value is evaluated as a single number; an overflow into the high-order bit (bit 15) causes an error condition. However, if the conversion type is octal, a full 16-bit value may be specified.

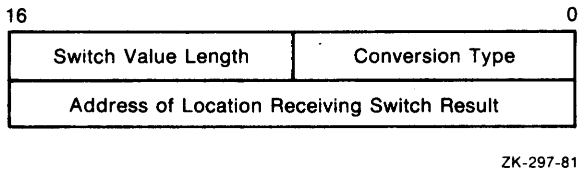


Figure 6-3 Format of Switch Value Descriptor Table Entry

Examples of how the CSI$SV macro call may appear in a program follow:

```
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 these examples, the first parameter in the CSI$SV macro defines the conversion type. The next two parameters, in all cases, define the address and the length of the program location that is to receive the resultant switch value.

You may reserve the required storage for the first switch value table ASVTBL: as follows:

```
ASVAL:  .BLKW   4                ;ASCII VALUE STORAGE
```

You can similarly reserve the required storage for the second switch value table NUVAL: 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$ND - Define End of Descriptor Table** - CSI$ND terminates descriptor tables with a 1-word entry. Switch descriptor tables and switch value descriptor tables must be terminated with a 1-word end-of-table entry. You can create this word, which contains 0, with the CSI$ND macro call.

This macro call takes no arguments. The examples near the ends of Sections 6.2.4.1 and 6.2.4.2 illustrate the use of this macro call.

CHAPTER 7

THE TABLE-DRIVEN PARSER (TPARS)

This chapter describes the table-driven parser (TPARS), which parses command lines. TPARS permits you to define and parse command lines in a unique syntax using TPARS-supplied macros, built-in variables, and your own code.

TPARS parses command lines according to syntax and semantics or meaning. The command line is made up of syntax elements. TPARS evaluates each syntax element of the command line based on a predefined arrangement of those elements. TPARS parses command lines using a table that you define. You can build a state table, which contains states and transitions, using the TPARS STATE$ and TRAN$ macros. A state delimits and represents a single syntax element on a command line. A transition is a statement that defines the processing required for parsing a given syntax element and contains instructions for further parsing at another state. TPARS uses subexpressions to resolve complex syntax elements. On the semantic level, TPARS also resolves the semantics or meaning of each element based on definitions supplied within action routines of your program. These action routines use TPARS macros, built-in variables, and your code to permit you to define and parse command lines.

The parser routine that you write is included in your programs that parse command lines. TPARS is invoked from within an executing program by means of a CALL instruction. The CALL invokes the parser routine as well as the TPARS processor. For further information on the interrelationships among the calling program, the user-defined parser routine, and the TPARS processor, refer to Section 7.5.

7.1 CODING TPARS SOURCE PROGRAMS

This section describes the three TPARS macros required to initialize and define the state table. Also included in this section is information describing action routines, TPARS built-in variables, and TPARS subexpressions.

7.1.1 TPARS Macros: ISTAT$, STATE$, and TRAN$

TPARS provides macros that allow you to write a state table for parsing a unique command line. The ISTAT$ macro initializes a state table, the STATE$ macro defines a state (a particular syntax element) in your state table, and the TRAN$ macro defines the conditions for transition to another state.

7.1.1.1 **ISTAT$ Macro - Initialize the State Table** - ISTAT$ initializes the state table. The state table is built using two macros, STATE$ and TRAN$, which are described in Sections 7.1.1.2 and 7.1.1.3, respectively. This state table is built into a program section. Keyword strings that you define for parsing command lines are also accumulated in a program section. A third program section is also provided for a keyword pointer table used to enter the list of keyword strings. The ISTAT$ macro initializes these program sections.

A blank STATE$ macro must follow the TPARS state table.

Format

```
ISTAT$ statetable,keytable,$DEBUG
```

Parameters

statetable

The label that you assign to the state table. TPARS recognizes this label as the start of the state table.

keytable

The label that you assign to the keyword table. TPARS recognizes this label as the start of the keyword table.

$DEBUG

Directs the Assembler to list addresses of the state transition table in the assembly listing. These addresses are useful for tracing TPARS operation using a debug routine that you supply (see Section 7.1.3.4). When you do not include $DEBUG, state transition table addresses are not listed.

The state table is built in a program section named $STATE, the keyword strings are accumulated in a program section named $KSTR, and the keyword pointer table is built in a program section named $KTAB.

If you define the symbol $RONLY, each of these program sections is generated as read-only. You generate a read-only state table by specifying the symbol $RONLY before the ISTAT$ macro in the form:

```
$RONLY = 1
ISTAT$ statetable,keytable,$DEBUG
     .
     .
     .
STATE$
```

7.1.1.2 **STATE$ Macro - Defining a Syntax Element** - STATE$ declares the beginning of a state. This macro delimits one command line syntax element from another. A blank STATE$ macro must follow the TPARS state table.

Format

```
     STATE$ [label]
```

Parameter

label

An alphanumeric symbol that defines the address of the state.

Each state defined by a STATE$ macro consists of any number of transitions defined by TRAN$ macros. The TRAN$ macros parse each syntax element.

7.1.1.3 **TRAN$ Macro - Defining a Transition** - The TRAN$ macro allows you to match each syntax element in a command line to a given type, supply a symbolic address to the next TRAN$ macro, supply an address of an action routine that may be required to further process the syntax element, and supply a mask that you may use as a flag in the parsing process.

Format

```
     TRAN$ type[,label][,action][,mask][,maskaddr]
               [,$EXIT]
```

Parameters

type

The type of command line syntax element being parsed. You code the type parameter using one of the following types of command line elements:

Element Type	**Description**
$ANY	Matches any single character.
$ALPHA	Matches any single alphabetic character (A-Z).
$DIGIT	Matches any single digit (0-9).
$LAMDA	Matches an empty string. This transition is always successful. LAMDA transitions are useful for getting action routines called without passing any of the input string.
$NUMBR	Matches any number. A number consists of a string of digits; a concluding period is optional. Numbers not followed by a period are interpreted as octal. Numbers followed by a period are interpreted as decimal and the decimal point is included in the matching string. A number is terminated by any Ononnumeric character. Values through 2**32-1 are converted to 32-bit unsigned integers.
$DNUMB	Matches a decimal number. The string of digits is interpreted as decimal. With the exception that the matched string does not include the trailing decimal point, TPARS treats $DNUMB the same way it treats $NUMBR.

Element Type	Description
$STRNG	Matches any alphanumeric character string. The string will not be null.
$RAD50	Matches any legal Radix-50 string, that is, any string containing alphanumeric characters, or the period (.), or dollar sign ($) characters. If you require Radix-50 conversion, the action routine in your code must convert this number.
$BLANK	Matches a string of blank and/or tab characters.
$EOS	Indicates the position of the end of an input string. Once TPARS has reached the end of the input string, $EOS is the equivalent of that position as many times as $EOS is encountered in the state table.
char	Matches a single character in the syntax element whose ASCII code corresponds to the value of char. The value of char must be a 7-bit ASCII code; that is, the value must be in the range 0-177 (octal). Specify a single quote (') before char, such as 'A or 'X.
"keyword"	Matches a specified keyword. Keywords can be any length, can contain only alphanumeric characters, must be in uppercase, and are terminated by the first nonalphanumeric character encountered in parsing the keyword. The maximum number of keywords allowed in a state table is 64.
!label	Matches the string processed by passing control to and executing the state table section that starts with a STATE$ macro that has the label parameter specified here as !label. In effect, this type parameter passes control to a STATE$ macro subroutine or subexpression. For information on TPARS subexpressions, see Section 7.1.4.

[label]
[$EXIT]

The label associated with a STATE$ macro to which execution control will pass after the code for this TRAN$ transition is executed. If the label parameter is omitted, execution control passes on to the next sequential STATE$ macro. A null label parameter is allowed only for the last transition in a state; a TRAN$ macro with a null label field must follow a TRAN$ macro.

Specifying $EXIT in the label field terminates TPARS execution and returns control to the calling program. $EXIT also terminates a TPARS subexpression.

action

The label of an action routine that you include in the parser routine of your code. This routine can include TPARS built-in variables, described in Section 7.1.3 below.

mask

A maskword to be stored in a location pointed to by the maskword address whenever the TRAN$ macro is executed. If you specify mask, you must specify the maskaddr parameter as well (see the following parameter). This maskword is ORed into maskaddr (described below) when the transition is taken (after the action routine is called).

maskaddr

The label for an address into which TPARS stores the value specified by the mask parameter. You must specify the maskaddr parameter if you specify mask.

The mask and maskaddr parameters provide a convenient means for flagging the execution of a particular transition.

7.1.2 Action Routines and Built-In Variables

Action routines process command line elements at the semantic level. That is, a given syntax element can have more than one meaning. Action routines determine and validate the meaning of the syntax elements.

You write action routines in your parsing program to perform unique functions related to your program's requirements.

7.1.2.1 **TPARS Built-In Variables** - TPARS provides the following built-in variables for action routines:

.PSTCN	Returns the character count of the portion of the input string matched by this transition. This character count is valid for all syntax types recognized by TPARS, including subexpressions.
.PSTPT	Returns the address of the portion of the input string matched by this transition. This address is valid for all syntactical types recognized by TPARS, including subexpressions.
.PNUMH	Returns the high-order binary value of the number returned by a $NUMBR or $DNUMB syntax type specification.
.PNUMB	Returns the low-order binary value of the number returned by a $NUMBR or $DNUMB syntax type specification.
.PCHAR	Returns the character found by the $ANY, $ALPHA, $DIGIT, or char syntax type specifications.
.PFLAG	Returns the value of the flag word passed to TPARS by register 1 (R1). Action routines can modify this word to change options dynamically.
.TPDEB	Contains the entry address of the optional debug routine that you write.

R3	Returns the byte count of the remainder of the input string. When the action routine is called, the string does not include the characters matched by the current transition.
R4	Returns the address of the remainder of the input string. When the action routine is called, the string does not include the characters matched by the current transition.

7.1.2.2 Calling Action Routines

- Action routines are called by a JSR PC instruction. Action routines may modify registers R0, R1, and R2; all other registers must be preserved.

7.1.2.3 Using Action Routines to Reject a Transition

- Action routines can reject a transition by returning to CALL+4 rather than to CALL+2. That is, the action routine performs the same function as an ADD #2,(SP) before returning to the caller. This technique allows additional processing of syntax types and extending the syntax types beyond the set provided by TPARS.

When an action routine rejects a transition, that transition has no effect. TPARS continues to attempt to match the remaining transitions in the state.

7.1.2.4 Optional Debug Routine for RSX-11 Users

- A debug routine that you supply can be called by TPARS at each state transition allowing TPARS operation to be traced. For example, the routine can be written to display the contents of R5 each time the routine is called; R5 contains the current transition table address. By comparing the addresses displayed with the TPARS assembly listing showing the state transition table addresses, TPARS operation can be monitored.

If a debug routine that you supply is to be called by TPARS, your task must first specify the entry point address for the debug routine in TPARS location .TPDEB, as follows:

```
        MOV     #DENTER,.TPDEB
```

Then, invoke TPARS with the .TPARD entry point (rather than with .TPARS). TPARS is invoked as described in Section 7.4.

Upon entry to the debug program, CPU registers contain the following:

R3 = Length of remainder of input string

R4 = Address of remainder of input string

R5 = Current address of transition table

The debug routine must save and restore all registers prior to returning to TPARS.

For addresses displayed by the debug routine to be useful, you must obtain an assembly listing showing the addresses of the state transition tables. These addresses are listed by the assembler if the optional $DEBUG parameter is provided in the ISTAT$ macro call (see Section 7.1.1.1).

7.1.3 TPARS Subexpressions

A TPARS subexpression is a series of states and transitions analogous to a subroutine. In general, such a series of states and transitions is used more than once during the parsing process.

Subexpressions begin with a STATE$ macro specifying the label of the subexpression. You follow this macro by the states and transitions that comprise the body of the subexpression. To terminate the subexpression, specify a TRAN$ macro with the $EXIT keyword specified in the label field. The general form of a subexpression is shown in the example below.

In this example, control is directed to the subexpression by a TRAN$ macro that specifies a !label syntax element as the type parameter:

```
     TRAN$ !UIC,NEXT
```

TPARS then directs control to the STATE$ macro with the label UIC:

```
     STATE$       UIC
     TRAN$ '[

     STATE$
     TRAN$ $NUMBR,,SETGN

     STATE$
     TRAN$ <',>

     STATE$
     TRAN$ $NUMBR,,SETPN

     STATE$
     TRAN$ '],$EXIT
```

When the UIC subexpression completes processing, control passes to the state labeled NEXT.

7.2 GENERAL CODING CONSIDERATIONS

This section contains information on how to arrange syntax types in a state table and how to direct TPARS to ignore blanks and table characters in a command line, and rules for entering special characters (commas and angle brackets).

7.2.1 Suggested Arrangement of Syntax Types in a State Table

The transitions in a state may represent several syntax types; a portion of a string being scanned often matches more than one syntax type. Therefore, the order in which you enter the types in the state table is critical. Transitions are always scanned in the order in which they are entered, and the first transition matching a string being scanned is the transition taken. Therefore, the following order is recommended for states containing more than one syntax type:

```
     char
     keyword
     $EOS
     $ALPHA
     $DIGIT
     $BLANK
```

```
$NUMBR
$DNUMB
$STRNG
$RAD50
$ANY
$LAMDA
```

Placement of !label transitions in a state depends on the types and positions of other syntax types in the state, as well as on the syntax types in the starting state of the subexpression.

7.2.2 Ignoring Blanks and Tabs in a Command Line

Bit zero of the low byte of Register 1 (R1) controls processing of blanks and tab characters. If this bit is 1 when TPARS is invoked, blanks and tab characters are processed in the same way any other ASCII character is processed; they are treated as syntax elements that require validation by TPARS. If this bit is set to 0, blanks and tab characters are interpreted as terminator characters; they are ignored as syntax elements. In neither case does TPARS modify the command line.

When blanks are being ignored, the $BLANK syntax type never matches an element on the command line. Also, when this option is in effect, values returned to the !label syntax type by .PSTCN or .PSTPT may contain blanks or tabs, even though none were requested. The examples below show how TPARS parses the string:

```
ABC DEF
```

with and without the blank-suppress option.

In the first example, an extra state is required to parse the blank:

```
STATE$
TRAN$         $STRNG

STATE$
TRAN$         $BLANK

STATE$
TRAN$         $STRNG
```

When TPARS is directed to ignore blanks and tab characters, the same string can be parsed using only two states:

```
STATE$
TRAN$         $STRNG

STATE$
TRAN$         $STRNG
```

7.2.3 Entering Special Characters

In char syntax elements, MACRO-11 interprets commas (,), semicolons (;), and angle brackets (< >) as special characters. The comma is interpreted as an argument separator and angle brackets are used to parenthesize special characters.

To include a comma or a semicolon in a char syntax element string, use angle brackets:

```
     TRAN$ <',>
```

Angle brackets cannot be passed as string elements in macro arguments. If required in a "char" expression, they must be expressed symbolically, for example:

```
     LA = '<
     TRAN$ LA
```

7.2.4 Recognition of Keywords

When TPARS encounters a transition table entry that specifies a keyword, it first scans from the current point in the input string in search of a delimiter (nonalphanumeric) character. The characters between the current input point and the next delimiter are then assumed to be a possible keyword and are matched against the entries in the keyword table. For this reason, the following example will not work as expected:

```
     STATE$
     TRAN$    "NO",STATE1,SETNEG
     TRAN$    $LAMDA,,SETPOS

     STATE$   STATE1
     TRAN$    "AA",...
     TRAN$    "BB",...
```

When TPARS encounters the keyword NO, it scans and attempts to match the string "NOAA" or "NOBB". If exact matching is requested, neither the "NO" transition nor the "AA" transition will match. In addition, if keyword matching is limited to two characters, the "NO" transition will match but TPARS will skip past "NOAA" so that the "AA" transition can be taken. You can use the following example to achieve the desired operation:

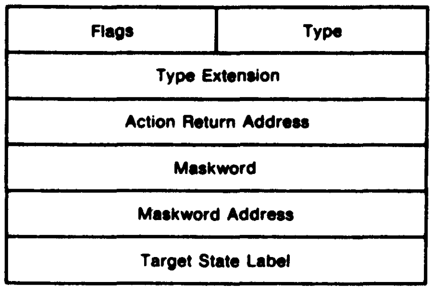
```
     STATE$
     TRAN$    !NONO,STATE1,SETNEG
     TRAN$    $LAMDA,,SETPOS

     STATE$   STATE
     TRAN$    "AA",...
     TRAN$    "BB",...
          .
          .
          .
     STATE$   NONO
     TRAN$    'N

     STATE$
     TRAN$    'O,$EXIT
```

In this example, TPARS attempts to match the subexpression NONO to the "NO" prefix one character at a time. This bypasses the keyword scanning of TPARS, allowing the input pointer to be left pointing at "AA" or "BB". If NONO fails, the input pointer will not be changed and the scan can continue by looking for "AA" or "BB".

7.3 PROGRAM SECTIONS GENERATED BY TPARS MACROS

TPARS macros generate three program sections. Data for the STATE$ macro are stored in the program section $STATE, whereas data for the TRAN$ macro are stored in program sections $KSTR and $KTAB. $KTAB contains addresses for each of the entries of the keyword syntax type. $KSTR contains the keyword entries separated by character code 377 (octal).

Each state consists of its transition entries concatenated in the order in which you specify them. The state label, if specified, is equated to the address of the first transition in the state. Each transition consists of from one to six words, as follows:



ZK-314-81

The type byte of the first word may contain the following values:

$LAMDA	= 300	
$NUMBR	= 302	
$STRNG	= 304	
$BLANK	= 306	
$SUBXP	= 310	Used in the !label type.
$EOS	= 312	
$DNUMB	= 314	
$RAD50	= 316	
$ANY	= 320	
$ALPHA	= 322	
$DIGIT	= 324	
char	= ASCII code for the specified character	
keyword	= 200+n (See explanation below.)	

The value of keyword is 200+n, where n is an index into the keyword table. The keyword table is an array of pointers to keyword strings, which are stored in the program section $KSTR. Keyword strings in $KSTR are separated from each other by 377 (octal).

Bits in the flags byte indicate whether parameters for the TRAN$ macro are specified:

Bit	**Meaning**
0	Type extension is specified.
1	Action routine label is specified.
2	Target state label is specified.
3	Maskword is specified.
4	Maskword address is specified.
7	Indicates last transition in the current state.

7.4 INVOKING TPARS

You control execution of TPARS using the calling conventions and options described in this section. You invoke TPARS from within an executing program by the instruction

```
     CALL    .TPARS
```

When a debug routine that you specify traces a TPARS operation (see Section 7.1.3.4), a special entry point is called, as follows:

```
     CALL    .TPARD
```

When your task calls TPARS in this manner, TPARS calls the debug routine at each state transition. If your task invokes TPARS by the .TPARS entry point, the debug routine entry point address in .TPDEB is cleared and the debug routine is not called.

7.4.1 Register Usage and Calling Conventions

When TPARS is invoked, registers in the calling program must contain the following information:

```
     R1 = Options word
     R2 = Pointer to the keyword table
     R3 = Length of the string to be parsed
     R4 = Address of the string to be parsed
     R5 = Label of the starting state in the state table
```

On return from TPARS processing, registers contain the following information:

```
     R3 = Length of the unscanned portion of the string
     R4 = Address of the unscanned portion of the string
```

The values of all other registers are preserved.

The Carry bit in the Processor Status Word returns 0 for a successful parse; the Carry bit is set when TPARS finds a syntax error.

For an example of a calling sequence for TPARS, refer to Section 7.6.1.

7.4.2 Using the Options Word

The low byte of the options word contains flag bits. The only flag bit defined is bit zero, which controls processing of blanks. If bit zero is set to 1, blanks are interpreted as syntax elements. If bit zero is set to 0, blanks are ignored as syntax elements.

The high byte of the options word controls abbreviation of keywords. If the high byte is set to 0, keywords being parsed must exactly match their corresponding entries in the state table. If the high byte is set to a number, keywords being parsed may be abbreviated to that number of characters. Keywords in the string that are longer than the number specified must be spelled correctly up to the length specified by the number.

TPARS clears the Carry bit in the Processor Status Word when it completes processing successfully. This occurs when a transition is made to $EXIT that is not within a subexpression.

If a syntax error occurs, TPARS sets the Carry bit in the Processor Status Word and terminates.

A syntax error occurs when there are no syntax elements in the current state that match the portion of the string being scanned. Illegal type codes and errors in the state table can also cause a syntax error.

TPARS processing requires that the addresses in the state table and the keyword tables be reliable; bad addresses may cause program termination.

The only syntax types that can match the end of the string are $EOS and $LAMDA.

7.5 HOW TO GENERATE A PARSER PROGRAM USING TPARS

Three processing steps generate a parser program using TPARS, as shown in Figure 7-1. The source program must contain .MCALL statements for three macros: ISTAT$, STATE$, and TRAN$. These three .MCALL statements must precede the statements that comprise the state table and action routines.

Assembling the source module produces an object module composed of three program sections. The assembly listing showing the code produced by the state table macros is not straightforward. The binary output of the macros is delayed by one statement. Thus, if you enable the listing of macro-generated binary code during assembly of the code, the binary code appearing after a macro call is, in fact, the result of the preceding macro call. Error messages generated by macro calls are similarly delayed. This is the reason an additional STATE$ macro is required to terminate the state table.

When the parser program is linked and is in task image form, it can be invoked from within your executing task, as shown in Figure 7-2.

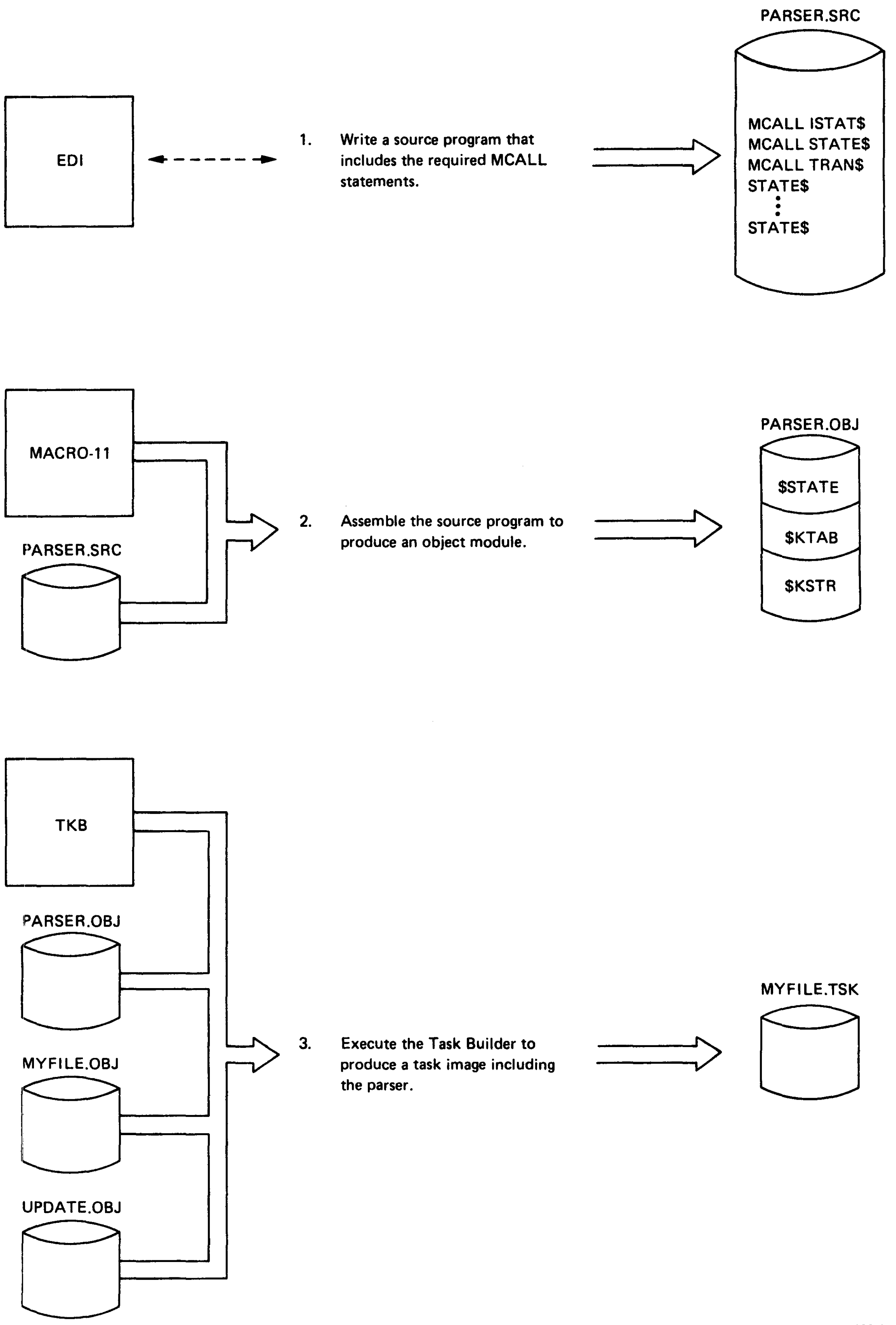

PARSER.SRC
EDI
1. Write a source program that includes the required MCALL statements.
MCALL ISTAT$
MCALL STATE$
MCALL TRAN$
STATE$
STATE$
MACRO-11
PARSER.SRC
PARSER.OBJ
2. Assemble the source program to produce an object module.
$STATE
$KTAB
$KSTR
TKB
PARSER.OBJ
MYFILE.OBJ
UPDATE.OBJ
3. Execute the Task Builder to produce a task image including the parser.
MYFILE.TSK
ZK-298-81

Figure 7-1 Processing Steps Required to Generate a Parser Program Using TPARS

Figure 7-2 shows the CALL .TPARS statement that invokes the parser program and the TPARS processor. As the parser executes the state table, it calls action routines. These action routines access code in the TPARS processor to perform such functions as returning the values of the built-in variables. When the state table completes execution, TPARS receives control and passes control back to the calling program.

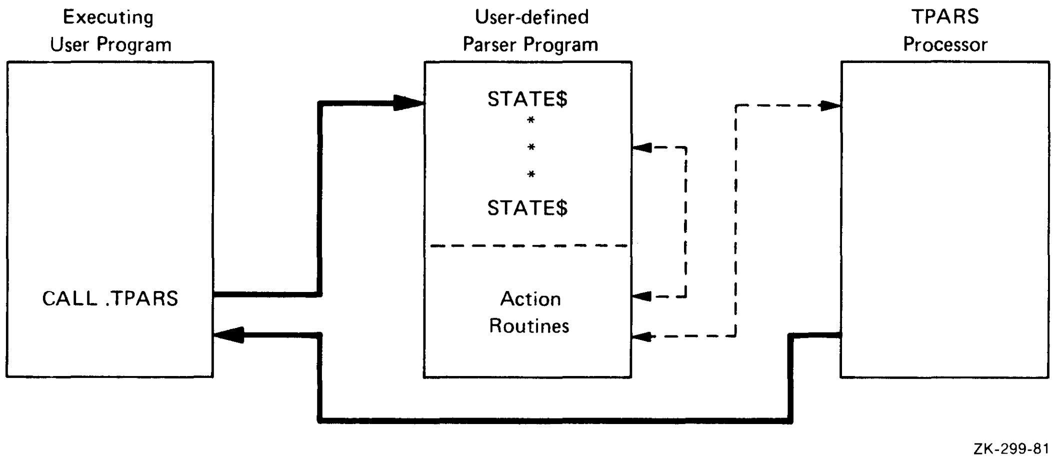


Figure 7-2 Flow of Control When TPARS Is Called from an Executing User Program

7.6 PROGRAMMING EXAMPLES

This section includes three programming examples of how TPARS can be used in your program. The first example shows the code required to parse a UFD command line for RSX-11. The second example shows the use of subexpressions and how to reject transitions. The third example shows how to use subexpressions to parse complex command lines.

7.6.1 Parsing a UFD Command Line

This example shows the code required to parse a UFD command line. It includes a state table and action routines. The general form of the UFD command line is as follows:

```
UFD DK0:LABEL[201,202]/ALLOC=100./PRO=[RWED,RWED,RWE,R]
```

The action routines in this parser program return the following values:

$UDEV	Device name (2 ASCII characters)
$UUNIT	Unit number (binary)
$UVNML	Byte count of the volume label string
$UVNAM	Address of the volume label string
$UUIC	Binary UIC for which to create a directory
$UALL	Number of directory entries to preallocate
$UPRO	Binary protection word for UFD
$FLAGS	Flags word containing the following bits: UF.ALL Set if allocation was specified UF.PRO Set if protection was specified

The label and the /ALLOC and /PRO switches are optional. The calling sequence for this routine is as follows:

```
     CLR      R1
     MOV      #UFDKTB,R2
     MOV      COUNT,R3
     MOV      ADDR,R4
     MOV      #START,R5
     CALL     .TPARS
     BCS      ERROR
```

The following is an example of the parser routine that you write:

```
                          .TITLE   STATE TABLE FOR UFD COMMAND LINE

                          .MCALL   ISTAT$,STATE$,TRAN$

     ; TO BE USED WITH BLANK SUPPRESS OPTION

                          ISTAT$   UFDSTB,UFDKTB

     ; READ OVER COMMAND NAME

                          .GLOBL   START

                          STATE$   START
                          TRAN$    "UFD"

     ; READ DEVICE AND UNIT NUMBER

                          STATE$
                          TRAN$    $ALPHA,,SETDV1

                          STATE$
                          TRAN$    $ALPHA,,SETDV2

                          STATE$
                          TRAN$    $NUMBR,DEV1,SETUNT
                          TRAN$    $LAMDA

                          STATE$   DEV1
                          TRAN$    ':

     ; READ VOLUME LABEL

                          STATE$
                          TRAN$    $STRNG,RUIC,SETLAB
                          TRAN$    $LAMDA

     ; READ UIC

                          STATE$   RUIC
                          TRAN$    !UIC

     ; SCAN FOR OPTIONS AND END OF LINE

                          STATE$   OPTS
                          TRAN$    $EOS,$EXIT
                          TRAN$    '/

                          STATE$
                          TRAN$    "ALLOC",ALC,,UF.ALL,$FLAGS
                          TRAN$    "PRO",PRO,,UF.PRO,$FLAGS
```

```
; SET ALLOCATION

                  STATE$  ALC
                  TRAN$   '=

                  STATE$
                  TRAN$   $NUMBR,OPTS,SETALC

; PROTECTION

                  STATE$  PRO
                  TRAN$   '=

                  STATE$
                  TRAN$   '[,,IGROUP

                  STATE$  SPRO
                  TRAN$   '],OPTS,ENDGRP
                  TRAN$   <',>,SPRO,NXGRP
                  TRAN$   'R,SPRO,SETRP
                  TRAN$   'W,SPRO,SETWP
                  TRAN$   'E,SPRO,SETEP
                  TRAN$   'D,SPRO,SETDP

; SUBEXPRESSION TO READ AND STORE UIC

                  STATE$  UIC
                  TRAN$   '[

                  STATE$
                  TRAN$   $NUMBR,,SETGN

                  STATE$
                  TRAN$   <',>

                  STATE$
                  TRAN$   $NUMBR,,SETPN

                  STATE$
                  TRAN$   '],$EXIT

                  STATE$

; STATE TABLE SIZE: 60 WORDS
; KEYWORD TABLE SIZE:     8 WORDS
; KEYWORD POINTER SPACE:  3 WORDS

 .SBTTL           ACTION ROUTINES FOR THE COMMAND LINE PARSER

; DEVICE NAME CHAR 1

SETDV1::MOVB      .PCHAR,$UDEV
                  RETURN

; DEVICE NAME CHAR 2

SETDV2::MOVB      .PCHAR,$UDEV+1
                  RETURN

; UNIT NUMBER

SETUNT::MOV       .PNUMB,$UUNIT
                  RETURN
; VOLUME LABEL
```

```
SETLAB::MOV        .PSTCN,$UVNML
                   MOV      .PSTPT,$UVNAM
                   RETURN

; PPN - GROUP NUMBER

SETGN::            MOVB     .PNUMB,$UUIC+1
                   BR       TSTPPN

; PPN - PROGRAMMER NUMBER

SETPN::            MOVB     .PNUMB,$UUIC
TSTPPN:            TST      .PNUMH          ; CHECK FOR 0 HIGH ORDER
                   BNE      10$
                   TSTB     .PNUMB+1        ; CHECK FOR BYTE VALUE
                   BEQ      20$
10$:               ADD      #2,(SP)         ; BAD VALUE - REJECT TRANSITION
20$:               RETURN

; NUMBER OF ENTRIES TO ALLOCATE

SETALC::MOV        .PNUMB,$UALL
                   RETURN

; SET PERMISSIONS
; INITIALIZE

IGROUP::MOV        #4,GRCNT

; MOVE TO NEXT PERMISSIONS CATEGORY

NXGRP::            SEC                      ; FORCE ONES
                   ROR      $UPRO
                   ASR      $UPRO           ; SHIFT TO NEXT GROUP
                   ASR      $UPRO
                   ASR      $UPRO
                   DEC      GRCNT           ; COUNT GROUPS
                   BGE      30$             ; TOO MANY IS AN ERROR
BADGRP:            ADD      #2,(SP)         ; IF SO, REJECT TRANSITION
30$:               RETURN

; SET READ PERMIT

SETRP::            BIC      #FP.RDV*10000,$UPRO
                   RETURN

; SET WRITE PERMIT

SETWP::            BIC      #FP.WRV*10000,$UPRO
                   RETURN

; SET EXTEND PERMIT

SETEP::            BIC      #FP.EXT*10000,$UPRO
                   RETURN

; SET DELETE PERMIT

SETDP::            BIC      #FP.DEL*10000,$UPRO
                   RETURN

; END OF PROTECTION SPEC
```

```
ENDGRP::TST         GRCNT   ; CHECK THE GROUP COUNT
                    BNE     BADGRP          ; MUST HAVE 4
                    RETURN

                    .END    UFD
```

7.6.2 Using Subexpressions and Rejecting Transitions

The following example is an excerpt from a state table that parses a string in which the first character is interpreted as a quote character. This typical construction occurs in many editors and programming languages. The action routines associated with the state table return the byte count and address of the string in the locations QSTC and QSTP. The quoting character is returned in location QCHAR.

```
; MAIN LEVEL STATE TABLE
;
; PICK UP THE QUOTE CHARACTER

                    STATE$  STRING
                    TRAN$   $ANY,,SETQ

; ACCEPT THE QUOTED STRING

                    STATE$
                    TRAN$   !QSTRG,,SETST

; GOBBLE UP THE TRAILING QUOTE CHARACTER

                    STATE$
                    TRAN$   $ANY,NEXT,RESET

; SUBEXPRESSION TO SCAN THE QUOTED STRING
; THE FIRST TRANSITION WILL MATCH UNTIL IT IS REJECTED
; BY THE ACTION ROUTINE

                    STATE$  QSTRG
                    TRAN$   $ANY,QSTRG,TESTQ
                    TRAN$   $LAMDA,$EXIT
                    STATE$

; ACTION ROUTINES
;
; STORE THE QUOTING CHARACTER

SETQ:               MOVB    .PCHAR,QCHAR
                    INCB    .PFLAG          ; TURN OFF SPACE FLUSH
                    RETURN

; TEST FOR QUOTING CHARACTER IN THE STRING

TESTQ:              CMPB    .PCHAR,QCHAR
                    BNE     10$
                    ADD     #2,(SP)         ; REJECT TRANSITION ON MATCH
10$:                RETURN

; STORE THE STRING DESCRIPTOR

SETST:              MOV     .PSTPT,QSTP
                    MOV     .PSTCN,QSTC
                    RETURN
```

```
     ; RESET THE SPACE FLUSH FLAG

     RESET:              DECB    .PFLAG
                         RETURN
```

7.6.3 Using Subexpressions to Parse Complex Command Lines

The following excerpt from a state table shows how subexpressions are used to parse complex command lines.

The state table accepts a number followed by a keyword qualifier. Depending on the keyword, the number is interpreted as either octal or decimal. The binary value of the number is returned in the tagged NUMBER. The following types of strings are accepted:

```
     10/OCTAL
     359/DECIMAL
     77777/OCTAL

     ; MAIN STATE TABLE ENTRY - ACCEPT THE EXPRESSION AND
     ; STORE ITS VALUE

               STATE$
               TRAN$    !ONUMB,NEXT,SETNUM
               TRAN$    !DNUMB,NEXT,SETNUM

     ; SUBEXPRESSION TO ACCEPT OCTAL NUMBER

               STATE$   ONUMB
               TRAN$    $NUMBR

               STATE$
               TRAN$    '/

               STATE$
               TRAN$    "OCTAL",$EXIT

     ; SUBEXPRESSION TO ACCEPT DECIMAL NUMBER

               STATE$   DNUMB
               TRAN$    $DNUMB

               STATE$
               TRAN$    '/

               STATE$
               TRAN$    "DECIMAL",$EXIT
               STATE$

     ; ACTION ROUTINE TO STORE THE NUMBER

     SETNUM:   MOV      .PNUMB,NUMBER
               MOV      .PNUMH,NUMBER+2
               RETURN
```

The contents of .PNUMB and .PNUMH remain undisturbed by all state transitions except the $NUMBR and $DNUMB types.

Because of the way in which subexpressions are processed, calls to action routines from within subexpressions must be handled with care.

When a subexpression is encountered in a transition, TPARS saves its current context and calls itself, using the label of the subexpression as the starting state. If the subexpression parses successfully and returns by means of $EXIT, the transition is taken and control passes to the next state. If the subexpression encounters a syntax error, TPARS restores the saved context and tries to take the next transition in the state.

However, TPARS provides no means for resetting original values changed by action routines that were called by subexpressions. Therefore, action routines called from subexpressions should store results in an intermediate area. Data in this intermediate area can then be accessed by an action routine called from the primary level of the state table.

CHAPTER 8

SPOOLING

FCS provides facilities at both the macro and subroutine level to queue files for subsequent printing; thus, your task can queue a print job. There are several ways for your task to spool output for printing but you cannot control the printing from within your task as you can with the PRINT command. You can, however, use the DCL SET QUEUE command to alter the attributes of the print job once the job appears in the queue.

8.1 PRINT$ MACRO

A task issues the PRINT$ macro 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. The file is placed in the default queue PRINT. If the device is not specified, LP: is used.

The file to be spooled must be open when the PRINT$ macro is issued. Once the file is queued, PRINT$ closes the file. Error returns differ from normal FCS conventions, and are described in Section 8.3.

The PRINT$ macro call has the following format:

Format

```
PRINT$ fdb,err,,dev,unit,pri,forms,copies,presrv
```

Parameters

fdb

The address of the associated FDB. This parameter need not be present if the address of the associated FDB is already in R0.

err

The address of an optional, error-handling routine that you code. See Section 8.3.

The following parameters are not applicable to RSX-11M/M-PLUS and Micro/RSX: dev, unit, pri, forms, copies, presrv

8.2 .PRINT SUBROUTINE

Your task can open a file on disk, send output to the disk, and close the file either by using the PRINT$ macro call or by calling the .PRINT subroutine to spool the output. The .PRINT subroutine is called to queue a file for printing. R0 must contain the address of the associated FDB. The file must be open when .PRINT is called. The file is placed in the default queue PRINT and then the .PRINT routine closes the file. One copy of the file is printed on the LP: device. In your task, it may be preferable to call the .PRINT subroutine if the routine resides in FCSRES. Using the PRINT$ macro causes all the code of .PRINT to appear in your task each time it is used.

Section 8.3 describes error handling for the .PRINT file control routine.

8.2.1 Opening a File on Disk and Using the PRINT Command

As stated in the opening of this section, your task can open a file on disk, send output to that disk, and close the file. When the task exits, the PRINT command can print the file. This is the only method that gives you access to the PRINT command qualifier. Other than using the qualifier, your task can use PRINT$ or .PRINT; you then wait until the job is in the queue and alter its attributes with the SET QUEUE command.

If you run your task from an indirect command file or batch job that includes a PRINT command after the task exits, the difference between spooling from within a task or from outside it is negligible.

You can use the SPWN$ directive in the task to issue the PRINT command. (Refer to the *RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual*.)

8.2.2 Opening a File on LP:

Your task can use the OPEN$ macro to name the output device. FCS opens the file on pseudo device SP0:. The file is placed in the device-specific queue for the device you named. When your task has finished writing to this file, close it with a CLOSE$ macro. The file is deleted after it is printed. This file does not remain in any directory but is identified by a file-ID number labeled FID in the SHOW QUEUE display.

8.3 ERROR HANDLING

The error returns provided with PRINT$ and .PRINT differ from the standard FCS error returns. Unlike FCS error returns, PRINT$ and .PRINT 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 (for example, the PRINT$ macro cannot close the file), the Carry 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 Carry bit is set and the directive status word contains an error code. Directive status word error codes are listed in the *RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual.*

Normally, once you determine that the C-bit is set, any error routine that you code should first test F.ERR and then test the directive status word error code.

APPENDIX A

FILE DESCRIPTOR BLOCK

A File Descriptor Block (FDB) contains file information that is used by File Control Services (FCS) and the file control primitives. The layout of an FDB is illustrated in Figure A-1. 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 you refer 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, whereas the offset names remain constant.

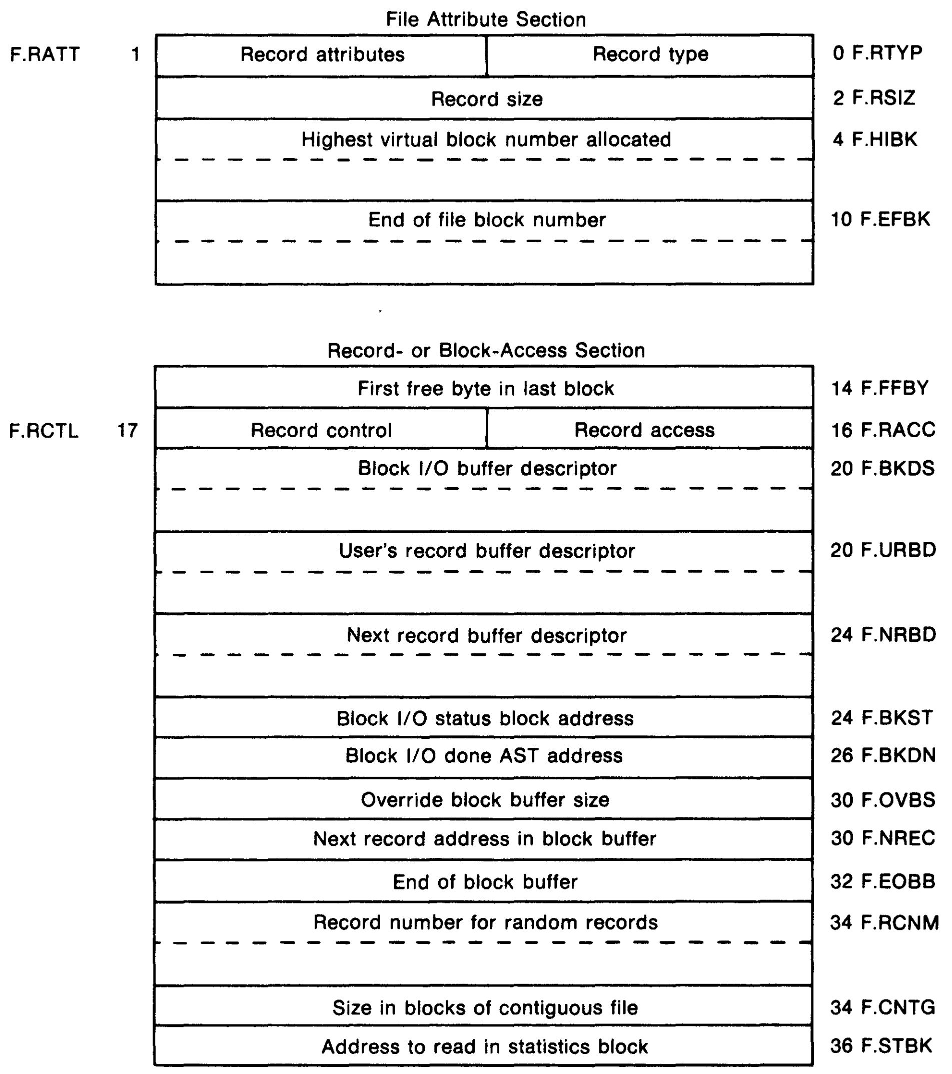

File Attribute Section
F.RATT 1
Record attributes
Record type
0 F.RTYP
Record size
2 F.RSIZ
Highest virtual block number allocated
4 F.HIBK
End of file block number
10 F.EFBK
Record- or Block-Access Section
First free byte in last block
14 F.FFBY
F.RCTL 17
Record control
Record access
16 F.RACC
Block I/O buffer descriptor
20 F.BKDS
User's record buffer descriptor
20 F.URBD
Next record buffer descriptor
24 F.NRBD
Block I/O status block address
24 F.BKST
Block I/O done AST address
26 F.BKDN
Override block buffer size
30 F.OVBS
Next record address in block buffer
30 F.NREC
End of block buffer
32 F.EOBB
Record number for random records
34 F.RCNM
Size in blocks of contiguous file
34 F.CNTG
Address to read in statistics block
36 F.STBK
ZK-3010/1-84

Figure A-1 File Descriptor Block Format

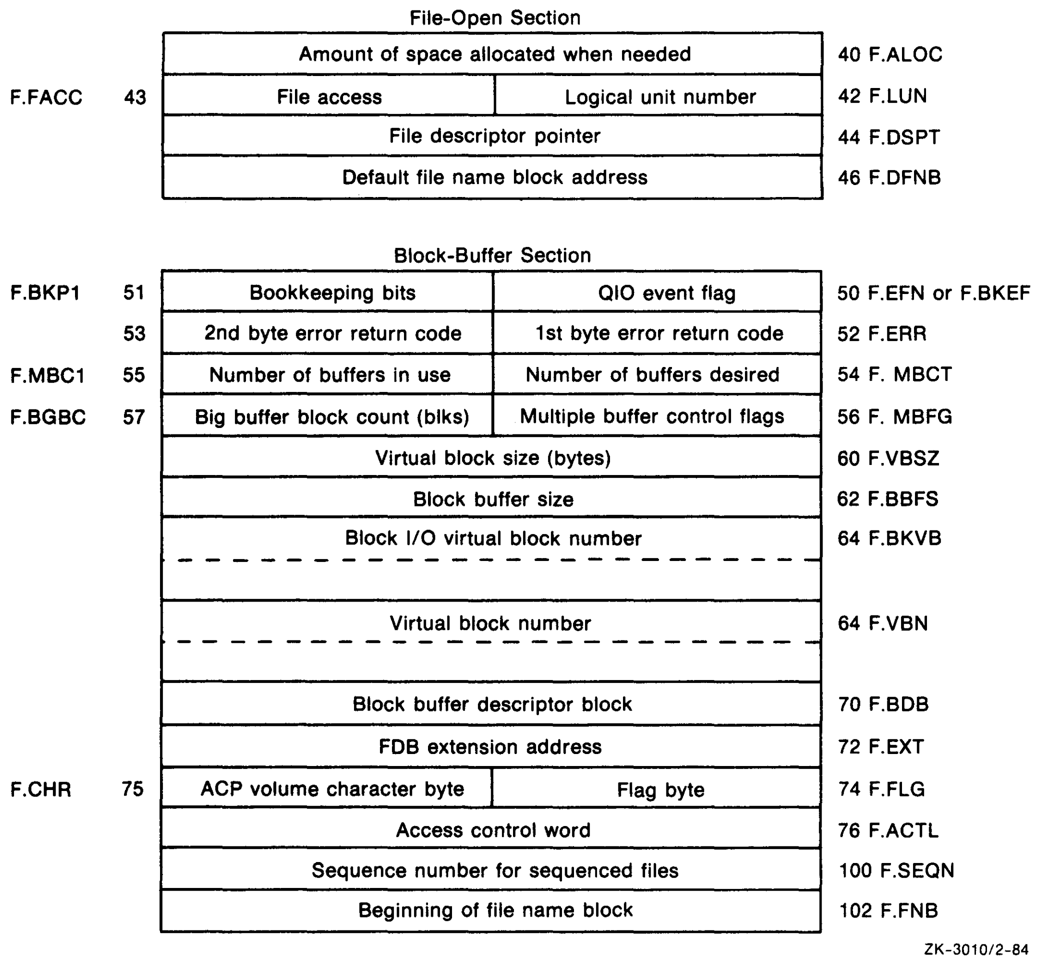


Figure A-1 (Cont.) File Descriptor Block Format

Table A-1
FDB Offset Definitions

Symbolic Offset Name	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:
		
		F.RTYP = 1 to indicate fixed-length records (R.FIX).
		F.RTYP = 2 to indicate variable-length records (R.VAR).
		F.RTYP = 3 to indicate sequenced records (R.SEQ).
		F.RTYP = 4 to indicate stream ASCII records (R.STM).

(Continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
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 = 1 to indicate the "print file format" (FD.PRN). FCS allows this attribute but does not interpret the format word. 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. The format of the block number is high-order word followed by low-order word.
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.

(Continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
		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), for example, a teletypewriter or line printer; a value of 0 indicates a block-oriented device, for example, 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. Bit 4 = 1 to indicate a single directory device (FD.SDI). An MFD is used, but no UFDs are present. Bit 5 = 1 to indicate a block-oriented device that is inherently sequential in nature (FD.SQD), such as magnetic tape. A record-oriented device is assumed to be sequential in nature; therefore, this bit is not set for such devices. Bit 6 = 1 to indicate input spooling (FD.ISP). Bit 7 = 1 to indicate output spooling (FD.OSP).

(Continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
		Bit 8 = 1 for DIGITAL use (FD.PSE).
		Bit 9 = 1 for DIGITAL use (FD.COM).
		Bit 10 = 1 for DIGITAL use (FD.F11).
		Bit 11 = 1 for DIGITAL use (FD.MNT).
F.BKDS or	4	Contains the block I/O buffer descriptor.
F.URBD		Contains the user record buffer descriptor.
F.NRBD	4	Contains the next record buffer descriptor. The record buffer descriptor contains the size of the buffer in the first word and the address of the buffer in the second word.
or F.BKST and	2	Contains the address of the I/O status block for block I/O.
F.BKDN	2	Contains the address of the AST service routine for block I/O.
F.OVBS or	2	Override block buffer size. This field has meaning only before the file is opened.
F.NREC	2	Contains the address of the next record in the block.
F.EOBB	2	Contains a value defining the end-of-block buffer.
F.RCNM or	4	Contains the number of the record for random access operations. The format of the record number is the high-order word followed by the low-order word.
F.CNTG and	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 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 your program.

(Continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
F.ALOC	2	Contains the number of blocks to be allocated when the file must be extended. 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 reading 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 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). If Bit 3 above is 0: Bit 6 = 1 if an existing file is being appended (FA.APD). If Bit 3 above is 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	1	Contains the block I/O event flag.
F.EFN		Contains the record I/O event flag.

(Continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
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 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.
F.MBC1	1	Indicates the actual number of buffers currently in use if the multibuffering version of FCS is in use.
F.MBFG	1	Multibuffering flag word. Contains either one of the multibuffering 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 if the big-buffer version of FCS is in use. Buffer offset for reading ANSI magnetic tape in record mode.
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 virtual block number in the user program for block I/O.
F.VBN		Contains the virtual block number. The format of the virtual block number is the high-order word followed by the low-order word.

(Continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
F.BDB	2	Contains the address of the block buffer descriptor block. This location always contains a nonzero value if the file is open and 0 if the file is closed.
F.EXT	2	Address of FDB extension.
F.FLG	1	Flag byte.
F.CHR	1	Volume characteristics byte. Bit 0 = 1 to indicate ANSI magnetic tape formats D or F.
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. 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.SEQN	2	Contains the sequence number for sequenced records.
F.FNB	-	The symbolic offset of the beginning 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 you refer 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 change from release to release, whereas the offset names remain constant.

Table B-1
Filename Block Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
N.FID	6	File identification field
N.FNAM	6	File name field; specified as nine characters that are stored in Radix-50 format
N.FTYP	2	File type field; specified as three characters that 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.)

(Continued on next page)

Table B-1 (Cont.)
Filename Block Offset Definitions

Symbolic Offset Name	Size (in bytes)	Contents
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 File Descriptor Block (FDB) and their significance are described in Table B-2.

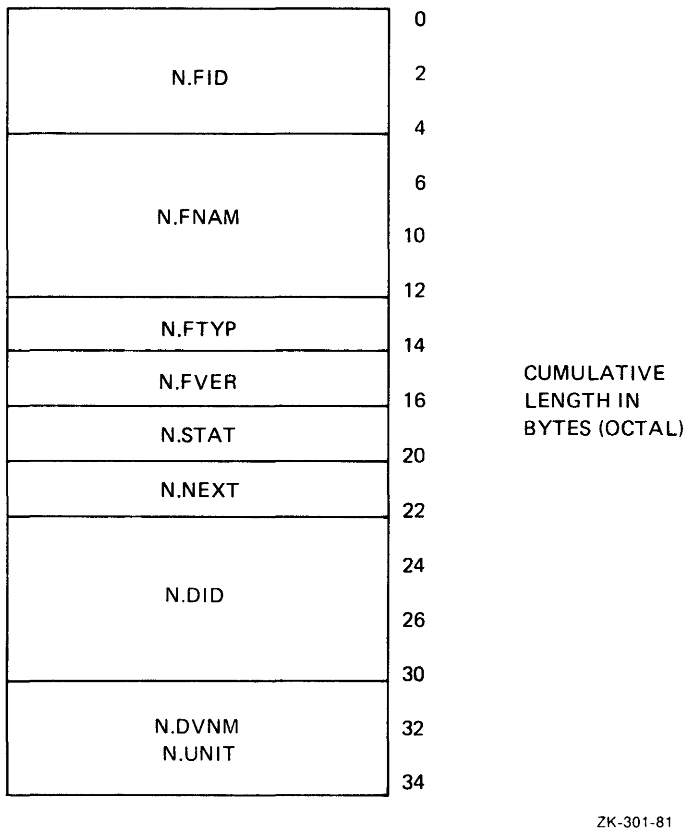


Figure B-1 Filename Block Format

Table B-2
Filename Block Status Word (N.STAT)

Symbolic Offset Name	Value (in octal)	Meaning
NB.VER [1]	1	Set if explicit file version number is specified
NB.TYP [1]	2	Set if explicit file type is specified
NB.NAM [1]	4	Set if explicit file name 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 file name is specified
NB.DIR [1]	100	Set if explicit directory string (UIC) is specified
NB.DEV [1]	200	Set if explicit device name string is specified
NB.SD1 [2]	400	Set if group portion of UIC contains wildcard specification
NB.SD2 [2]	1000	Set if owner portion of UIC contains wildcard specification
NB.ANS	2000	Set if file name is in ANSI format.
NB.WCH	4000	Set if wild character processing is required.

NOTE

Other bits are set as required by FCS and PIP for processing.

1. Indicates bits that are set if the associated information is supplied through an ASCII dataset descriptor.

2. Although NB.SD1 and NB.SD2 are defined, they are neither set nor supported by FCS.

The filename block format for ANSI magnetic tape file names is shown in Figure B-2.

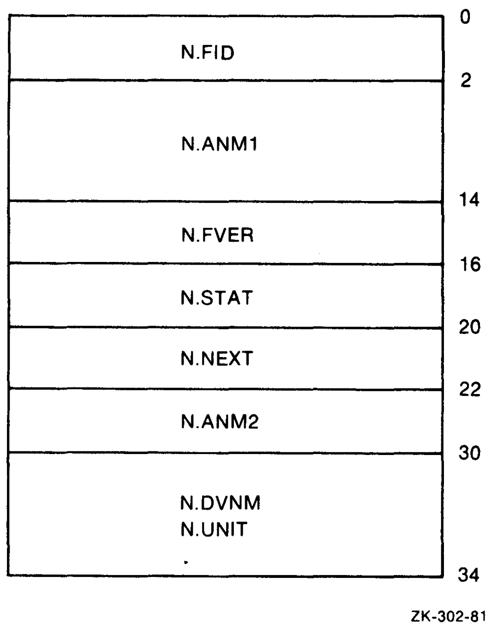


Figure B-2 ANSI Filename Block Format

The filename block offset definitions for ANSI magnetic tape are shown in Table B-3.

Table B-3
Filename Block Offset Definitions for ANSI Magnetic Tape

Symbolic Offset Name	Size (in bytes)	Definition
N.FID	2	File identification field
N.ANM1	12	First 12 bytes of ANSI filename string
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.ANM2	6	Remainder of the ANSI filename string
N.DVNM	2	ASCII device name field
N.UNIT	2	Unit number field (binary)

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 _RSX-11M/M-PLUS and Micro/RSX Executive Reference Manual._

Table C-1
Summary of I/O-Related System Directives

Directive	Function and Parameters
ALUN$	Assign Logical Unit Number -- Assigns a logical unit number to a physical device. **Syntax** ALUN$ lun,dev,unt lun = Logical unit number dev = Physical device name (two ASCII characters) unt = Physical device unit number
GLUN$	Get Logical Unit Number Information -- Fills a 6-word buffer with information about a physical unit to which the LUN is assigned. **Syntax** GLUN$ lun,buf lun = Logical unit number buf = Address of a 6-word buffer in which the LUN information is to be stored
GMCR$	Get MCR Command Line -- Transfers an 80-byte MCR command line to the task issuing GMCR$. No parameters are required in this Directive.
QIO$	Queue I/O Request -- Places an I/O request in the device queue associated with the specified logical unit number.

(Continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

Directive	Function and Parameters
	Syntax QIO$ fnc,lun,efn,pri,isb,ast,prl fnc = I/O function code lun = Logical unit number efn = Event flag number pri = Priority of the request (ignored but must be present) 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>
QIOW$	Queue I/O Request and Wait -- Places an I/O request in the device queue associated with the specified logical unit number. The Executive suspends the task until the specified event flag is set. **Syntax** QIOW$ fnc,lun,efn,pri,isb,ast,prl fnc = I/O function code lun = Logical unit number efn = Event flag number pri = Priority of the request (ignored but must be present) 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>
RCST$	Receive Data or Stop -- Instructs the system to dequeue a 13-word data block for the task issuing RCST$; the data block was queued for the task with a Send Data directive (SDAT$) or a Send, Request, and Connect directive (SDRC$).

(Continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

Directive	Function and Parameters
	Syntax `RCST$ tname,buf` tname = Name of the sending task (if not specified, data may be received from any task) buf = Address of a 15-word buffer to receive the sender task name and data
RCVD$	Receive Data -- Receives a 13-word data block that has been queued (FIFO) by a Send Data directive (see SDAT$ and SDRC$ which follow). **Syntax** `RCVD$ tsk,buf` tsk = Name of the sending task buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block)
RCVX$	Receive Data or Exit -- Receives a 13-word data block if queued by a Send Data directive (see SDAT$ and SDRC$ following), or the task exits if no data is queued. **Syntax** `RCVX$ tsk,buf` tsk = Name of the sending task (if not specified, data may be received from any task) buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block)

SDAT$	Send Data -- Queues (FIFO) a 13-word block of data for a task to receive, and declares a significant event.
	Syntax
	`SDAT$ tsk,buf,efn`
	tsk = Name of the receiving task
	buf = Address of the 13-word data buffer
	efn = Event flag number

SDRC$	Send, Request, and Connect -- Executes a Send Data directive to the specified task, requests the Executive to activate the task if it is not already active, and then connects to the task.

(Continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

Directive	Function and Parameters
	Syntax SDRC$ tname,buf,efn,east tname = Target task name of the offspring task to be connected buf = Address of a 13-word send buffer efn = The event flag to be cleared on issuance and set when the offspring task exits or emits status east = Address of an AST routine to be called when the offspring task exits or emits status
SDRP$	Send Data Request and Pass Offspring Control Block -- Sends a data packet for the specified task, chains to the requested task, and requests it if it is not already active. **Syntax** SDRP$ task,bufadr,buflen,efn,flag task = Name of the task to be chained to bufadr = Address of the buffer to be given to the requested task buflen = Length of the buffer to be given to the requested task efn = Event flag flag = Flags byte controlling the execution of this directive. The flag bits are defined as follows: SD.REX = 128. Force this task to exit upon successful completion of this directive SD.RAL = 1. Pass all connections to the requested task (default is pass none). If you specify this flag, do not specify the parent task name. NOTE The target task may not be a CLI task.

(Continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

Directive	Function and Parameters
	SD.RNX = 2. Pass the first connection in the queue, if there is one, to the requested task. If you specify this flag, do not specify the parent task name.
SMSG$	Send Message -- Creates and sends a formatted data packet to a system-defined target task. The only valid target for the Send Message directive is the Error Logger. The task that issues the SMSG$ directive must be privileged. **Syntax** SMSG$ tgt,buf,len,<pri,...,prn> tgt = Target identifier buf = Address of optional data buffer len = Length in bytes of optional data buffer <pri,...,prn> = Target-specified parameter list
VRCD$	Variable Receive Data -- Instructs the system to dequeue a variable-length data block for the task issuing VRCD$. The block was queued by the Variable Send Data directive. If you specify the sending task, only data sent by that task is received. **Syntax** VRCD$ task,bufadr,buflen task = Sender task name bufadr = Buffer address buflen = Buffer size in words (256. words maximum). The default is 13. words. The first two words are the sender task name. The data block follows.
VRCS$	Variable Receive Data or Stop -- Instructs the system to dequeue a variable-length data block for the task issuing VRCS$. The block was queued by a Variable Send Data directive. If there is no packet, the task issuing VRCS$ is stopped. The sending task is expected to issue an Unstop directive after sending the data. When you specify a sender task, only data sent by that task is received. **Syntax** VRCS$ task,bufadr,buflen task = Sender task name

(Continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

Directive	Function and Parameters
	bufadr = Buffer address buflen = Buffer size in words (256. words maximum). The default is 13. words. The first two words are the sender task name. The data block follows.
VRCX$	Variable Receive Data or Exit -- Instructs the system to dequeue a variable-length data block for the issuing task. The data block was queued for the task by a Variable Send Data directive. If you specify a sender task, only data sent by that task is received. If no data has been sent to the task issuing VRCX$, the task exits. **Syntax** VRCX$ task,bufadr,buflen task = The name of the sending task bufadr = The buffer address buflen = The buffer length (a maximum of 256. words). The default is a minimum of 13. words. The first two words are the sender task name.
VSDA$	Variable Send Data -- Instructs the system to queue a variable-length data block for the specified task to receive. If you specify an event flag, a significant event is declared when the directive executes successfully. **Syntax** VSDA$ task,bufadr,buflen task = Receiving task name bufadr = Buffer address buflen = Buffer size in words (a maximum of 256. words). The default is 13. words.
VSRC$	Variable Send, Request, and Connect -- Performs a Variable Send Data to the specified task, requests the task if it is not already active, and then connects to the task. **Syntax** VSRC$ tname,buf,buflen,efn,east,esb tname = Target task name of the offspring task to be connected

(Continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

Directive	Function and Parameters
	buf = Address of send buffer
	buflen = Length of the buffer (a maximum of 256. words). The default is 13. words.
	efn = The event flag cleared when the directive is issued and set when the offspring task exits or emits status
	east = Address of an AST routine to be called when the offspring task exits or emits status
	esb = Address of an 8-word status block to be written when the offspring task exits or emits status: Word 0 -- Offspring task exit status Word 1 -- TKTN abort code Words 2-7 -- Reserved

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,80.      ;Init record attributes
          FDOP$A  OUTLUN,,OFNAM    ;Init file open section
FDBIN:    FDBDF$                   ;Allocate space for input FDB
          FDRC$A  ,RECBUF,80.      ;Init record attributes
          FDOP$A  INLUN,,IFNAM     ;Init file open section
RECBUF:   .BLKB   80.              ;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(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   CRCOPA            ;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.
CFDSPT: .WORD    0,0               ;Device descriptor
        .WORD    0,0               ;Directory descriptor
        .WORD    ONAM$Z,ONAM       ;Filename descriptor
IFDSPT: .WORD    0,0               ;Device descriptor
        .WORD    0,0               ;Directory descriptor
        .WORD    INAMSZ,INAM       ;Filename descriptor
ONAM:   .ASCII   /OUTPUT.DAT/
        ONAMSZ=.-ONAM
        .EVEN
INAM:   .ASCII   /INPUT.DAT/
        INAMSZ=.-INAM
        .EVEN
START:  FINIT$                     ;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.
CFDSPT: .WORD   0,0             ;Device descriptor
        .WORD   0,0             ;Directory descriptor
        .WORD   ONAM$Z,ONAM     ;Filename descriptor
IFDSPT: .WORD   0,0             ;Device descriptor
        .WORD   0,0             ;Directory descriptor
        .WORD   INAMSZ,INAM     ;Filename descriptor
ONAM:   .ASCII  /OUTPUT.DAT/
        ONAMSZ=.-ONAM
        .EVEN
INAM:   .ASCII  /INPUT.DAT/
        INAMSZ=.-INAM
        .EVEN
START:  FINIT$                  ;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 ([0,0]INDEXF.SYS) of a Files-11 volume consists of virtual blocks, starting with virtual block 1, the bootstrap block. Virtual block 2 is the home block. The structure of an 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
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.

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 (UFDs).
Checkpoint File Header	This block identifies the file that is used for the checkpoint areas for all checkpointable tasks. In RSX-11, a task can also have checkpoint space in the task image itself.

The remainder of the index file consists of file header blocks for user files, as shown in the list 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.DFPR
6	Reserved	--
1	Default number of retrieval pointers in a window	H.WISZ
1	Default number of blocks to extend files	H.FIEX
1	Number of entries in directory LRU	H.LRUC
11.	Available space	--
2	Checksum of words 0-28	H.CHK1
14.	Creation date and time	H.VDAT
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

(Continued on next page)

Table F-1 (Cont.)
File Header Block

Area	Size (in bytes)	Content	Offset
	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	File name (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
	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

(Continued on next page)

Table F-1 (Cont.)
File Header Block

Area	Size (in bytes)	Content	Offset
	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
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; for example, the index file is number 1, the storage bit map is file number 2, and so forth.
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 reused.
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 UICs 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, you specify 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. When the file is extended (for example, by a WRITE$ or PUT$ macro), bit UC.CON is cleared whether or not the extension requests contiguous blocks. 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 by FCS, although RMS does use them.

F.2 IDENTIFICATION AREA

The information in the identification area of the file header block consists of the following:

File name	The file's creator specifies a file name 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 0 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 (two bytes, three bytes, and two bytes, respectively). This date is meaningful only if the revision number is a nonzero value.

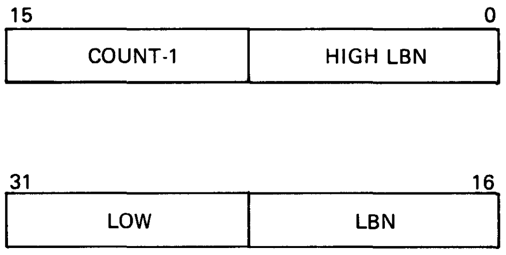
Revision time	Six bytes are used to record the time at which the file was last revised. This information is recorded in ASCII form in the format hour, minute, and second (two bytes each).
Creation date	The date on which the file was created is kept in a 7-byte field having the same format as that of 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 that of 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 that of 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:



ZK-303-81

NOTE

The remaining paragraphs in this appendix apply to RSX-11M/M-PLUS and Micro/RSX systems that support the multiheader version of F11ACP.

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 numbers the headers of the file sequentially, starting with a 0 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 you.

APPENDIX G

SUPPORT OF ANSI MAGNETIC TAPE STANDARD

This appendix defines the ANSI magnetic tape labeling standard, which is a level three implementation of the ANSI standard Magnetic Tape Labels and File Structure for Information Interchange (X3.27-1978). The exceptions are that ANSI does not support spanned records and that DIGITAL's tape system does not support user-supplied labels. User-supplied labels may appear on a tape; however, they are accessible to application programs through the unlabeled tape feature only.

G.1 VOLUME AND FILE LABELS

Tables G-1, G-2, G-3, and G-4 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 ANSI "a" character. An "a" character is defined by the ANSI standard as any of the uppercase letters A through Z, the digits 0 through 9, and the following special characters: space ! " % & ' () * + , - . / : ; <=> ?
11	Accessibility	1	Any ANSI "a" character. A space indicates no restriction. You can specify the "a" character with the

(Continued on next page)

Table G-1 (Cont.)
Volume Label Format

Character Position	Field Name	Length (in bytes)	Contents
			/VOLUME_ACCESSABILITY:"c" qualifier in the DCL INITIALIZE commands. Any ANSI "a" character is allowed. The default character is a space. Refer to the RSX-11M/M-PLUS MCR Operations Manual or RSX-11M/M-PLUS Command Language Manual for more information on INITIALIZE.
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	3

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. A numeric 1 (position 51)

The system identification consists of the following character sequence:

D%x

x The machine code, which 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. The /OWNER switch to the MCR INI command allows you to overwrite these characters. The /OWNER="owner" qualifier to the DCL INITIALIZE command allows you to overwrite these characters. (Refer to the

RSX-11M/M-PLUS MCR Operations Manual and the RSX-11M/M-PLUS Command Language Manual for more information.) In the case of tapes produced on PDP-11 systems, the default system identification is D%B and the volume protection code and UIC are interpreted as described below.

The volume protection code in positions 41 through 44 defines access protection for the volume for four classes of users. Each class of user has access privileges specified in one of the four columns, as follows:

Position	Class
41	System (UIC no greater than [7,255])
42	Owner (group and member numbers match)
43	Group (group number matches)
44	World (any user not in one of the above)

One of the following access codes can be specified for each character position:

Code	Privilege
0	No access
1	Read access 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 numeric 1.

The following is an example of the owner identification field.

Owner identifier - D%B14100631461

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 [063,146].

G.1.2 User Volume Labels

User volume labels are never written or passed back to you. 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 non-DIGITAL systems, the conventions listed above should be followed. If they are not, refer to Section G.1.3.1.

Tables G-2, G-3, and G-4 describe the HDR1, HDR2, and HDR3 labels, respectively.

Table G-2
File Header Label (HDR1)

Character Position	Field Name	Length (in bytes)	Contents
1-3	Label identifier	3	HDR
4	Label number	1	1
5-21	File identifier	17	Any ANSI "a" character. See Table G-1.
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

(Continued on next page)

Table G-2 (Cont.)
File Header Label (HDR1)

Character Position	Field Name	Length (in bytes)	Contents
61-73	System code	13	The three letters DEC, followed by the name of the system that produced the volume. See Section G.1.1.1. Examples: DECFILE11A DECSYSTEM10 Pad name with spaces.
74-80	Reserved	7	Spaces.

Table G-3
File Header Label (HDR2)

Character Position	Field Name	Length (in bytes)	Contents
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.

(Continued on next page)

Table G-3 (Cont.)
File Header Label (HDR2)

Character Position	Field Name	Length (in bytes)	Contents
			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. Supported only on input to Files-11.
53-80	Reserved	28	Spaces.

Table G-4
File Header Label (HDR3)

Character Position	Field Name	Length (in bytes)	Contents
1-3	Label identifier	3	HDR
4	Label number	1	3
5-68	System-dependent	64	File attributes specified at creation time. Each of the 32 bytes of user file attributes is expanded into two hexadecimal characters. The first seven words of this area are a direct image of the first seven words of the FDB when the file is opened. These are the same words in the file attribute section of the File Descriptor Block given in Appendix A. The other nine words are not used by FCS though they are used by RMS. The following list translates the user file attribute bytes to the corresponding hexadecimal character pair.

(Continued on next page)

Table G-4 (Cont.)
File Header Label (HDR3)

Character Position	Field Name	Length (in bytes)	Contents

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

Using the list, the eighth hexadecimal character pair is the expansion of the fifth user file attribute byte, and the fourth user file attribute byte is expanded into the first hexadecimal character pair. The hexadecimal pair is the normal representation of the contents of the byte; that is, if the byte contains a 15., the hexadecimal representation of it is 0F.

Character Position	Field Name	Length (in bytes)	Contents
69-80	Reserved	10	Spaces.

G.1.3.1 **File Identifier Processing by Files-11** - The magnetic tape ACP processes Files-11 type file identifiers as described below. However, if the file name is enclosed in quotes, it is processed as an ANSI file name, all "a" characters are legal, all 17 positions may be used, and the only conversion that takes place is making all lowercase characters into uppercase characters and converting all characters that are not "a" characters to question marks.

At file input, the file identifier is handled as follows:

1. The first nine characters at a maximum are processed by an ASCII to Radix-50 converter. The conversion continues until one of the following occurs:

   a. A conversion failure.

   b. Nine characters are converted.

   c. 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

   If an invalid version number is computed, it will be changed to 1.

At file output, the file identifier is handled as follows:

1. The file name is placed in the first positions in the file identifier field. It can occupy up to nine positions and is followed by a period.

2. The file type of up to three characters is placed after the period. The remaining positions are padded with spaces.

3. The version number is then placed in the generation and generation version number fields, as described in the following formulas:

   a. Generation number $= \left(\frac{\text{version \# - 1}}{100}\right) + 1$

   b. Generation version # $= \left(\text{version \# - 1}\right)$ Modulo 100

NOTE

In both calculations, remainders are ignored.

The following are examples.

Files-11 Version No.	**Generation No.**	**Generation Version No.**
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 3 contain EOV instead of HDR.

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 3 contain EOF instead of HDR.

2. The block count contains the number of data blocks in the file.

G.1.6 User File Labels

User file labels are never written or passed back to you. 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. The file processor can read and process additional sequences.

The minimum block size and fixed-length record size is 18 bytes. The maximum block size is 8192 bytes. FCS adapts to input files of varying block size.

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. You can override these block and record sizes with the MAG command (see Section G.5), and MOUNT command. The MAG command controls block and record size on unlabeled tapes and on ANSI level 1 and 2 tapes.

The meaning of the symbols used in the file structure illustrations is as follows:

1. The asterisk (*) indicates a tape mark. As defined by ANSI, a tape mark is a special control block recorded on magnetic tape to serve as a separator between files and file labels.

2. BOT indicates beginning of tape.

3. EOT indicates end of tape.

4. The comma (,) indicates the physical record delimiter.

G.2.1 Single File Single Volume

```
BOT,VOL1,HDR1,HDR2,HDR3*---DATA---*EOF1,EOF2,EOF3**
```

G.2.2 Single File Multivolume

```
BOT,VOL1,HDR1,HDR2,HDR3*---DATA---*EOV1,EOV2,EOV3**

BOT,VOL1,HDR1,HDR2,HDR3*---DATA---*EOF1,EOF2,EOF3**
```

G.2.3 Multifile Single Volume

```
BOT,VOL1,HDR1,HDR2,HDR3*---DATA---*EOF1,EOF2,EOF3*HDR1,HDR2,HDR3*---DATA
--*EOF1,EOF2,EOF3**
```

G.2.4 Multifile Multivolume

```
BOT,VOL1,HDR1,HDR2,HDR3*--DATA--*EOF1,EOF2,EOF3*HDR1,HDR2,HDR3*--DATA--*
EOV1,EOV2**

BOT,VOL1,HDR1,HDR2,HDR3*--DATA--*EOF1,EOF2,EOF3*HDR1,HDR2,HDR3*--DATA--*
EOF1,EOF2,EOF3**
```

G.3 END-OF-TAPE HANDLING

End-of-tape is handled by the magnetic tape file processor. Files are continued on the next volume provided that the volume is already mounted or mounted upon request. A request for the next volume is printed on CO: (console output pseudo device).

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.

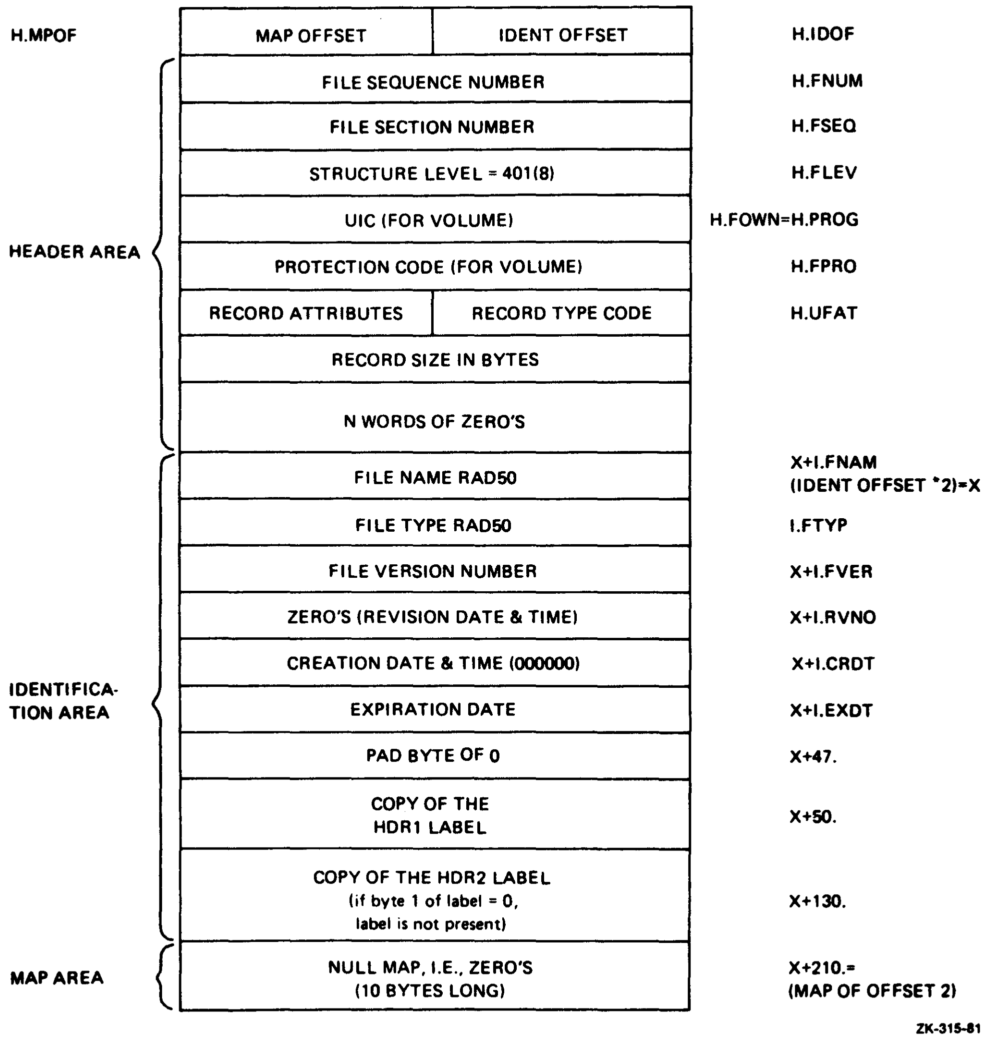


Figure G-1 ANSI Magnetic Tape File Header Block (FCS Compatible)

G.5 THE MAGNETIC TAPE CONTROL TASK

The Magnetic Tape Control Task (MAG) allows you to specify file attributes for unlabeled tapes, provides positioning functions for both unlabeled and ANSI tapes, and allows you to respond to requests for new tapes or volumes without mounting a new tape.

This command can only be used on mounted tapes. The keywords are valid for both unlabeled and ANSI tapes, unless otherwise noted.

The format of the MAG command is as follows:

```
>MAG SET mmnn:/keyword[/keyword[s]]
```

mmnn:

Magnetic tape unit on which the operation is to be performed.

Keywords:

```
/BS=number-of-bytes

/CC=NONE
    LIST
    FORTRAN

/EOF

/EOT

/EOV

/INITIALIZE="volume id"

/POS=number-of-files

/RS=number-of-characters

/REWIND
```

Keyword definitions:

BS	/BS=number-of-bytes. Number of characters (bytes) per block on a mounted tape. You can specify the number of characters in either decimal or octal. To specify a decimal number, terminate it with a period. The default is octal. This block size becomes the default for the tape. FCS uses this value on output when there is no HDR2 label present on an ANSI magnetic tape, and when no other value was specified on creation of the file. FCS reads the file attributes to obtain the block size when opening an existing file on ANSI magnetic tape. The value specified for block size must be greater than 14 (decimal) bytes. There is no maximum block size. FCS pads a block on ANSI magnetic tape to an even length to permit PUT$ operations with odd fixed-length records.
CC	/CC=type of carriage control. The default is /CC=NONE. Valid types of carriage control are as follows: NONE LIST FORTRAN

When reading from an ANSI tape, FCS uses this carriage control if:

- No HDR2 label is present.
- The HDR2 label contains a system identification other than DEC or OS.

If a HDR2 label is present, FCS uses the information in the HDR2 label and ignores the /CC keyword.

This keyword is the only way to specify carriage control for unlabeled tapes.

EOF

/EOF. Causes the ACP to return the I/O status code IE.EOF to the requesting task. You can use this keyword to terminate a request for another tape in a volume set.

This keyword is only valid for ANSI tapes.

EOT

/EOT. Causes the ACP to return the I/O status code IE.EOT to the requesting task. You can use this keyword to terminate a request for another tape in a volume set.

This keyword is only valid for ANSI tapes.

EOV

/EOV. Causes the ACP to return the I/O status code IE.EOV to the requesting task. You can use this keyword to terminate a request for another tape in a volume set.

This keyword is only valid for ANSI tapes.

INITIALIZE

/INITIALIZE="volume id". The volume label to which the tape will be initialized. This keyword allows you to create a new volume to satisfy a request from the ACP for a new output volume for an ANSI tape. The format of the volume identifier is identical to the format of the volume identifier specified for the MOUNT command. Refer to the INITIALIZE command in the RSX-11M/M-PLUS Command Language Manual or the RSX-11M/M-PLUS MCR Operations Manual for details.

This keyword is only valid for ANSI tapes.

POS

/POS=number-of-files. The number of files (tape marks) to be spaced over from the current tape position. For example, /POS=0 means access the current file; /POS=1 means space forward one file from the current position. The number of files may be specified in either decimal or octal. To specify a decimal number, terminate the number with a period. The default is octal.

The number of files specified must be between 0 and 23417(octal).

This keyword is only valid for mounted unlabeled tapes. This keyword is not necessary for labeled tapes, because files can be accessed by name.

If a tape containing ANSI or IBM labels is mounted as unlabeled, the formula to calculate the position of a data file is:

N= (N-1)*3+1

where N is the number of the desired file.

RS — /RS=number-of-characters. The number of characters per record for fixed-length records on tape.

Maximum record size is the block size.

When reading files from an ANSI tape, FCS uses this value for record size when no HDR2 label is present.

This keyword is the only way to specify record size for unlabeled tapes.

REWIND — /REWIND. Specifies that the tape is to be rewound to BOT. For ANSI tapes, this keyword rewinds to the beginning of the volume set.

G.5.1 MAG Command Example

```
>MOU MM0:/NOLABEL/TR=EBCDIC
>MAG SET MM0:/BS=80./RS=80./CC=LIST/REWIND

>! Read the first "file" on the tape to determine the actual tape format

>PIP TI:=MM0:X
VOL1X234870                             NASHUA
HDR1DEC2.KP72132    X2348700010001       80256 802860000000IBM OS/VS 370
HDR2F008000008030DEC2K009/DEC2U009   B   60337

>! This information shows that the tape in fact has labels that
>! resemble ANSI labels (this tape is in the format of
>! another computer manufacturer). The actual block size and record size
>! can be determined from the labels.
>! Use the MAG SET command to set the actual block and record size.

>MAG SET MM0:/BS=800./RS=80.

>! Use the MAG SET /POS command to position to the next file on the tape.
>! You could also position to the file by specifying
>! a position to PIP as follows: PIP TI:=MM0:"POS=R1"
>! where R indicates REWIND and 1 is the number of tape
>! marks to space over.

>MAG SET MM0:/REWIND/POS=1

>! Read the data file. Note that PIP requires a dummy file name, even
>! though the tape is already positioned at the correct file.

>PIP TI:=MM0:X
001     321     03054          MERRIMACK              603
002     456     03060          NASHUA                 603
003     789     02165          NEWTON                 617
004     124     01845          NORTH ANDOVER          617
005     345     64801          JOPLIN                 714
```

G.5.2 MAG Command Error Messages

MAG -- Device not mounted or mounted foreign

Explanation: Specified device must be either a mounted ANSI tape or a mounted unlabeled tape. A foreign tape that is mounted foreign is not known to the magnetic tape ACP and is therefore the same as an unmounted tape. The MAG command only supports mounted tapes.

User Action: Mount the device with the /NOLABEL qualifier if it is an unlabeled tape. Do not use the MAG command with tapes that are mounted foreign.

MAG -- Get command line failure

Explanation: An illegal indirect command file name was specified or MAG could not find the specified indirect command file.

User Action: Check the specification for the indirect command file and reenter the command line.

MAG -- Illegal combination of keywords

Explanation: Keywords were specified that required both an unlabeled tape and an ANSI tape.

User Action: Refer to the keyword descriptions in this chapter to determine which switches require ANSI tapes and which require unlabeled tapes.

MAG -- Illegal file attributes combination

Explanation: A record size was specified that was not less than or equal to the block size.

User Action: Specify a record size less than or equal to the block size.

MAG -- Illegal switch value

Explanation: One of the following:

- /POS value was greater than 9999.
- /BS value was less than 14.
- /RS value was less than 14.

User Action: Determine which value was illegal and retype the command line.

MAG -- Invalid device or unit

Explanation: Specified device does not exist or is not a magnetic tape.

User Action: Determine the correct device or unit and retype the command line.

MAG -- Operation is only valid for mounted ANSI tapes

Explanation: An operation that is only valid for ANSI magnetic tape was attempted on an unlabeled tape.

User Action: Use only valid commands.

MAG -- Operation is only valid for unlabeled tapes

Explanation: An operation that is only valid for unlabeled tapes was attempted on an ANSI tape.

User Action: Use only valid commands.

MAG -- Privilege violation

Explanation: On systems with multiuser protection, only the terminal to which the tape drive is allocated may change the tape characteristics.

User Action: Reenter the command from the terminal that owns the tape drive.

MAG -- Requested operation inconsistent with tape state

Explanation: The user specified /EOF, /EOV, or /EOT and the magnetic tape ACP rejected it.

or

The user specified /INITIALIZE and the ACP rejected it.

User Action: Determine the state of the tape and type only commands that can be performed in the current state.

MAG -- Syntax error

Explanation: The command was specified incorrectly.

User Action: Check the correct syntax as described in this chapter, and then reenter the command line.

G.6 UNLABELED TAPE

A tape that contains no labeling information is called an unlabeled tape. An unlabeled tape contains either blocked or unblocked fixed-length records. When a tape is mounted with the /NOLABEL qualifier on the MOUNT command, FCS and RMS can access records on the tape using standard read and write operations. This is different from mounting a tape foreign (MOU/FOREIGN). When a tape is mounted foreign (or not mounted at all on RSX-11M systems), records on the tape must be accessed directly using the QIO$ operations that are defined for the magnetic tape driver.

G.6.1 Block Size on Tapes Mounted /NOLABEL

Under certain conditions, if a file is written to a tape, its block size will be even and one more than the value specified in the MOUNT command. The conditions where this occurs are as follows:

- The tape is mounted /NOLABEL.
- The mount command specifies an odd record size.
- The mount command specifies an odd block size.

FCS adds the padding character, an octal 136 circumflex (^), to odd-sized byte blocks because of a hardware restriction; some tape drives will not allow an odd number of bytes to be transferred to or from tape. Therefore, blocks of data are padded with the circumflex character so that even blocks of data can be written to tape on any tape drive.

G.6.2 Tape Positioning

Any tape motion before the first read operation must be explicitly requested in one of the following ways:

- The FA.ENB!FA.RWD bit may be set in F.ACTL to request a rewind of the volume set prior to create or find-file operations.
- The MAG SET /POS command may be specified to space forward a specified number of files from 0 to 9999.
- The MAG SET /REWIND command may be specified to rewind the tape to BOT.
- The file may be referenced by the name "POS=[R][nnnn]" where R indicates that the tape is to be rewound, and nnnn is the number of files (tape marks) to space forward. For example, to read the second file on a tape:

```
>PIP TI:=MM:"POS=0001"
```

Each tape mark delimits a file. All positioning operations are in terms of tape marks. If any type of label is present on the tape, it will be treated as a file.

When a file is deaccessed, position within the file is always consistent.

G.6.3 Specifying File Attributes

You can specify the attributes for files to be read from tape in three ways:

1. The MOUNT command (see the RSX-11M/M-PLUS MCR Operations Manual or the RSX-11M/M-PLUS Command Language Manual).
2. Any create operation request. You can issue the create request from within your program by creating a file and closing it without writing any data, or by using the RMS DEFINE utility. The FCS Create routine returns the error code IE.BTP (bad record type) if an attempt is made to set the record type to anything other than fixed length.
3. The MAG SET command (see Section G.5).

G.6.4 Tape Translation

You can request translation for a tape when you mount it with the MOUNT command (see the RSX-11M/M-PLUS MCR Operations Manual or the RSX-11M/M-PLUS Command Language Manual). If you have requested translation, your data buffer (the FCS or RMS block buffer) is translated within your task. Therefore, on a write operation, the data in your task is destroyed.

You can add up to three installation-dependent translation routines to the magnetic tape ACP by adding routines with the following format:

```
USERn:: MOV #TBLPTR,R0          ;n is 1, 2, or 3.
        RETURN
TBLPTR: .WORD INTRAN
        .WORD OUTRAN
INTRAN: <A 256 byte table for input translation>
OUTRAN: <A 256 byte table for output translation>
```

An example of the EBCDIC translation tables that are provided with your system is shown in Section G.6.5.

You must include these translation routines in the Overlay Description Language (ODL) for the magnetic tape ACP when the MTAACP task is built. Comments within the files MTABLD.ODL and MTABLD.CMD indicate where these routines may be added.

G.6.5 Example of EBCDIC Translation Tables

```
EBCDIC::        MOV     #TBLPTR,R0
                RETURN
;
TBLPTR: .WORD   EBCASC
        .WORD   ASCEBC
;
        .NLIST BEX
;
EBCASC: .BYTE   000,001,002,003,040,011,040,177,040,040
        .BYTE   040,013,014,015,016,017,020,021,022,023
        .BYTE   040,040,010,040,030,031,040,040,040,035
        .BYTE   040,037,040,040,034,040,040,012,027,033
        .BYTE   040,040,040,040,040,005,006,007,040,040
        .BYTE   026,040,040,036,040,004,040,040,040,040         ; 50.
        .BYTE   024,025,040,032,040,040,040,040,040,040
        .BYTE   040,040,040,040,133,056,074,050,053,135
        .BYTE   046,040,040,040,040,040,040,040,040,040
        .BYTE   041,044,052,051,073,136,055,057,040,040
        .BYTE   040,040,040,040,040,040,174,054,045,137         ; 100.
        .BYTE   076,077,137,040,040,040,040,040,040,040
        .BYTE   040,140,072,043,100,047,075,042,040,141
        .BYTE   142,143,144,145,146,147,150,151,040,040
        .BYTE   040,040,040,040,040,152,153,154,155,156
        .BYTE   157,160,161,162,040,040,040,040,040,040         ; 150.
        .BYTE   040,176,163,164,165,166,167,170,171,172
        .BYTE   040,040,040,040,040,040,040,040,040,040
        .BYTE   040,040,040,040,040,040,040,040,040,040
        .BYTE   040,040,173,101,102,103,104,105,106,107
        .BYTE   110,111,040,040,040,040,040,040,175,112         ; 200.
        .BYTE   113,114,115,116,117,120,121,122,040,040
        .BYTE   040,040,040,040,134,040,123,124,125,126
        .BYTE   127,130,131,132,040,040,040,040,040,040
        .BYTE   060,061,062,063,064,065,066,067,070,071
        .BYTE   040,040,040,040,040,040                         ; 250.
;
ASCEBC: .BYTE   000,001,002,003,067,055,056,057
        .BYTE   026,005,045,013,014,015,016,017
        .BYTE   020,021,022,023,074,075,062,046
        .BYTE   030,031,077,047,042,035,065,037
        .BYTE   100,132,177,173,133,154,120,175         ; 40
        .BYTE   115,135,134,116,153,140,113,141
        .BYTE   360,361,362,363,364,365,366,367
```

```
                .BYTE   370,371,172,136,114,176,156,157
                .BYTE   174,301,302,303,304,305,306,307         ; 100
                .BYTE   310,311,321,322,323,324,325,326
                .BYTE   327,330,331,342,343,344,345,346
                .BYTE   347,350,351,112,340,117,137,155
                .BYTE   171,201,202,203,204,205,206,207         ; 140
                .BYTE   210,211,221,222,223,224,225,226
                .BYTE   227,230,231,242,243,244,245,246
                .BYTE   247,250,251,300,152,320,241,007
;
                .BYTE   000,001,002,003,067,055,056,057
                .BYTE   026,005,045,013,014,015,016,017
                .BYTE   020,021,022,023,074,075,062,046
                .BYTE   030,031,077,047,042,035,065,037
                .BYTE   100,132,177,173,133,154,120,175         ; 200
                .BYTE   115,135,134,116,153,140,113,141
                .BYTE   360,361,362,363,364,365,366,367
                .BYTE   370,371,172,136,114,176,156,157
                .BYTE   174,301,302,303,304,305,306,307         ; 240
                .BYTE   310,311,321,322,323,324,325,326
                .BYTE   327,330,331,342,343,344,345,346
                .BYTE   347,350,351,112,340,117,137,155
                .BYTE   171,201,202,203,204,205,206,207         ; 300
                .BYTE   210,211,221,222,223,224,225,226
                .BYTE   227,230,231,242,243,244,245,246
                .BYTE   247,250,251,300,152,320,241,007
```

G.7 EXAMPLE USING AN INDIRECT COMMAND FILE TO READ A TAPE

```
.ENABLE QUIET
.ENABLE SUBSTITUTION

.; This command file is invoked with the command
.;      @MTA outspec=Mx:infile
.; and searches a tape mounted unlabeled (which has an ANSI-like structure)
.; for the file "infile" and copies it to outspec.
.;
.; Parse the command line;  OUTSPC gets outspec,
.;                          DEV    gets Mx,
.;                          INFILE gets the file name to find on tape.

.PARSE COMMAN " " OUTSPC COMMAN
.PARSE COMMAN "=" OUTSPC INSPEC
.PARSE INSPEC ":" DEV INFILE
.IF INFILE EQ "" .GOTO NOTMT
.SETS INFILE INFILE+"                 "
.SETS INFILE INFILE[1:17.]
.SETS JUNK DEV[1:1]
.IF JUNK NE "M" .GOTO NOTMT

.;
.; Make a name for the temp file.
.;

.TESTFILE TI:
.PARSE <FILSPC> ":" TMP JUNK
.SETS TMP TMP+".TMP"

.;
.; Always start at the beginning of the tape.
```

```
MAG SET 'DEV':/REWIND

.;
.; Labels have a block and record size of 80.

MAG SET 'DEV':/BS:80./RS:80.
.LOOK:

.;
.; Put the labels in a temp file so Indirect can look at them

PIP 'TMP'='DEV':DUMMY.NAM
.OPENR 'TMP'
.READLB:
.READ LABEL
.IFT <EOF> .GOTO NOSUCH
.SETS LABELT LABEL[1:3]

.;
.; Skip any Volume header labels

.IF LABELT = "VOL" .GOTO READLB
.IF LABELT NE "HDR" .GOTO ILLFMT
.SETS LABELT LABEL[4:4]
.IF LABELT NE "1" .GOTO ILLFMT
.SETS LABELT LABEL[5:21.]

.;
.; If the names do not match, go get the next set of labels.

.IF LABELT NE INFILE .GOTO TRYNXT

.;
.; We have found the file, see if there is a HDR2 with size info.

.READ LABEL
.IFT <EOF> .GOTO READFL
.SETS LABELT LABEL[1:4]
.IF LABELT NE "HDR2" .GOTO READFL

.;
.; Yes, we have a HDR2 label.

.SETS LABELT LABEL[37.:37.]
.SETS CC "LI"
.IF LABELT = "A" .SETS CC "FO"
.IF LABELT = "M" .SETS CC "NO"
.SETS BS LABEL[6:10.]
.SETS RS LABEL[11.:15.]

.;
.; Set up the block size, record size, and carriage control
.; based on what was in HDR2.

MAG SET 'DEV':/BS:'BS'./RS:'RS'./CC:'CC'
.SETS LABELT LABEL[5:5]
.IF LABELT EQ "F" .GOTO READFL
.DISABLE QUIET

!MTA - Warning, Record Format is 'LABELT'; only F Format is fully supported.

.ENABLE QUIET
.READFL:
.CLOSE
```

```
.;
.; Transfer the file.

PIP 'OUTSPC'='DEV':"POS=1"
.GOTO ENDIT
.TRYNXT:
.CLOSE
MAG SET 'DEV':/POS=3
.GOTO LOOK
.ILLFMT:
.DISABLE QUIET
.DISABLE MCR

MTA - Tape is not in a format that I understand.

.GOTO ENDIT
.NOTMT:
.DISABLE QUIET
.DISABLE MCR

MTA - Input file spec must specify a magnetic tape device and a file name.

.EXIT
.NOSUCH:
.DISABLE QUIET
.DISABLE MCR

MTA - No such file -- 'INSPEC'

.ENDIT:
.ENABLE MCR
.ENABLE QUIET
PIP 'TMP';_/DE/NM
.EXIT
```

APPENDIX H

STATISTICS BLOCK

The format of the statistics block is shown in Figure H-1 below. The statistics block is allocated manually in your program as described in 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

ZK-304-81

Figure H-1 Statistics Block Format

APPENDIX I

ERROR CODES

This appendix includes the code for:

- I/O error codes
- Directive Status Word error codes
- I/O function codes

```
        .TITLE       QIOMAC - QIOSYM MACRO DEFINITION
;
; DATE OF LAST MODIFICATION:
;
;       John R. Covert       16-Nov-1984
;
;
; ***** ALWAYS UPDATE THE FOLLOWING TWO LINES TOGETHER
        .IDENT       /0375/
        QI.VER=0375

;
; COPYRIGHT (C) 1983, 1984
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
;
; THIS SOFTWARE IS FURNISHED UNDER A LICENSE FOR USE ONLY ON A
; SINGLE COMPUTER SYSTEM AND MAY  BE  COPIED   ONLY  WITH  THE
; INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE,  OR
; ANY OTHER COPIES THEREOF, MAY NOT BE PROVIDED  OR  OTHERWISE
; MADE AVAILABLE TO ANY OTHER PERSON   EXCEPT FOR  USE ON SUCH
; SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE  TERMS.  TITLE
; TO AND OWNERSHIP OF THE SOFTWARE SHALL AT ALL  TIMES  REMAIN
; IN 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:
;
;       QIOSY$              ;DEFINE SYMBOLS
;
```

```
; TO OBTAIN GLOBAL DEFINITION OF THESE SYMBOLS USE:
;
;       QIOSY$ DEF$G        ;SYMBOLS DEFINED GLOBALLY
;
; THE MACRO CAN BE CALLED ONCE ONLY AND THEN
; REDEFINES ITSELF AS NULL.
;-

       .MACRO QIOSY$ $$$GBL,$$$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$   $$$GBL             ;DEFINE GENERAL I/O FUNCTION CODES
       .MCALL   SPCIO$
       SPCIO$   $$$GBL             ;DEVICE-DEPENDENT I/O FUNCTION CODES
       .MACRO   QIOSY$       ARG,ARG1,ARG2       ;RECLAIM MACRO STORAGE
       .ENDM    QIOSY$
       .ENDC
       .ENDM    QIOSY$

;
; 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.
;

       .ENABL   LC
       .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 EXECUTIVE AND DRIVERS
;

       .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.,<Shareable 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, CRC, or framing error>
        .IOER.   IE.NFW,-69.,<Path lost to partner> ;THIS CODE MUST BE ODD
        .IOER.   IE.DIS,-69.,<Path lost to partner> ;DISCONNECTED (SAME AS NFW)
        .IOER.   IE.PNT,-71.,<Partition/Region not in system>
        .IOER.   IE.NDR,-72.,<No dynamic space available> ; SEE ALSO IE.UPN
        .IOER.   IE.TMO,-95.,<Timeout on request>      ; see also IS.TMO
        .IOER.   IE.CNR,-96.,<Connection rejected>
        .IOER.   IE.MII,-99.,<Media inserted incorrectly>
        .IOER.   IE.SPI,-100.,<Spindown ignored>


;
; 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 read/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>
        .IOER.   IE.ALC,-84.,<Allocation failure>
        .IOER.   IE.ULK,-85.,<Unlock error>
        .IOER.   IE.WCK,-86.,<Write check failure>
        .IOER.   IE.DSQ,-90.,<Disk quota exceeded>

;
; 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.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, PSI, AND DECDATAWAY CODES
;

        .IOER.    IE.NNN,-68.,<No such node>
        .IOER.    IE.BLB,-70.,<Bad logical buffer>
        .IOER.    IE.URJ,-73.,<Connection rejected by user>
        .IOER.    IE.NRJ,-74.,<Connection rejected by network>
        .IOER.    IE.NDA,-78.,<No data available>
        .IOER.    IE.IQU,-91.,<Inconsistent qualifier usage>
        .IOER.    IE.RES,-92.,<Circuit reset during operation>
        .IOER.    IE.TML,-93.,<Too many links to task>
        .IOER.    IE.NNT,-94.,<Not a network task>
        .IOER.    IE.UKN,-97.,<Unknown name>

;
; ICS/ICR ERROR CODES
;
        .IOER.    IE.NLK,-79.,<Task not linked to specified ICS/ICR interrupts>
        .IOER.    IE.NST,-80.,<Specified task not installed>
        .IOER.    IE.FLN,-81.,<Device offline when offline request was issued>


;
; TTY ERROR CODES
;

        .IOER.    IE.IES,-82.,<Invalid escape sequence>
        .IOER.    IE.PES,-83.,<Partial escape sequence>


;
; RECONFIGURATION CODES
;

        .IOER.    IE.ICE,-47.,<Internal consistancy error>
        .IOER.    IE.ONL,-67.,<Device online>
        .IOER.    IE.SZE,-98.,<Unable to size device>

;
; PCL ERROR CODES
;

        .IOER.    IE.NTR,-87.,<Task not triggered>
        .IOER.    IE.REJ,-88.,<Transfer rejected by receiving CPU>
        .IOER.    IE.FLG,-89.,<Event flag already specified>
```

```
;
; SUCCESSFUL RETURN CODES---
;

      DEFIN$    IS.PND,+00.        ;OPERATION PENDING
      DEFIN$    IS.SUC,+01.        ;OPERATION COMPLETE, SUCCESS
      DEFIN$    IS.RDD,+02.        ;FLOPPY DISK SUCCESSFUL COMPLETION
                                   ;OF A READ PHYSICAL, AND DELETED
                                   ;DATA MARK WAS SEEN IN SECTOR HEADER
      DEFIN$    IS.TNC,+02.        ;(PCL) SUCCESSFUL TRANSFER BUT MESSAGE
                                   ;TRUNCATED (RECEIVE BUFFER TOO SMALL).
      DEFIN$    IS.CHW,+04.        ;(IBM COMM) DATA READ WAS RESULT OF
                                   ;IBM HOST CHAINED WRITE OPERATION
      DEFIN$    IS.BV,+05.         ;(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.
      DEFIN$    IS.DAO,+02.        ;SUCCESSFUL BUT WITH DATA OVERRUN
                                   ;(NOT TO BE CONFUSED WITH IE.DAO)


;
; TTY SUCCESS CODES
;

      DEFIN$    IS.CR,<15*400+1>    ;CARRIAGE RETURN WAS TERMINATOR
      DEFIN$    IS.ESC,<33*400+1>   ;ESCAPE (ALTMODE) WAS TERMINATOR
      DEFIN$    IS.CC,<3*400+1>     ;CONTROL-C WAS TERMINATOR
      DEFIN$    IS.ESQ,<233*400+1>  ;ESCAPE SEQUENCE WAS TERMINATOR
      DEFIN$    IS.PES,<200*400+1>  ;PARTIAL ESCAPE SEQUENCE WAS TERMINATOR
      DEFIN$    IS.EOT,<4*400+1>    ;EOT WAS TERMINATOR (BLOCK MODE INPUT)
      DEFIN$    IS.TAB,<11*400+1>   ;TAB WAS TERMINATOR (FORMS MODE INPUT)
      DEFIN$    IS.TMO,+2.          ;REQUEST TIMED OUT
      DEFIN$    IS.OOB,+3.          ;OUT OF BAND TERMINATOR (TERM IN HIGH BYTE)
      DEFIN$    IS.TMM,+4.          ;READ COMPLETED, MANAGEMENT MODE SEQ RCVD

;
; Professional Bisync Success Codes
;
      DEFIN$    IS.RVI,+2. ; DATA SUCC. XMITTED; HOST ACKED W/RVI
      DEFIN$    IS.CNV,+3. ; DATA SUCC. XMITTED; HOST ACKED W/CONVERSATION
      DEFIN$    IS.XPT,+5. ; DATA SUCC. RECVD IN TRANSPARENT MODE
;
; Professional Bisync Abort Codes
;
; These codes are returned in the high byte of the first word of the IOSB
; when the low byte contains IE.ABO.
;
      DEFIN$    SB.KIL,-1.  ; ABORTED BY IO.KIL
      DEFIN$    SB.ACK,-2.  ; ABORTED BECAUSE TOO MANY ACKS RECD OUT OF SEQ
      DEFIN$    SB.NAK,-3.  ; ABORTED BECAUSE NAK THRESHOLD EXCEEDED
      DEFIN$    SB.ENQ,-4.  ; ABORTED BECAUSE ENQ THRESHOLD EXCEEDED
      DEFIN$    SB.BOF,-5.  ; ABORTED BECAUSE OF IO.RLB BUFFER OVERFLOW
      DEFIN$    SB.TMO,-6.  ; ABORTED BECAUSE OF TIMEOUT
      DEFIN$    SB.DIS,-7.  ; ABORTED BECAUSE HOST DISCONNECTED W/ DLE, EOT

; ******
;
; THE NEXT AVAILABLE ERROR NUMBER IS: -101.
;
; *****
      .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> ; SEE ALSO IE.NDR
        .QIOE.    IE.INS,-02.,<Specified task not installed>
        .QIOE.    IE.PTS,-03.,<Partition too small for task>
        .QIOE.    IE.UNS,-04.,<Insufficient dynamic storage for send>
        .QIOE.    IE.ULN,-05.,<Un-assigned LUN>
        .QIOE.    IE.HWR,-06.,<Device handler not resident>
        .QIOE.    IE.ACT,-07.,<Task not active>
        .QIOE.    IE.ITS,-08.,<Directive inconsistent with task state>
        .QIOE.    IE.FIX,-09.,<Task already fixed/unfixed>
        .QIOE.    IE.CKP,-10.,<Issuing task not checkpointable>
        .QIOE.    IE.TCH,-11.,<Task is checkpointable>
        .QIOE.    IE.RBS,-15.,<Receive buffer is too small>
        .QIOE.    IE.PRI,-16.,<Privilege violation>
        .QIOE.    IE.RSU,-17.,<Resource in use>
        .QIOE.    IE.NSW,-18.,<No swap space available>
        .QIOE.    IE.ILV,-19.,<Illegal vector specified>
        .QIOE.    IE.ITN,-20.,<Invalid table number>
        .QIOE.    IE.LNF,-21.,<Logical name not found>
;
;
;
        .QIOE.    IE.AST,-80.,<Directive issued/not issued from AST>
        .QIOE.    IE.MAP,-81.,<Illegal mapping specified>
        .QIOE.    IE.IOP,-83.,<Window has I/O in progress>
        .QIOE.    IE.ALG,-84.,<Alignment error>
        .QIOE.    IE.WOV,-85.,<Address window allocation overflow>
        .QIOE.    IE.NVR,-86.,<Invalid region ID>
        .QIOE.    IE.NVW,-87.,<Invalid address window ID>
        .QIOE.    IE.ITP,-88.,<Invalid TI parameter>
        .QIOE.    IE.IBS,-89.,<Invalid send buffer size ( .GT. 255.)>
        .QIOE.    IE.LNL,-90.,<LUN locked in use>
        .QIOE.    IE.IUI,-91.,<Invalid UIC>
        .QIOE.    IE.IDU,-92.,<Invalid device or unit>
        .QIOE.    IE.ITI,-93.,<Invalid time parameters>
        .QIOE.    IE.PNS,-94.,<Partition/region not in system>
        .QIOE.    IE.IPR,-95.,<Invalid priority ( .GT. 250.)>
        .QIOE.    IE.ILU,-96.,<Invalid LUN>
        .QIOE.    IE.IEF,-97.,<Invalid event flag ( .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
                                    ;
        DEFIN$    IS.SUP,3          ;LOGICAL NAME SUPERSEDED
                                    ;
        DEFIN$    IS.WAT,4          ;OPERATION INITIATED, WAIT FOR COMPLETION
                                    ;FROM "VAX-11 RSX" RMS-21 ELEP$ DIRECTIVE
;
;
        .IF       EQ,$$MSG
        .MACRO    DRERR$  A
        .ENDM     DRERR$
        .ENDC
        .ENDM     DRERR$

;
; DEFINE THE GENERAL I/O FUNCTION CODES - DEVICE INDEPENDENT
;
        .MACRO    FILIO$  $$$GBL
        .MCALL    .WORD.,DEFIN$
        .IF       IDN,<$$$GBL>,<DEF$G>
        .IFF
        ...GBL=0
        .ENDC
;
; GENERAL I/O QUALIFIER BYTE DEFINITIONS
;
        .WORD.    IQ.X,001,000      ;NO ERROR RECOVERY
        .WORD.    IQ.Q,002,000      ;QUEUE REQUEST IN EXPRESS QUEUE
        .WORD.    IQ.S,004,000      ;SYNONYM FOR IQ.UMD
        .WORD.    IQ.UMD,004,000    ;USER MODE DIAGNOSTIC STATUS REQUIRED
        .WORD.    IQ.LCK,200,000    ;MODIFY IMPLIED LOCK FUNCTION
;
; 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
        .WORD.    IO.SET,030,000    ;SET CHARACTERISTICS FUNCTION
;
; GENERAL DEVICE DRIVER 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.LDO,110,002    ;LOAD D-SPACE OVERLAY (DISK)
        .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.ULK,000,012    ;UNLOCK BLOCK
        .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
;
;
        .MACRO    FILIO$  A
        .ENDM     FILIO$
        .ENDM     FILIO$

;
; DEFINE THE I/O FUNCTION CODES THAT ARE SPECIFIC TO INDIVIDUAL DEVICES
;
        .MACRO    SPCIO$  $$$GBL
        .MCALL    .WORD.,DEFIN$
        .IF       IDN,<$$$GBL>,<DEF$G>
        ...GBL=1
        .IFF
        ...GBL=0
        .ENDC
;
; I/O FUNCTION CODES FOR SPECIFIC DEVICE-DEPENDENT FUNCTIONS
;
        .WORD.    IO.WLV,100,001   ;(DECTAPE) WRITE LOGICAL REVERSE
        .WORD.    IO.WLS,010,001   ;(COMM.) WRITE PRECEDED BY SYNC TRAIN
        .WORD.    IO.WNS,020,001   ;(COMM.) WRITE, NO SYNC TRAIN
        .WORD.    IO.WAL,010,001   ;(TTY) WRITE PASSING ALL CHARACTERS
        .WORD.    IO.WMS,020,001   ;(TTY) WRITE SUPPRESSIBLE MESSAGE
        .WORD.    IO.CCO,040,001   ;(TTY) WRITE WITH CANCEL CONTROL-O
        .WORD.    IO.WBT,100,001   ;(TTY) WRITE WITH BREAKTHROUGH
        .WORD.    IO.WLT,010,001   ;(DISK) WRITE LAST TRACK
        .WORD.    IO.WLC,020,001   ;(DISK) WRITE LOGICAL W/ WRITECHECK
        .WORD.    IO.WPB,040,001   ;(DISK) WRITE PHYSICAL BLOCK
        .WORD.    IO.WDD,140,001   ;(FLOPPY DISK) WRITE PHYSICAL W/ DELETED DATA
        .WORD.    IO.RSN,140,002   ;(MSCP DISK) READ VOLUME SERIAL NUMBER
        .WORD.    IO.RLV,100,002   ;(MAGTAPE,DECTAPE) READ REVERSE
        .WORD.    IO.RST,001,002   ;(TTY) READ WITH SPECIAL TERMINATOR
        .WORD.    IO.RAL,010,002   ;(TTY) READ PASSING ALL CHARACTERS
        .WORD.    IO.RNE,020,002   ;(TTY) READ WITHOUT ECHO
        .WORD.    IO.RNC,040,002   ;(TTY) READ - NO LOWERCASE CONVERT
        .WORD.    IO.RTM,200,002   ;(TTY) READ WITH TIME-OUT
        .WORD.    IO.RDB,200,002   ;(CARD READER) READ BINARY MODE
        .WORD.    IO.SCF,200,002   ;(DISK) SHADOW COPY FUNCTION
        .WORD.    IO.RHD,010,002   ;(COMM.) READ, STRIP SYNC
        .WORD.    IO.RNS,020,002   ;(COMM.) READ, DON'T STRIP SYNC
        .WORD.    IO.CRC,040,002   ;(COMM.) READ, DON'T CLEAR CRC
        .WORD.    IO.RPB,040,002   ;(DISK) READ PHYSICAL BLOCK
        .WORD.    IO.RDF,240,002   ;(DISK) READ DISK FORMAT
        .WORD.    IO.RLC,020,002   ;(DISK,MAGTAPE) READ LOGICAL W/ READCHECK
        .WORD.    IO.ATA,010,003   ;(TTY) ATTACH WITH ASTS
        .WORD.    IO.GTS,000,005   ;(TTY) GET TERMINAL SUPPORT CHARACTERISTICS
        .WORD.    IO.R1C,000,005   ;(AFC,AD01,UDC) READ SINGLE CHANNEL
        .WORD.    IO.INL,000,005   ;(COMM.) INITIALIZATION FUNCTION
        .WORD.    IO.TRM,010,005   ;(COMM.) TERMINATION FUNCTION
        .WORD.    IO.RWD,000,005   ;(MAGTAPE,DECTAPE) REWIND
        .WORD.    IO.SPB,020,005   ;(MAGTAPE) SPACE "N" BLOCKS
        .WORD.    IO.RPL,020,005   ;(DISK) REPLACE LOGICAL BLOCK (RESECTOR)
        .WORD.    IO.SPF,040,005   ;(MAGTAPE) SPACE "N" EOF MARKS
        .WORD.    IO.STC,100,005   ;SET CHARACTERISTIC
```

```
.WORD.    IO.SMD,110,005   ;(FLOPPY DISK) SET MEDIA DENSITY
.WORD.    IO.SEC,120,005   ;SENSE CHARACTERISTIC
.WORD.    IO.RWU,140,005   ;(MAGTAPE,DECTAPE) REWIND AND UNLOAD
.WORD.    IO.SMO,160,005   ;(MAGTAPE) MOUNT & SET CHARACTERISTICS
.WORD.    IO.HNG,000,006   ;(TTY) HANGUP DIAL-UP LINE
.WORD.    IO.HLD,100,006   ;(TMS) HANGUP BUT LEAVE LINE ON HOLD
.WORD.    IO.BRK,200,006   ;(PRO/TTY) SEND SHORT OR LONG BREAK
.WORD.    IO.RBC,000,006   ;READ MULTICHANNELS (BUFFER DEFINES CHANNELS)
.WORD.    IO.MOD,000,006   ;(COMM.) SETMODE FUNCTION FAMILY
.WORD.    IO.HDX,010,006   ;(COMM.) SET UNIT HALF DUPLEX
.WORD.    IO.FDX,020,006   ;(COMM.) SET UNIT FULL DUPLEX
.WORD.    IO.SYN,040,006   ;(COMM.) SPECIFY SYNC CHARACTER
.WORD.    IO.EOF,000,006   ;(MAGTAPE) WRITE EOF
.WORD.    IO.ERS,020,006   ;(MAGTAPE) ERASE TAPE
.WORD.    IO.DSE,040,006   ;(MAGTAPE) DATA SECURITY ERASE
.WORD.    IO.RTC,000,007   ;READ CHANNEL - TIME BASED
.WORD.    IO.SAO,000,010   ;(UDC) SINGLE CHANNEL ANALOG OUTPUT
.WORD.    IO.SSO,000,011   ;(UDC) SINGLE SHOT, SINGLE POINT
.WORD.    IO.RPR,000,011   ;(TTY) READ WITH PROMPT
.WORD.    IO.MSO,000,012   ;(UDC) SINGLE SHOT, MULTI-POINT
.WORD.    IO.RTT,001,012   ;(TTY) READ WITH TERMINATOR TABLE
.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.LOD,000,030   ;(LPA11) LOAD MICROCODE
.WORD.    IO.MDI,000,031   ;(LPS11) SYNCHRONOUS DIGITAL INPUT
.WORD.    IO.DCI,000,031   ;(UDC) CONTACT INT - DISCONNECT
.WORD.    IO.PAD,000,031   ;(PSI) DIRECT CONTROL OF X.29 PAD
.WORD.    HT.RPP,010,000   ;(PSI) RESET PAD PARAMETERS SUBFUNCTION
.WORD.    IO.XMT,000,031   ;(COMM.) TRANSMIT SPECIFIED BLOCK WITH ACK
.WORD.    IO.XNA,010,031   ;(COMM.) TRANSMIT WITHOUT ACK
.WORD.    IO.INI,000,031   ;(LPA11) INITIALIZE
.WORD.    IO.HIS,000,032   ;(LPS11) SYNCHRONOUS HISTOGRAM SAMPLING
.WORD.    IO.RCI,000,032   ;(UDC) CONTACT INT - READ
.WORD.    IO.RCV,000,032   ;(COMM.) RECEIVE DATA IN BUFFER SPECIFIED
.WORD.    IO.CLK,000,032   ;(LPA11) START CLOCK
.WORD.    IO.CSR,000,032   ;(BUS SWITCH) READ CSR REGISTER
.WORD.    IO.MDO,000,033   ;(LPS11) SYNCHRONOUS DIGITAL OUTPUT
.WORD.    IO.CTI,000,033   ;(UDC) TIMER - CONNECT
.WORD.    IO.CON,000,033   ;(COMM.) CONNECT FUNCTION
                           ;(VT11) - CONNECT TASK TO DISPLAY PROCESSOR
                           ;(BUS SWITCH) CONNECT TO SPECIFIED BUS
                           ;(COMM./PRO) DIAL TELEPHONE AND ORIGINATE
.WORD.    IO.ORG,010,033   ;(COMM.) INITIATE CONNECTION IN ORIGINATE MODE
.WORD.    IO.ANS,020,033   ;(COMM.) INITIATE CONNECTION IN ANSWER MODE
.WORD.    IO.STA,000,033   ;(LPA11) START DATA TRANSFER
                           ;(XJDRV) - SHOW STATE
.WORD.    IO.DTI,000,034   ;(UDC) TIMER - DISCONNECT
.WORD.    IO.DIS,000,034   ;(COMM.) DISCONNECT FUNCTION
                           ;(VT11) - DISCONNECT TASK FROM DISPLAY PROCESSOR
                           ;(BUS SWITCH) SWITCHED BUS DISCONNECT
.WORD.    IO.MDA,000,034   ;(LPS11) SYNCHRONOUS D/A OUTPUT
.WORD.    IO.DPT,010,034   ;(BUS SWITCH) DISCONNECT TO SPECIF PORT NO.
.WORD.    IO.RTI,000,035   ;(UDC) TIMER - READ
.WORD.    IO.CTL,000,035   ;(COMM.) NETWORK CONTROL FUNCTION
```

```
        .WORD.    IO.STP,000,035  ;(LPS11,LPA11) STOP IN PROGRESS FUNCTION
                                  ;(VT11) - STOP DISPLAY PROCESSOR
        .WORD.    IO.SWI,000,035  ;(BUS SWITCH) SWITCH BUSSES
        .WORD.    IO.CNT,000,036  ;(VT11) - CONTINUE DISPLAY PROCESSOR
                                  ;(XJDRV) - SHOW COUNTERS
        .WORD.    IO.ITI,000,036  ;(UDC) TIMER - INITIALIZE
;
; EXTENDED I/O FUNCTION
;
        .WORD.    IO.EIO,000,037  ;(TTY) TSA EXTENDED I/O
;
; PRO 300 SERIES BITMAP FUNCTIONS
;
;   NOTE: THESE FUNCTIONS ARE FOR DEC USE ONLY AND ARE SUBJECT TO CHANGE
;

        .WORD.    IO.RSD,030,014  ; READ SPECIAL DATA
        .WORD.    IO.WSD,010,013  ; WRITE SPECIAL DATA
        DEFIN$    SD.TXT,0        ; TEXT DATA TYPE FOR SPECIAL DATA
        DEFIN$    SD.GDS,1        ; GIDIS DATA TYPE FOR SPECIAL DATA

;
; PROFESSIONAL 300 BISYNC DRIVER (XJDRV) FUNCTIONS
;
        .WORD.    SB.PRT,020,003  ; ATTACH AS A PRINTER
        .WORD.    SB.CLR,010,036  ; CLEAR COUNTERS (IO.CNT SUBFUNCTION)
        .WORD.    SB.RDY,010,033  ; SET DEVICE STATE READY (IO.STA SUBFUNC)
        .WORD.    SB.NRD,020,033  ; SET DEVICE STATE NOT READY

        .WORD.    IO.LBK,000,035  ; PERFORM LOOPBACK TEST

        .WORD.    SB.CBL,010,035  ; PERFORM CABLE LOOPBACK TEST
        .WORD.    SB.CLK,020,035  ; DEVICE PERFORMS LINE CLOCKING
;
; COMMUNICATIONS FUNCTIONS
;

        .WORD.    IO.CPR,010,033  ;CONNECT NO TIME-OUTS
        .WORD.    IO.CAS,020,033  ;CONNECT WITH AST
        .WORD.    IO.CRJ,040,033  ;CONNECT REJECT
        .WORD.    IO.CBO,110,033  ;BOOT CONNECT
        .WORD.    IO.CTR,210,033  ;TRANSPARENT CONNECT
        .WORD.    IO.GNI,010,035  ;GET NODE INFORMATION
        .WORD.    IO.GLI,020,035  ;GET LINK INFORMATION
        .WORD.    IO.GLC,030,035  ;GET LINK INFO CLEAR COUNTERS
        .WORD.    IO.GRI,040,035  ;GET REMOTE NODE INFORMATION
        .WORD.    IO.GRC,050,035  ;GET REMOTE NODE ERROR COUNTS
        .WORD.    IO.GRN,060,035  ;GET REMOTE NODE NAME
        .WORD.    IO.CSM,070,035  ;CHANGE SOLO MODE
        .WORD.    IO.CIN,100,035  ;CHANGE CONNECTION INHIBIT
        .WORD.    IO.SPW,110,035  ;SPECIFY NETWORK PASSWORD
        .WORD.    IO.CPW,120,035  ;CHECK NETWORK PASSWORD
        .WORD.    IO.NLB,130,035  ;NSP LOOPBACK
        .WORD.    IO.DLB,140,035  ;DDCMP LOOPBACK

;
; ICS/ICR I/O FUNCTIONS
;
        .WORD.    IO.CTY,000,007  ;CONNECT TO TERMINAL INTERRUPTS
        .WORD.    IO.DTY,000,015  ;DISCONNECT FROM TERMINAL INTERRUPTS
        .WORD.    IO.LDI,000,016  ;LINK TO DIGITAL INTERRUPTS
        .WORD.    IO.UDI,010,023  ;UNLINK FROM DIGITAL INTERRUPTS
        .WORD.    IO.LTI,000,017  ;LINK TO COUNTER MODULE INTERRUPTS
        .WORD.    IO.UTI,020,023  ;UNLINK FROM COUNTER MODULE INTERRUPTS
```

```
       .WORD.    IO.LTY,000,020  ;LINK TO REMOTE TERMINAL INTERRUPTS
       .WORD.    IO.UTY,030,023  ;UNLINK FROM REMOTE TERMINAL INTERRUPTS
       .WORD.    IO.LKE,000,024  ;LINK TO ERROR INTERRUPTS
       .WORD.    IO.UER,040,023  ;UNLINK FROM ERROR INTERRUPTS
       .WORD.    IO.NLK,000,023  ;UNLINK FROM ALL INTERRUPTS
       .WORD.    IO.ONL,000,037  ;UNIT ONLINE
       .WORD.    IO.FLN,000,025  ;UNIT OFFLINE
       .WORD.    IO.RAD,000,021  ;READ ACTIVATING DATA

;
; IP11 I/O FUNCTIONS
;
       .WORD.    IO.MAO,010,007  ;MULTIPLE ANALOG OUTPUTS
       .WORD.    IO.LEI,010,017  ;LINK EVENT FLAGS TO INTERRUPT
       .WORD.    IO.RDD,010,020  ;READ DIGITAL DATA
       .WORD.    IO.RMT,020,020  ;READ MAPPING TABLE
       .WORD.    IO.LSI,000,022  ;LINK TO DSI INTERRUPTS
       .WORD.    IO.UEI,050,023  ;UNLINK EVENT FLAGS
       .WORD.    IO.USI,060,023  ;UNLINK FROM DSI INTERRUPTS
       .WORD.    IO.CSI,000,026  ;CONNECT TO DSI INTERRUPTS
       .WORD.    IO.DSI,000,027  ;DISCONNECT FROM DSI INTERRUPTS
       .WORD.    IO.RAM,000,032  ;READ ANALOG MAPPING TABLES
       .WORD.    IO.RLK,000,013  ;READ RESOURCE LINKAGES
       .WORD.    IO.EBT,000,011  ;CHECK EBIT STATUS

;
; PCL11 I/O FUNCTIONS
;

       .WORD.    IO.ATX,000,001  ;ATTEMPT TRANSMISSION
       .WORD.    IO.ATF,000,002  ;ACCEPT TRANSFER
       .WORD.    IO.CRX,000,031  ;CONNECT FOR RECEPTION
       .WORD.    IO.DRX,000,032  ;DISCONNECT FROM RECEPTION
       .WORD.    IO.RTF,000,033  ;REJECT TRANSFER


       .MACRO    SPCIO$  A
       .ENDM     SPCIO$
       .ENDM     SPCIO$

;
; DEFINE THE I/O CODES FOR USER-MODE DIAGNOSTICS.  ALL DIAGNOSTIC
; FUNCTIONS ARE IMPLEMENTED AS A SUBFUNCTION OF I/O CODE 10 (OCTAL).
;

       .MACRO    UMDIO$    $$$GBL
       .MCALL    .WORD.,DEFIN$
       .IF IDN   <$$$GBL>,<DEF$G>
...GBL=1
       .IFF
...GBL=0
       .ENDC

;
; DEFINE THE GENERAL USER-MODE I/O QUALIFIER BIT.
;

       .WORD.    IQ.UMD,004,000  ;USER-MODE DIAGNOSTIC REQUEST

;
; DEFINE USER-MODE DIAGNOSTIC FUNCTIONS.
;

       .WORD.    IO.HMS,000,010  ;(DISK) HOME SEEK OR RECALIBRATE
       .WORD.    IO.BLS,010,010  ;(DISK) BLOCK SEEK
       .WORD.    IO.OFF,020,010  ;(DISK) OFFSET POSITION
```

```
        .WORD.    IO.RDH,030,010   ;(DISK) READ DISK HEADER
        .WORD.    IO.WDH,040,010   ;(DISK) WRITE DISK HEADER
        .WORD.    IO.WCK,050,010   ;(DISK) WRITECHECK (NONTRANSFER)
        .WORD.    IO.RNF,060,010   ;(DECTAPE) READ BLOCK NUMBER FORWARD
        .WORD.    IO.RNR,070,010   ;(DECTAPE) READ BLOCK NUMBER REVERSE
        .WORD.    IO.LPC,100,010   ;(MAGTAPE) READ LONGITUDINAL PARITY CHAR
        .WORD.    IO.RTD,120,010   ;(DISK) READ TRACK DESCRIPTOR
        .WORD.    IO.WTD,130,010   ;(DISK) WRITE TRACK DESCRIPTOR
        .WORD.    IO.TDD,140,010   ;(DISK) WRITE TRACK DESCRIPTOR DISPLACED
        .WORD.    IO.DGN,150,010   ;DIAGNOSE MICRO PROCESSOR FIRMWARE
        .WORD.    IO.WPD,160,010   ;(DISK) WRITE PHYSICAL BLOCK
        .WORD.    IO.RPD,170,010   ;(DISK) READ PHYSICAL BLOCK
        .WORD.    IO.CER,200,010   ;(DISK) READ CE BLOCK
        .WORD.    IO.CEW,210,010   ;(DISK) WRITE CE BLOCK

;
; MACRO REDEFINITION TO NULL
;

        .MACRO    UMDIO$  A
        .ENDM


        .ENDM     UMDIO$

;
; 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
        DEFIN$    SYM,LO
        .IF       GT,$$MSG
        .MCALL    .IOMG.
        .IOMG.    SYM,LO,<MSG>
        .ENDC
        .ENDM     .IOER.
;
; I/O ERROR CODES ARE DEFINED THROUGH THIS MACRO, WHICH THEN INVOKES THE
; ERROR MESSAGE-GENERATING MACRO; ERROR CODES -129 THROUGH -256
; ARE USED IN THE QIOSYM.MSG FILE
;
        .MACRO    .QIOE.       SYM,LO,MSG
        DEFIN$    SYM,LO
        .IF       GT,$$MSG
        .MCALL    .IOMG.
        .IOMG.    SYM,<LO-128.>,<MSG>
        .ENDC
        .ENDM     .QIOE.
;
; CONDITIONALLY GENERATE DATA FOR WRITING A MESSAGE FILE
;
        .MACRO    .IOMG.  SYM,LO,MSG
        .WORD     -^O<LO>
        .ENABL    LC
        .ASCIZ    ^MSG^
        .DSABL    LC
        .EVEN
        .IIF      LT,^O<$$$MAX+<LO>>,$$$MAX=-^O<LO>
        .ENDM     .IOMG.
```

```
;
; DEFINE THE SYMBOL SYM WHERE LO IS THE LOW-ORDER BYTE, HI IS THE HIGH BYTE
;
        .MACRO    .WORD.  SYM,LO,HI
        DEFIN$    SYM,<HI*400+LO>
        .ENDM     .WORD.

        .DSABL    LC
```

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 file name in bytes (stored in Radix-50)
S.FNB	- Size of filename block in bytes
S.FNBW	- Size of filename block in words
S.FNTY	- Size of file name and file type in words (stored in Radix-50)
S.FSR2	- Size of FSR2 (basic impure area)
S.FTYP	- Size of file type in bytes (in Radix-50)
S.NFEN	- Size of a complete file name in bytes -- file ID, name, type, and version

APPENDIX K

RSX-11M/M-PLUS FCS LIBRARY SYSGEN OPTIONS

K.1 FCS LIBRARY OPTIONS

The system manager has the option of selecting one of several FCS libraries as the default FCS library. You can replace the default library in SYSLIB with one of the other libraries shown in Table K-1 by using the /RP switch to the LBR utility. Refer to the *RSX-11M/M-PLUS Utilities Manual* for more information. Table K-1 contains the FCS libraries that are available with each RSX-11M, RSX-11M-PLUS, and Micro/RSX system, and a brief description of each.

Table K-1
FCS Library Descriptions

FCS Library Support	Description
[1,1]FCS.OBJ	Standard FCS routines. Distributed and included in SYSLIB.OLB as the default FCS library routines for RSX-11M.
[1,1]FCSMTA.OBJ	Includes standard FCS routines, plus ANSI magnetic tape support and "big buffering" (see Section 2.2.1.6 for block buffer size override specification). Distributed and included as the default FCS library routines for RSX-11M-PLUS and Micro/RSX.
[1,1]FCSMBF.OBJ	Provides multiple buffering support, big buffering support, and ANSI magnetic tape support in addition to the standard FCS routines.

K.2 .FCTYP

The FCS routine .FCTYP returns a description of the FCS conditional assembly parameters that were set when FCS was built.

The format of the call is:

```
    CALL    .FCTYP
```

There are no input parameters.

The information is returned in R1. The bits set in the mask word returned in R1 correspond to the conditional assembly parameters as shown in Table K-2.

Table K-2
.FCTYP Values

Conditional Assembly Symbol	R1 Bit Mask Symbol	Meaning
R$$ANI	FT.ANI	ANSI magnetic tape support
R$$BBF	FT.BBF	Big buffer support
R$$MBF	FT.MBF	Multibuffer support

RSX-11M/M-PLUS and Micro/RSX
I/O Operations Reference Manual
AA-FD14A-TC

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____________

Name ____________ Date ____________

Organization ____________

Street ____________

City ____________ State ____________ Zip Code or Country ____________