IAS/RSX-11
I/O Operations
Reference Manual

Order No. AA-2515D-TC

RSX-11M Version 3.2
RSX-11M-PLUS Version 1.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, December 1975
Revised: December 1976
December 1977
June 1979

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

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

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

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

CONTENTS

TABLES (Cont.)

SUMMARY OF TECHNICAL CHANGES

This revision of the I/O Operations Reference Manual contains changes and additions relative to the items listed as follows:

1. File control services described in this manual are included in RSX-11M-PLUS systems.

2. "Big buffer" support is provided, allowing block-buffer sizes in excess of 1 disk block to be specified in the File Descriptor Block for record I/O operations. (See Section 2.2.1.6.) By using this technique, generally fewer disk operations are required, and file access time is reduced accordingly.

3. The Table-Driven Parser (TPARS) has a new entry point added allowing the use of a user-supplied debug routine. The use of a debug routine will allow monitoring TPARS operation by automatically entering the debug routine at each state transition.

4. Appendix K is added, listing RSX-11M/M-PLUS FCS library SYSGEN options. A brief description of each option is provided.

PREFACE

MANUAL OBJECTIVES

The purpose of this manual is to familiarize the users of an RSX-11M, RSX-11M-PLUS, or IAS operating system with the file control services (FCS) facility provided with the system.

INTENDED AUDIENCE

Since the file control services described herein pertain to both MACRO-11 and FORTRAN programs, the reader is assumed to be familiar with the manuals describing these program-development tools. Also, since the development of programs in an RSX-11 or IAS environment necessarily involves the use of the Task Builder, the reader is likewise assumed to be familiar with this system program.

STRUCTURE OF THE DOCUMENT

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

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

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

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

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

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

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

Chapter 8 describes the queuing of files for printing. This facility is available at both the MACRO level and subroutine level.

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

Appendix E illustrates the structure of the index file of a Files-11 volume, while Appendix F describes in detail the format and content of a file-header block. The format and content of magnetic tape labels are similarly described in Appendix G. The format and content of the statistics block are described in Appendix H.

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

Appendix K lists RSX-11M/M-PLUS FCS library SYSGEN options, including a brief description of each.

ASSOCIATED DOCUMENTS

Other manuals closely allied to the purposes of this document are described briefly in the appropriate IAS and RSX-11 Documentation Directories. The Documentation Directories define the intended readership of each manual in the appropriate set and provide a brief synopsis of each manual's contents.

CONVENTIONS USED IN THIS DOCUMENT

Unless otherwise noted, the term RSX-11 refers to both RSX-11M and RSX-11M-PLUS.

CHAPTER 1

FILE CONTROL SERVICES

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

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

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

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

FCS thus is virtually transparent to the user so that records can be read or written in logical units that are consistent with particular application requirements.

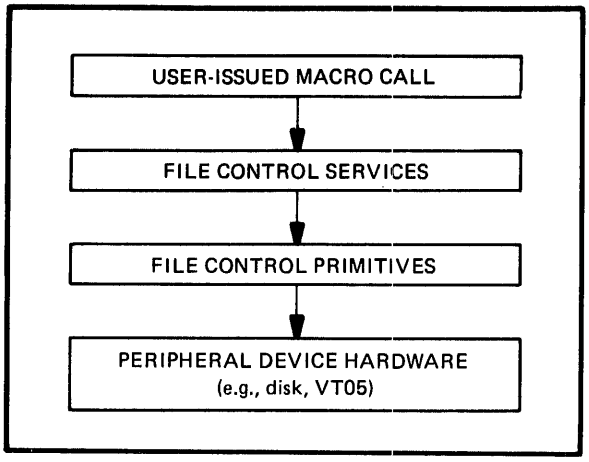


Figure 1-1 File-Access Operation

FCS provides an extensive set of macros to simplify the user's interface to the system's I/O facilities. In addition to generating calls to FCS subroutines, these macros create and maintain certain data structures that are required when performing any file I/O operations. The required data structures include the following:

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

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

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

4. A file storage region (FSR) that is used by FCS for working storage.

The FDB is described in detail in Appendix A and Appendix B. The dataset descriptor and the default filename block are described in detail in Section 2.4. The FSR is described in Section 1.2.

1.1 FILE ACCESS METHODS

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

1.2 FILE STORAGE REGION (FSR)

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

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

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

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

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

Record I/O operations are described in detail in Section 1.5.

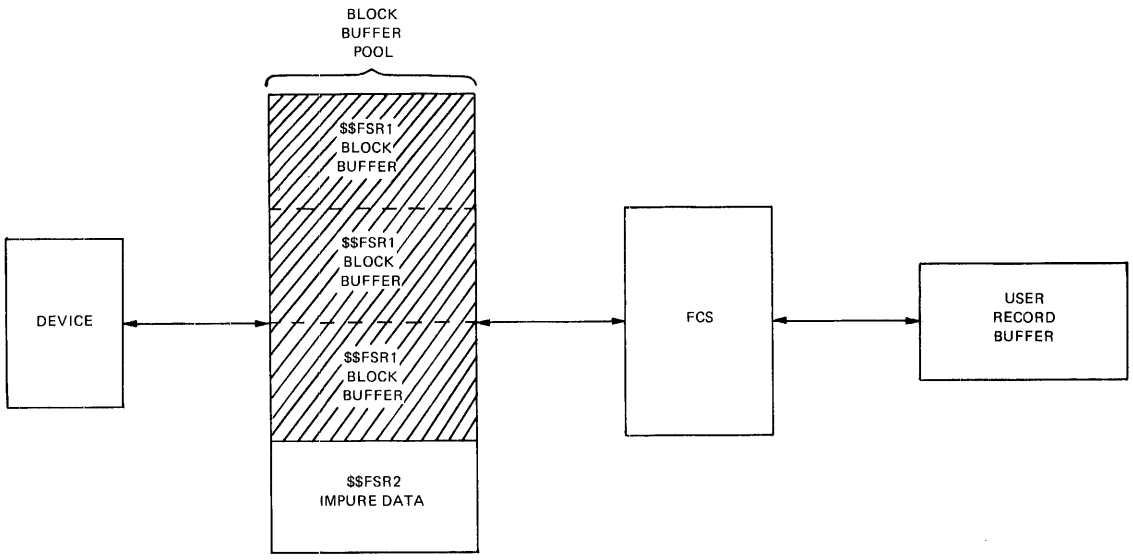


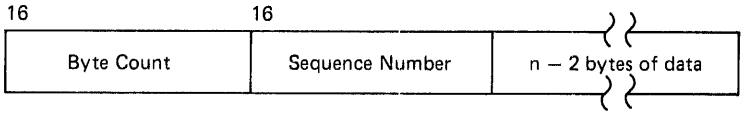
Figure 1-2 Record I/O Operations

1.3 DATA FORMATS FOR FILE-STRUCTURED DEVICES

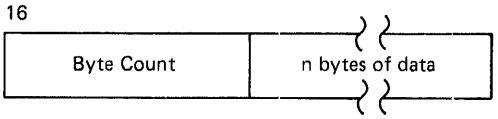
Data are transferred between peripheral devices and memory in blocks. A data file consists of virtual blocks, each of which may contain one or more logical records created by a user program. In FCS terms, a virtual block in a file consists of 512(10) bytes. The size of the logical records in the virtual blocks is under the control of the user program that originally writes the records.

When creating a new file, a user program can specify that the records in the file need not all be the same size. Such records are known as variable-length records. Conversely, if the user 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 two-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, while a logical block number is a volume relative value. Ordinarily, records may cross block boundaries. This 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.

1.3.1 Data Formats for ANSI Magtape

Both fixed- and variable-length records may be used on magtape; their format conforms to the ANSI standard.

On magtape, 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 greater than 8 bytes (14 bytes for a write function) and up to 2048 bytes with the use of the FDBF$ macro (see Section 2.2.1.6). 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 end of 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), which appears where the next byte count should be, signals the end of data in that block.

1.4 BLOCK I/O OPERATIONS

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

When block I/O is used, the number of the virtual block to be processed is specified as a parameter in the appropriate READ$/WRITE$ macro call; the virtual blocks so specified are processed directly in a reserved buffer in user memory space. READ$ and WRITE$ can be used only on block-structured devices.

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

1.5 RECORD I/O OPERATIONS

The GET$ and PUT$ macro calls (see Sections 3.9 and 3.12, respectively) are provided for processing individual user records in files. Using the FSR block buffers (see Section 1.2), the GET$ and PUT$ routines perform the necessary blocking and deblocking of records within the virtual blocks of the file, allowing the user program to access logical records.

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. The user 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. Successive GET$ or PUT$ operations in random-access mode can access records anywhere within the file. To do so, the user program need only modify the record number specified as part of the random record operation. After each such random operation, FCS increments the record number used in the operation. If the user 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, i.e., control is returned to the user program only after the requested I/O operation is completed.

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

1.6 DATA-TRANSFER MODES

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

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

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

1.6.1 Move Mode

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

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

1.6.2 Locate Mode

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

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

1.7 MULTIPLE BUFFERING FOR RECORD I/O (IAS ONLY)

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

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

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

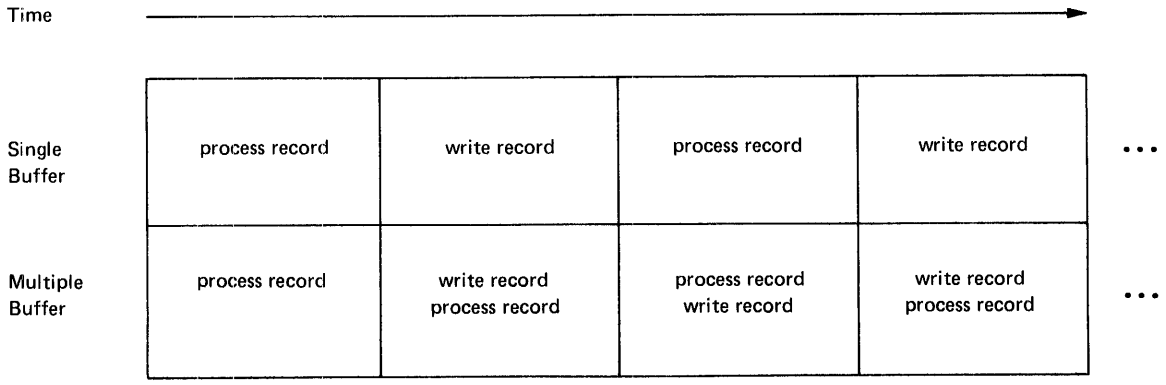


Figure 1-3 Single Buffering Versus Multiple Buffering

1.8 SHARED ACCESS TO FILES

Files-11 permits shared access to files according to established conventions. Two macro calls, among several available in FCS for opening files, may be issued to invoke these conventions. The OPNS$x macro call (see Section 3.2) is used specifically to open a file for shared access. The OPEN$x macro call (see Section 3.1), on the other hand, invokes generalized open functions that have shared-access implications only in relation to other I/O requests then 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. These macro calls and the shared-access conditions that they invoke are summarized below.

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

1. When the OPNS$R macro call is issued, read access to the file is granted unconditionally, regardless of the presence of one or more concurrent write-access requests to the file. (The OPNS$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 point 2.

2. While FCS allows concurrent write-access requests through the use of the OPNS$W, OPNS$M, OPNS$U, and OPNS$A macro, the synchronization of access to the file is the responsibility of the user tasks themselves. To avoid the retrieval or storage of inconsistent data, each such task must implement and use some user-defined mechanism that assures that the file is accessed in a serial fashion.

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 above 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 imply the presence of read requests from several separate tasks. The same task, for example, may open the same file using different logical unit numbers.

1.9 FILE DESCRIPTOR BLOCK (FDB)

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

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

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

1.10 DATASET DESCRIPTOR AND DEFAULT FILENAME BLOCK

Normally, either a dataset descriptor or a default filename block is specified for each file the user intends to open. These data structures provide FCS with the file specifications required for opening a file. Although either one or the other is usually defined, both can be specified for the same file. The dataset descriptor and the default filename block are summarized below and described in detail in Sections 2.4.1 and 2.4.2, respectively.

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

1.11 KEY TERMS USED THROUGHOUT THIS MANUAL

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

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

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

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

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

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

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

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

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

1.12 SYSTEM CHARACTERISTICS

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

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

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

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

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

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

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

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

NOTE

When I/O is being done using the READ$/WRITE macros, the IOSB parameter **must** be specified for F.ERR and the C-bit to be properly returned (see Section 3.15).

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

CHAPTER 2

PREPARING FOR I/O

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

- Define a file descriptor block (FDB) for each file that is to be opened simultaneously by the user program (see Section 2.2).
- Define a dataset descriptor and/or a default filename block (see Section 2.4.1 or 2.4.2, respectively) if the user intends to access these structures to provide required file specifications to FCS.
- 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 FORTRAN-IV User's Guide and the FORTRAN-IV-PLUS User's Guide.)

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

1. By the assembly-time FDB initialization macro calls (see Section 2.2.1).
2. By the run-time FDB initialization macro calls (see Section 2.2.2).
3. By the file-processing macro calls (see Chapter 3).

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

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

Table 2-1
Macro Calls Generating FDB Information

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

2.1 .MCALL DIRECTIVE - LISTING NAMES OF REQUIRED MACRO DEFINITIONS

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

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

The .MCALL directive takes the following general form:

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

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

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

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

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

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

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

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

NOTE

The macro FCSMC$ can be used to .MCALL the most commonly used FCS macros, as follows:

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

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

2.2 FILE DESCRIPTOR BLOCK (FDB)

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

2.2.1 Assembly-Time FDB Initialization Macros

Assembly-time initialization requires that the FDBDF$ macro call be issued (see Section 2.2.1.1) to allocate space for and to define the beginning address of the FDB. Additional macro calls can then be

issued to establish other required information in this structure. The assembly-time macros that accomplish these functions are described in the following sections. These macro calls take the general form shown below:

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

where: mcnam$A represents the symbolic name of the macro.

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

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

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

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

Also, a trailing comma need not be inserted to reflect the omission of a parameter beyond the last explicit specification. For example, the 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, i.e., if any field in the macro call contains a null specification, the corresponding cell in the FDB is not initialized and thus remains 0.

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

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

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

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

The FDB allocation macro takes the following form:

```
label:  FDBDF$
```

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

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

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

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

As noted earlier, the source program must embody one FDBDF$ macro call logically similar to those above for each file to be accessed simultaneously by the user program. FDB's 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 file to be opened simultaneously.

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

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

where: rtyp represents a symbolic value that defines the type of records to be built as the new file is created. 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.

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

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

FD.FTN - Indicates that the first byte in each record 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, e.g., a line printer or a terminal.

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

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

rsiz represents a numeric value that defines the size (in bytes) of fixed-length records to be written to the file. This value, which initializes FDB offset location F.RSIZ, need not be specified if R.VAR has been specified as the record type parameter above (for variable-length records). If R.VAR 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 represents 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 at file-create time, and further that the file is to be contiguous.

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

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

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

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

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

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

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

NOTE

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

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

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

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

2. If only 1 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 respectively. The volume default extend size cannot be established at the FCS level; this value must be established when the volume is initially mounted.

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

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

In the above example, the record attribute (ratt) parameter has been omitted, as indicated by the second comma (,) in the parameter string. Also, the cntg and aloc parameters have been omitted. Their omission, however, occurs following the last explicit specification, and their absence need not be indicated by trailing commas in the parameter string. Since the aloc parameter has been omitted, file extension (if it becomes necessary) 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 below:

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

The above 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.

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

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

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

The FDRC$A macro call takes the following format:

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

where: racc specifies which variation of block or record I/O is to be used to process the file. This parameter initializes the record-access byte (offset location F.RACC) in the FDB. The first value below applies only for block I/O (READ$/WRITE$) operations; all remaining values are specific to record I/O (GET$/PUT$) operations:

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

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

FD.RAN - Indicates that random-access mode is to be used in processing the file. If this value is not specified, sequential-access mode is used by default. Refer to Section 1.5 for a description of random-access mode.

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

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

Should the user wish to perform a PUT$ operation within the body of a file, the .POINT routine described in Section 4.10.1 may be called. This routine, which permits a limited degree of random access to a file, positions the file to a user-specified byte within a virtual block in preparation for the PUT$ operation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where:

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

bkds — represents a numeric value that specifies the size (in bytes) of the block to be read or written when a block I/O request (READ$ or WRITE$ macro call) is issued. This parameter initializes FDB offset location F.BKDS. The size specified must be an even, positive (sign bit must not be set) value; thus, the maximum number of bytes that can be specified is 32766. If an integral number of blocks are to be specified, the practical maximum number of bytes that can be specified is equal to 63 virtual blocks, or 32256(10) bytes.

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

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

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

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

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

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

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 an I/O status block address is specified. Thus, the user is advised to specify this parameter to allow the return of block I/O status information and to facilitate normal error reporting.

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

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

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

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

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

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

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

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

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

The FDOP$A macro call takes the following form:

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

where:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

actl represents a symbolic value that is used to specify the following control information in FDB location F.ACTL:

1. Magnetic tape position.

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

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

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

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

At file open and close, tapes are not rewound.

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

The volume default is used for the file window.

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

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

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

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

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

The number of retrieval pointers for a file window can be specified in the low-order byte of F.ACTL. The default number of retrieval pointers is the file-window mapping pointer count parameter (/WIN) included in the Initialize Volume or Mount Volume MCR commands; the default value for this parameter is 7. Retrieval pointers are used to point to contiguous blocks of the file on disk. Access to fragmented files may be optimized by increasing the number of retrieval pointers, i.e., by increasing the size of the window. Likewise, additional memory can be freed by reducing the number of pointers for files with little or no fragmentation, e.g., contiguous files.

As noted, if neither the dspt nor the dfnb parameter is specified, corresponding offset locations F.DSPT and F.DFNB contain 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$A macro call; they may also be initialized at run-time through the FDOP$R or the generalized OPEN$x macro call (see Section 3.1).

The following examples represent the FDOP$A macro call as it might appear in a source program:

```
FDOP$A  1,,DFNB

FDOP$A  2,OFDSPT

FDOP$A  2,OFDSPT,DFNB

FDOP$A  1,CSIBLK+C.DSDS

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

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

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

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

The fourth example shows a macro call 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 the invocation of the CSI$ macro call. All considerations relevant to the use of CSI as a dynamic command line processor are presented in Section 6.2.

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

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

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

The FDBF$A macro call takes the following format:

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

where:

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

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

ovbs — represents a numeric value that specifies an FSR block-buffer size, in bytes, which overrides the standard block size for the particular device associated with the file. This parameter initializes FDB offset location F.OVBS to the specified block-buffer size.

When ovbs is used in RSX-11 systems to specify an FSR block-buffer size for disks, the desired number of bytes is specified in integral multiples of 512(10), overriding the standard 512(10) (one sector) block-buffer size. Block-buffer sizes up to 63 sectors (32256(10) bytes) can be specified for disks. Increasing the block-buffer size in this manner greatly reduces average disk access time since several contiguous sectors are generally read or written during a typical disk access operation. An override block size of 2048(10) bytes (4 sectors) or 2560(10) bytes (5 sectors) is recommended since 2048(10) bytes also provides ANSI magtape buffer capability, and 2560(10) 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 the FDB is to be used for additional disk I/O operations, the ovbs parameter must be issued in an FDBF$R macro prior to accessing the disk.

NOTE

When block-buffer sizes greater than 1 sector (512(10)) bytes are specified, $$FSR1 size must be increased accordingly. This is done by specifying an appropriate value for the bufsiz parameter in the FSRSZ$ macro call (see Section 2.6.1).

In IAS systems, an override block size is allowed only for record-oriented devices (such as line printers) and sequential devices (such as magnetic tape units); if specified for a block-oriented device, the override block size is ignored. For spooled output to a record-oriented device, do not allocate a buffer less than a single sector (512(10) bytes).

Routines that will read ANSI-standard magnetic tape without prior knowledge of the format of the files that will be read must specify an override block size of 2048(10) 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 Executive Reference Manual for the operating system in use.) The standard block size for each device is established at SYSGEN time, or via the MCR SET/BUF command.

mbct represents a numeric value that specifies the multiple buffer count, i.e., the number of buffers to be used by FCS in processing the associated file. This parameter initializes FDB offset location F.MBCT. If this value is greater than 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 below).

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 following the mbfg parameter below.

If, in specifying the FSRSZ$ macro call (see Section 2.6.1), sufficient memory space has not been allocated to accommodate the number of buffers established by the mbct parameter, FCS allocates as many buffers as 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.

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

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

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

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

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

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

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

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

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

NOTE

When using write-behind multi-buffering, there is no gain in efficiency if the size of the file must be increased in order 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:

1. Space for the file can be completely pre-allocated, either by use of the "cntg" parameter in the FDOP$x macro, or by using the .EXTND subroutine.

2. The default extension amount can be increased from 5 blocks by use of the "aloc" parameter of the FDAT$x macro call. For example, if an "aloc" parameter of 10(10) is specified, the number of write-behind operations that will be overlapped will increase to 90%.

3. The file can be accessed using random I/O. Since issuing PUT$R macros to access random pre-existing locations in the file does not require extends, the percentage of overlapped operations is increased.

For IAS, the default buffer count can be changed, if desired, by modifying a location in $$FSR2, the second of two program sections comprising the FSR. A location defined as .MBFCT in $$FSR2 normally contains a default buffer count of 1. This default value may be changed, as follows:

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

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

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

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

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

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

```
FDBF$A  25.,,1

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

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

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

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

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

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

2.2.2 Run-Time FDB Initialization Macros

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

- FDAT$R - Initializes or alters the file-attribute section of the FDB.
- FDRC$R - Initializes or alters the record-access section of the FDB.
- FDBK$R - Initializes or alters the block-access section of the FDB (see Item 4 below).
- FDOP$R - Initializes or alters the file-open section of the FDB.
- FDBF$R - Initializes or alters the block-buffer section of the FDB.

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.

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

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

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

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

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

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

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

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

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

   c. Also, if the parameter is a register specifier (e.g., 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.

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

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

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

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

      ```
      VBNADR: .BLKW   2
      ```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
FDBOUT: FDBDF$
```

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

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

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

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

2. A general register specifier may be used as the initial parameter in a run-time macro call. When a register other than R0 is used, 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).

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

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

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

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

2.3 GLOBAL VERSUS LOCAL DEFINITIONS FOR FDB OFFSETS

Although the FDB offsets can be defined either locally or globally, the design of FCS does not require that the user 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 automatically 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/or macro-call parameters anywhere in the source program coding, as described in the following section.

2.3.1 Specifying Global Symbols in the Source Coding

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

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

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

```
FDBIN:  FDBDF$
```

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

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

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

An equivalent instruction is the following:

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

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

2.3.2 Defining FDB Offsets and Bit Values Locally

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

When issued, the FDOF$L and FCSBT$ macro calls define symbols in a manner 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 or not the FDOF$L and FCSBT$ macro calls are invoked should not in any way affect the coding style or the manner in which the FDB offsets and bit values are used.

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

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

2.4 CREATING FILE SPECIFICATIONS WITHIN THE USER PROGRAM

Certain information describing the file must be present in the FDB before the file can be opened. The file is located using a file specification 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 filename;
4. A file type (RSX-11) or file extension (IAS);
5. A file version number.

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

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

1. The dataset descriptor. This tabular structure may be created and initialized manually through the use of .WORD directives. Section 2.4.1 describes this data structure in detail.

2. The default filename block. In contrast to the manually-created dataset descriptor, the default filename block is created by issuing the NMBLK$ macro call. This macro call allocates a block of storage in the user program at assembly-time and initializes this structure with parameters supplied in the call. This structure is described in detail in Section 2.4.2.

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

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

Note, however, that the device name and the unit number need not be specified in either the dataset descriptor or the default filename block, since these values are defaulted to 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, the reader should refer to Appendix B.

2.4.1 Dataset Descriptor

The dataset descriptor is often oriented toward the use of a fixed (built-in) filename in the user program. A given application program, for example, may require access only to a limited and 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 may be created manually within the user program through the use of .WORD directives, contains information describing a file that the user intends to open during the

course of program execution. In creating this structure, any one or all of three possible string descriptors may be defined for a particular file, as follows:

1. A 2-word descriptor for an ASCII device name string;

2. A 2-word descriptor for an ASCII file directory string; and/or

3. A 2-word descriptor for an ASCII filename string.

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

DEVICE-NAME STRING DESCRIPTOR

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

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

```
DEVNM: .ASCII  /DK0:/

DEVNM: .ASCII  /TT10:/
```

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

DIRECTORY STRING DESCRIPTOR

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

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

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

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

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

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

FILENAME STRING DESCRIPTOR

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

This string consists of a filename up to 9 characters in length, an optional 3-character file type designator, and an optional file version number. The filename and file type must be

separated by a dot (.), and the file version number must be preceded by a semicolon. A filename string may be created as shown below:

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

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

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

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

```
FDBOUT: FDBDF$                      ;CREATES FDB.
        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 labelled FDBOUT is created. Observe further that the FDOP$A macro call takes as its second parameter the symbol OFDSPT. This symbol represents the address value stored in FDB offset location F.DSPT. This value enables the .PARSE routine (see Section 4.7.1) to access the fields of the dataset descriptor in building the filename block.

The symbol OFDSPT also appears in the label field of the first .WORD directive, defining the address of the dataset descriptor for the .PARSE routine. The .WORD directives each allocate 2 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 2 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 with the dataset descriptor. For each such ASCII string referenced in the dataset descriptor, however, corresponding statements must appear elsewhere in the source program to define the appropriate ASCII data string(s).

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

2.4.2 Default Filename Block - NMBLK$ Macro Call

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

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

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

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

where:

- label — represents a user-defined symbol that names the default filename block and defines its address. This label is the symbolic value 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 — represents the default filename. This parameter may consist of up to 9 ASCII characters. The character string is stored as 6 bytes in Radix-50 format, starting at offset location N.FNAM of the default filename block.
- ftyp — represents the default file type. This parameter may consist of up to 3 ASCII characters. The character string is stored as 2 bytes in Radix-50 format in offset location N.FTYP of the default filename block.
- fver — represents 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 — represents the default name of the device upon which the volume containing the desired file is mounted. This parameter consists of 2 ASCII characters that are stored in offset location N.DVNM of the default filename block.
- unit — represents a binary value identifying which unit (among several like units) is to be used in processing the file. If specified, this numeric value is stored in offset location N.UNIT of the default filename block.

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

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

As evident from the foregoing, all the default information supplied in the NMBLK$ macro call is stored in the default filename block at offset locations 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:

1. All the file information required by FCS to open the file is not present in the dataset descriptor. Missing information is then sought in the default filename block by the .PARSE routine (see Section 4.7.1), which is automatically invoked as a result of issuing any version of the generalized OPEN$x macro call.

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

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

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

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

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

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

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

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

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

```
NBOF$L
```

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

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

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

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

2.4.3 Dynamic Processing of File Specifications

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

2.5 OPTIMIZING FILE ACCESS

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

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

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

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

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

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

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

2. As a result of initializing the filename block manually by using the .PARSE routine (see Section 4.7.1) and the .FIND routine (see Section 4.8.1).

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

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

To understand how to effect the process of opening a file by file ID, note that the initial issuance of the generalized OPEN$x macro call (see Section 3.1) for a given file first invokes the .PARSE routine (see Section 4.7.1). The .PARSE routine is automatically linked into the user program, along with the code for OPEN$x. This routine first zeros the filename block and then fills it in with information taken from the dataset descriptor and/or the default filename block.

Thus, issuing the generalized OPEN$x macro call results in the invocation of the .PARSE routine each time a file is opened. The .PARSE function, however, can be bypassed altogether in subsequent OPEN$x calls by saving and restoring the filename block before attempting to 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 user memory space and later restored to the FDB at the desired point in program flow for use in reopening that same file.

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

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

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

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

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

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

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

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

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

Although it is necessary to save only the file-identification, device-name, and unit-number fields of the filename block in anticipation of reopening a file by file ID, the user is advised to save the entire filename block. The filename, file-type, file-version, and directory-ID fields, etc., may also be relevant. For example, an OPEN$x, save, CLOSE$, restore, OPEN$x, and DELET$ sequence would require saving and restoring the entire filename block. Though the user 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.

2.5.2 Manually Initializing the Filename Block

In addition to saving and restoring the filename block in anticipation of reopening a file by file ID, the filename block can also be initialized manually. If the user chooses to do so, the .PARSE and .FIND routines (see Sections 4.7.1 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 the user program is operating in an overlaid environment. In this case, it is highly desirable that the code for opening a file be broken into small segments in the interest of conserving memory space. Since the body of code for the OPEN$x and .PARSE functions is sizable, two other types of macro calls for opening files are provided for use with overlaid programs. The OFID$ and OFNB$ macro calls (see Sections 3.5 and 3.6, respectively) are specifically designed for this purpose.

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

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

If conservation of memory is an objective, and if the user program will be opening both new and existing files, it is recommended that only the OFNB$ routine be included in one branch of the overlay; 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) zeros the file-identification field (N.FID) of the filename block. This action prevents the field (which pertains to a previous operation) from being used mistakenly to open a file for a current operation. Thus, if a user later intends to open a file by file ID using information presently in the filename block, the entire filename block (not just N.FID) must be saved before closing the file. Then, at the appropriate point in program flow, the filename block may be restored to open the desired file by file ID.

2.6 INITIALIZING THE FILE STORAGE REGION

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

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

2.6.1 FSRSZ$ - Initialize FSR at Assembly-Time

The 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 the user, or defaulted, during the macro call.

NOTE

The FSRSZ$ macro allocates the FCS impure area that is pointed to by a fixed location in user 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.

The format of the FSRSZ$ macro is:

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

where: fbufs represents a numeric value that is established by the user as follows:

1. 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 the user intends 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, as shown below:

```
FSRSZ$  0
```

No other parameters need be specified unless the function of the psect parameter (described below) is required.

2. 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, an RSX-11M user might want to access simultaneously three files for block I/O and two files for record I/O. Since RSX-11M provides single buffering only for record I/O, and since block I/O does not require pool space in the FSR, this user would specify the following FSRSZ$ macro call:

```
FSRSZ$  2
```

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

3. 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 an RSX-11D or IAS user's program will simultaneously access four disk files for record I/O operations. Assume further that the user wants double-buffering for three of the disk files and has, therefore, specified a multiple buffer count of 2 in the FDBF$A macro calls (refer to Section 2.2.1.6) for the associated files. This user would then issue the following FSRSZ$ 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 below), could also be specified as required.

bufsiz represents 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 2 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 the user wishes to explicitly specify block-buffer-pool requirements, the following formula must be applied:

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

where:

bsize1, bsize2, etc. — are 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 the user explicitly specifies block-buffer-pool requirements.

mbc1, mbc2, etc. — are the multiple buffer counts (refer to Section 2.2.1.6) specified for the respective files. For the purposes of this formula, RSX-11M users must use multiple buffer counts equal to 1.

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

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

psect — specifies 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.

2.6.2 FINIT$ - Initialize FSR at Run-Time

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

```
label:  FINIT$
```

where:

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

The FINIT$ macro call should be issued in the program's initialization code. 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 below. This requirement derives from the fact that such a program performs all its initialization at run-time, rather than at assembly-time.

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

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

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

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

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

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

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

2.7 INCREASING THE SIZE OF THE FILE STORAGE REGION

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

2.7.1 FSR-Extension Procedures for MACRO-11 Programs

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

1. Modify the parameters in the FSRSZ$ macro call to redefine the buffer-pool requirement of files open simultaneously for record I/O processing. Reassemble the program.

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

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

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

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

In calculating the total length of the FSR, either of the formulas below may be used:

1. length = (S.BFHD*fbufs)+bufsiz

2. length = fbufs*(S.BFHD+512.)

where:

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

```
BDOFF$  DEF$L
```

fbufs — is 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 2.6.1.

bufsiz — represents 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 2.6.1.

512. — is the standard default buffer size.

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

2.7.2 FSR-Extension Procedures for FORTRAN Programs

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

```
ACTFIL = files
```

where:

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

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

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

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

2.8 COORDINATING I/O OPERATIONS

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

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

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

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

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

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

2.8.1 Event Flags

I/O operations proceed concurrently with other system activity. After an I/O request has been queued, the system does not force an implied wait for the issuing task until the requested operation is completed. Rather, the operation proceeds in parallel with the execution of the issuing task, and it is the task's responsibility to synchronize the execution of I/O requests. Tasks use event flags in synchronizing

these activities. With respect to event flags, the system merely executes primitive operations that manipulate, test, and/or wait for these indicators of internal task activity.

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

Specific FDB-initialization and I/O-initiating macro calls in FCS enable the user to specify event flags, if desired, that are unique to 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 2.2.1.6 or 2.2.2, respectively).

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

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

Through event-associated system directives, the user can clear event flags, set event flags, test whether a specified event flag is set, or cause a task to be suspended until a specified event flag is set. These event-associated directives are described in detail in the Executive Reference Manual of the host operating system. The setting and checking of event flags allow tasks in a real-time system to communicate with each other and thereby synchronize their execution. Event flags and related device-dependencies are described in detail in the IAS Device Handlers Reference Manual and the RSX-11M/M-PLUS I/O Drivers Reference Manual.

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

2.8.2 I/O Status Block

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

This optional parameter, called the I/O status block address, is made available to FCS through any of the macro calls identified above. When this parameter is supplied, the system returns status information to a 2-word block reserved in the user program. Although the I/O status block is used principally as a QUEUE I/O housekeeping mechanism for containing certain device-dependent information, this area also contains information of particular interest to the user.

Specifically, the second word of the I/O status block is filled in with the number of bytes transferred during a READ$ or WRITE$ operation. When performing READ$ operations, it is good practice always 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 magnetic tape units, almost all virtual blocks are 512(10) bytes or less in length.) For WRITE$ operations, the specified number of bytes are always transferred; otherwise an error condition exists.

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

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

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

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

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

```
IOSTAT: .BLKW   2
```

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

2.8.3 AST Service Routine

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

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

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

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

If an AST service routine is not provided by the user, some other mechanism, such as event flags or the I/O status block, must be used as a means of determining block I/O completion. In the absence of such a routine, for example, the user may test the low-order byte of the first word in the I/O status block to determine if the block I/O transfer has been completed. A WAIT$ macro call (see Section 3.17) 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.

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

CHAPTER 3

FILE-PROCESSING MACRO CALLS

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

OPEN$ - To open and prepare a file for processing;

OPNS$ - To open and prepare a file for processing and to allow shared access to that file (depending on the mode of access);

OPNT$ - To create and open a temporary file for processing;

OFID$ - To open an existing file using file-identification information in the filename block;

OFNB$ - To open a file using filename information in the filename block;

CLOSE$ - To terminate file processing in an orderly manner;

GET$ - To read logical data records from a file;

GET$R - To read fixed-length records from a file in random mode;

GET$S - To read records from a file in sequential mode;

PUT$ - To write logical data records to a file;

PUT$R - To write fixed-length records to a file in random mode;

PUT$S - To write records to a file in sequential mode;

READ$ - To read virtual data blocks from a file;

WRITE$ - To write virtual data blocks to a file;

DELET$ - To remove a named file from the associated volume directory and to deallocate the space occupied by the file; and

WAIT$ - To suspend program execution until a requested block I/O operation is completed.

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

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

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

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

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

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

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

3.1 OPEN$x - GENERALIZED OPEN MACRO CALL

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

```
OPEN$x
```

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

R - Read an existing file;

W - Write (create) a new file;

M - Modify an existing file without changing its length;

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

or

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

NOTE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. Initializes the file attribute section of the file-header block. 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 a particular type of macro call is issued explicitly specifying 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$/PUT$) operations are to be used in processing the file.

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

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

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

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

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

NOTE

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

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

3.1.1 Format of Generalized OPEN$x Macro Call

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

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

where:

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

fdb — represents a symbolic value of the address of the associated FDB.

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

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

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

racc represents the record-access byte. One or more symbolic values may be specified in this field to initialize the record-access byte (F.RACC) in the associated FDB. Any combination of the following parameters may be specified by separating them with exclamation points:

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

FD.RAN - Indicates that random access to the file is to be used for record I/O (GET$/PUT$) operations. If this parameter is not specified, FCS uses sequential access by default. Refer to Section 1.5 for a description of random-access mode.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
OPEN$A  #OUTFDB
```

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

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

NOTE

R0 can only be used to pass the FDB address parameter. Any other use of R0 will fail when issuing the OPEN$A macro call.

3.1.2 FDB Requirements for Generalized OPEN$x Macro Call

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

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

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

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

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

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

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

      R.SEQ - Indicates that sequenced records are to be written into the file.

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

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

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

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

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

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

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

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

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

FD.RAN - Indicates that random-access mode (GET$/PUT$ record I/O) is to be used in processing the file. If this parameter is not specified, sequential-access mode results by default. Refer to Section 1.5 for a description of random-access mode.

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

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

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

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

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

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

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

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

   a. The override block size (ovbs parameter) must be specified in either the FDBF$A or the FDBF$R macro call to initialize FDB offset location F.OVBS. This parameter need be specified only if the standard default block size in effect for the associated device is to be overridden. The override block size is specified to improve I/O system throughput 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.)

   b. The multiple-buffer count (mbct parameter) must be specified in either the FDBF$A or the FDBF$R macro call to initialize FDB offset location F.MBCT. (The mbct parameter is not applicable to RSX-11M/M-PLUS.) 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 not overlaid, nor is it zeroed when the file is closed.

      If the multiple-buffer count is not established as described above, multiple-buffered operations can still be invoked by changing the default buffer count in the FSR. A default buffer count of 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 multiple buffering is to be employed, the appropriate control flag must be specified as the mbfg parameter in either the FDBF$A or the FDBF$R macro call to appropriately initialize FDB offset location F.MBFG. Either of two symbolic values may be specified for this purpose, as follows:

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

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

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

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

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

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

```
STBLK:  .BLKW   5
```

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

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

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

3.2 OPNS$x - OPEN FILE FOR SHARED ACCESS

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

3.3 OPNT$W - CREATE AND OPEN TEMPORARY FILE

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

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

In using the OPNT$W macro call, the user bears the responsibility for marking the temporary file for deletion, as described in the procedure below. Then, after all operations associated with that file are completed, closing the file 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$W macro call takes the same parameters as the generalized OPEN$x macro call, the former executes faster because no directory entries are made for a temporary file.

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

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

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

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

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

5. Reopen the file by issuing any of the FCS macro calls 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 below, to mark the file for deletion:

```
CALL    .MRKDL
```

   The .MRKDL routine is described in Section 4.13.1.

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

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

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

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

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

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

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

NOTE

If the OPNT$D macro call is issued for use with a temporary file containing sensitive information, it is recommended that the user 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

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

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

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

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

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

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

2. Call the .FIND routine (see Section 4.8.1). This routine locates an appropriate directory entry for the file (by filename) 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 call.

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

Unlike the other FCS macro calls for opening files, the OFID$x macro call requires a nonzero value in the first word of the file identification field (N.FID) in order 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.

3.6 OFNB$x OPEN FILE BY FILENAME BLOCK

The OFNB$x macro call is issued to open either an existing file or to create and open a new file using filename information in the filename block. Similar to the OFID$x macro call above, the OFNB$x call is

designed for use with overlaid programs. However, the OFNB$x macro call differs in two important respects: it can be issued to create a new file, and it can be issued to open a file by filename block.

The OFNB$x call has the same format and takes the same parameters as the generalized OPEN$x macro call as described in Section 3.1.1; i.e.,

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

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

1. The filename (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); and
6. The unit number (offset location N.UNIT).

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

3.6.1 Dataset Descriptor and/or Default Filename Block

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

1. Call the .PARSE routine (see Section 4.7.1). This routine takes information from a specified dataset descriptor and/or default filename block and fills in the appropriate offsets of a specified filename block.
2. Issue the OFNB$x macro call.

3.6.2 Default Filename Block Only

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

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

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

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

   b. Call the .GTDID routine (see Section 4.9.2) 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 call.

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

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

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

3.7 OPEN$ - GENERALIZED OPEN FOR SPECIFYING FILE ACCESS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

NOTE

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

3.8 CLOSE$ - CLOSE SPECIFIED FILE

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

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

3.8.1 Format of CLOSE$ Macro Call

The CLOSE$ macro call takes the following format:

```
CLOSE$  fdb,err
```

where:

- fdb — represents a symbolic value of the address of the associated FDB.
- err — represents the symbolic address of an optional user-coded error-handling routine.

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

```
CLOSE$  #FDBIN,CLSERR

CLOSE$  ,CLSERR

CLOSE$  R0
```

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

3.9 GET$ - READ LOGICAL RECORD

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

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

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

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

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

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

3.9.1 Format of GET$ Macro Call

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

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

where:

- fdb — represents a symbolic value of the address of the associated FDB.
- urba — represents the symbolic address of a user record buffer to be used for record I/O operations in move or locate mode. When specified, this parameter initializes FDB offset location F.URBD+2.
- urbs — represents a numeric value defining the size (in bytes) of the user record buffer. This parameter determines the largest record that can be placed in the user record buffer in move or locate mode. When specified, this parameter initializes offset location F.URBD in the associated FDB.
- err — represents the symbolic address of an optional user-coded error-handling routine.

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

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

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

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

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

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

```
GET$    R0,,,ERROR

GET$    ,#RECBUF,#25.,ERROR

GET$    #INFDB
```

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

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

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

NOTE

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

3.9.2 FDB Mechanics Relevant to GET$ Operations

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

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

If operating in random-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$R operation to point to the next record in the FSR block buffer. Thus, unless the user program alters the values in locations F.RCNM and F.RCNM+2 before each issuance of the GET$ or GET$R macro call, the next record in sequence is read. The specified user-record-buffer size (i.e., the urbs parameter) always determines the largest record that can be read during a GET$ operation.

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

3.9.2.2 **GET$ Operations in Locate Mode** - In performing GET$ Operations in locate mode, the user should take into account the following:

NOTE

In the following discussion, reference is made to the FSR block buffer. If big-buffering is enabled (that is, an ovbs parameter value is specified in the FDBF$x macro call 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 since 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.

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

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

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

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

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

3. If a GET$ operation is performed in locate mode, and the record is contained entirely within the FSR block buffer, the address of the record within the FSR block buffer and the size of that record are returned to offset locations F.NRBD+2 and F.NRBD, respectively, in the associated FDB. However, if that record crosses buffer boundaries, it is moved to the user record buffer. In this case, the address of the user record buffer and the size of the record are returned to offset locations F.NRBD+2 and F.NRBD, respectively.

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

3.10 GET$R - READ LOGICAL RECORD IN RANDOM MODE

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

The GET$ and GET$R macro calls are identical, except that the parameter list of GET$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$x macro call, then neither GET$ nor GET$R will read the desired record.

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

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

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

hrcnm — represents a numeric value specifying the high-order 15 bits of the number of the record to be read. This value is stored in FDB offset location F.RCNM. If specified, the combination of this parameter and the lrcnm parameter above determines the number of the desired record. Thus, an unsigned value having a total of 41 bits of magnitude may be used in defining the record number.

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

If F.RCNM is used to express a desired record number for any given GET$R operation, this cell must be cleared before issuing a subsequent GET$R macro call that requires 16 bits or less to express the desired record numberX 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$R macro call, the next sequential record is read since the record number in offset locations F.RCNM+2 and F.RCNM is automatically incremented with each GET$ operation. In the case of the first GET$R after opening the file, record number 1 is read, because the record number has been initialized to 0 by the OPEN. If other than the next sequential record is to be read, the user must explicitly specify the number of the desired record.

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

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

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

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

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

NOTE

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

3.11 GET$S - READ LOGICAL RECORD IN SEQUENTIAL MODE

The GET$S macro call is used to read logical records from a file in sequential mode. Although the routine invoked by the GET$S macro call requires less memory than that invoked by GET$ (see Section 3.9), GET$S has the same format and takes the same parameters. The GET$S

macro call is designed specifically for use in an overlaid environment 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 utilization 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 call is used to write 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 increments this value after each PUT$ or PUT$R operation to point to the next sequential record position. Thus, unless the user program alters this value before issuing another PUT$ 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 the user's choice. This buffer is not necessarily the user record buffer previously described in the context of record I/O operations. In other words, the user may build records in an area of a program apart from that normally defined by the user record-buffer descriptors in the FDB (F.URBD+2 and F.URBD). In this case, the address of the record buffer so used and the size of the record are specified in the PUT$ macro call, and the record thus built is then moved into the FSR block buffer.

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

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

A random PUT$ operation in locate mode requires the use of the .POSRC routine. This operation is described in detail in Section 4.9.2. Specific FDB requirements for PUT$ operations are presented in Section 3.12.2 below.

3.12.1 Format of PUT$ Macro Call

The PUT$ macro call takes the following format:

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

where: fdb represents a symbolic value of the address of the associated FDB.

nrba — represents the symbolic address of the next record buffer, i.e., the address of the record to be PUT$. This parameter initializes FDB offset location F.NRBD+2.

nrbs — represents a numeric value specifying the size of the next record buffer, i.e., the length of the record to be PUT$. This parameter initializes FDB offset location F.NRBD.

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

The following examples represent the uses of the PUT$ macro call:

```
PUT$    #FDBADR,,,ERRRT

PUT$    ,,#160.,ERRRT

PUT$    R0
```

In the first example, note that the next record-buffer address (nrba parameter) and the next record-buffer size (nrbs parameter) are null. These null specifications imply that the current values in offset locations F.NRBD+2 and F.NRBD of the associated FDB are suitable to the current operation. Note also that fixed-length records could also be written in locate mode by issuing this macro call.

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

The last 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 above example; it cannot be used to pass any other parameter in the PUT$ macro call.

3.12.2 FDB Mechanics Relevant to PUT$ Operations

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

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

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

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

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

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

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

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

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

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

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

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

When a file is initially opened for PUT$ operations in locate mode, FCS sets up offset location F.NRBD+2 to point to the area in the FSR block buffer where the next record is to be built. Then, each PUT$

operation thereafter in locate mode updates the address value in this cell to point to the area in the FSR block buffer where the next record is to be built. Thus, after each PUT$ operation in locate mode, F.NRBD+2 points to the area where the next record is to be built. This logic dictates whether the user record buffer is required in locate mode.

In this regard, the following generalizations apply:

NOTE

In the following discussion, reference is made to the FSR block buffer. If big buffering is enabled (that is, an ovbs parameter value is specified in the FDBF$x macro call, as described in Section 2.2.1.6) the FSR block buffer will be more than 1 block long. As a result, it may not be necessary to move a record even though it crosses block boundaries since 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.

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

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

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

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

   Because the records are variable in length, it must be assumed that the largest possible record is PUT$, as defined by the size of the user record buffer (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 the user record buffer.

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

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

If the largest record that the user intends to PUT$ is 80 bytes, for example, then the largest defined record size should not be specified as 132 bytes (or any length larger than that intended to be PUT$). Aside from having to allocate a smaller user record buffer, PUT$ operations in locate mode are more efficient if this precaution is observed. Exercising care in this regard reduces the tendency to move records from the user record buffer to the FSR block buffer when they might otherwise be built directly in the FSR block buffer.

3.13 PUT$R - WRITE LOGICAL RECORD IN RANDOM MODE

The PUT$R macro call is used to write fixed-length records to a file in random mode. As noted in Section 3.10 in connection with the GET$R macro call, operations in random-access mode require the user to be intimately familiar with the contents of such files. The PUT$R macro call likewise relies entirely on the user for the specification of the number of the record before a specified PUT$ operation can be performed. Since the usual purpose of a PUT$R operation is to update known records in a file, it is assumed that the user also knows the number of such records within the file.

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

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

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

where: lrcnm represents a numeric value specifying the low-order 16 bits of the number of the record to be processed. This parameter serves the same purpose as the corresponding parameter in the GET$R macro call (see Section 3.10), except that it identifies the record to be written.

hrcnm — represents a numeric value specifying the high-order 15 bits of the number of the record to be processed. This parameter serves the same purpose as the corresponding parameter in the GET$R macro call, except that it identifies the record to be written.

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

If F.RCNM is used in expressing a desired record number for any given PUT$R operation, the user must clear this cell before issuing a subsequent PUT$R macro call that requires 16 bits or less in expressing the desired record number; otherwise, any residual value in F.RCNM results in an incorrect record number.

The lrcnm and hrcnm parameters initialize offset locations F.RCNM+2 and F.RCNM, respectively, in the associated FDB. If these values are not specified in a subsequent PUT$R macro call, the next sequential record is written, since FCS automatically increments the record number in these cells after each PUT$ 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, the user must explicitly specify the number of the desired record.

NOTE

A random mode PUT$ operation executed in locate mode must be preceded by a call to .POSRC. Since locate mode allows the user 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 the use of the PUT$R macro call follow:

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

PUT$R   #FDBADR,#RECBUF,,R4

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

In the first example, the presence of RECBUF as the next record buffer address (nrba) parameter merely indicates that the user is specifying the address of the record. Although specifying this address repeatedly is unnecessary, it is not invalid. Normally, a buffer address is specified dynamically, since other PUT$ macro calls may be referencing different areas in memory; thus, the address of the record must be explicitly specified in each PUT$ macro call. Note also that the next record buffer size (nrbs) parameter is null, since this parameter is required only in the case of writing variable-length records. Also, the second of the two available parameters for defining the record number is null.

Note in the second and third examples that R4 and a memory location (LRN) are used to specify the logical record number. Such a specification assumes that the user has preset the desired record number in the referenced location.

NOTE

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

3.14 PUT$S - WRITE LOGICAL RECORD IN SEQUENTIAL MODE

The PUT$S macro call is used to write logical records to a file in sequential mode. Although the routine invoked by the PUT$S macro call requires less memory than that invoked by PUT$ (see Section 3.12), PUT$S has the same format and takes the same parameters. The PUT$S macro call is designed specifically for use in an overlaid environment 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 call is issued to read a virtual block of data from a block-oriented device (e.g., a disk or DECtape). In addition, if certain optional parameters are specified in the READ$ macro call, status information is returned to the I/O status block (see Section 2.8.2), and/or the program traps to a user-coded AST service routine at the completion of block I/O operations (see Section 2.8.3).

In issuing the READ$ (or WRITE$) macro call, the user is responsible for synchronizing all block I/O operations. For this reason, the WAIT$ macro call is provided (see Section 3.17), allowing the user to suspend program execution until a specified READ$/WRITE$ operation has been completed. It is important, however, that the user test the contents of F.ERR in the FDB for error codes immediately after issuing the READ$/WRITE$ 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$/WRITE operations can return error codes distinct from those that can be present on completion of a WAIT$ operation. For example, IE.EOF will be returned upon completion of the READ$ operation, but not upon completing of WAIT$.

When the WAIT$ macro call is issued in conjunction with a READ$ (or WRITE$) macro call, the user must ensure that the event flag number and the I/O status block address specified in both macro calls are the same.

When the WTSE$ macro call is issued to wait 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.

NOTE

The READ$ macro call cannot be used when accessing ANSI magtapes.

3.15.1 Format of READ$ Macro Call

From the format below, note that the parameters of the READ$ macro call are identical to those of the FDBK$A or the FDBK$R macro call, with the exception of the fdb and err parameters. Certain FDB parameters may be set at assembly-time (FDBK$A), initialized at run-time (FDBK$R), or set dynamically by the READ$ macro call. In any case, certain information must be present in the FDB before the specified READ$ (or WRITE$) operation can be performed. These requirements are noted in Section 3.15.2 below.

The READ$ macro call takes the following format:

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

where:

fdb — represents a symbolic value of the address of the associated FDB.

bkda — represents the symbolic address of the block I/O buffer in the user program. This parameter need not be specified if offset location F.BKDS+2 has been previously initialized through either the FDBK$A or the FDBK$R macro call.

bkds — represents a numeric value specifying the size (in bytes) of the virtual block to be read. This parameter need not be specified if offset location F.BKDS has been previously initialized through either the FDBK$A or the FDBK$R macro call. In any case, the maximum block size that may be specified for file-structured devices is 32256 bytes.

bkvb — represents the symbolic address of a 2-word block in the user program containing the number of the virtual block to be read. This parameter causes offset locations F.BKVB and F.BKVB+2 to be initialized with the virtual-block number; F.BKVB+2 contains the low-order 16 bits of the virtual-block number, and F.BKVB contains the high-order 15 bits.

As noted in connection with the FDBK$A macro call described in Section 2.2.1.4, assembly-time initialization of the virtual-block number in the FDB is ineffective, since the generalized OPEN$x macro call sets the virtual-block number in the FDB to 1. The virtual-block number can be made available to FCS only through the FDBK$R macro call or the I/O-initiating READ$ (or WRITE$) macro call after the file has been opened. The virtual-block number is created as described in Item 4 of Section 2.2.2.1.

The READ$ function checks the specified virtual block number to ensure that it does not reference a nonexistent block, i.e., 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 the user 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 above, automatic sequential operation results by default, beginning with virtual-block number 1. The virtual-block number is automatically incremented by the number of blocks read after each READ$ operation is performed.

bkef — represents a numeric value specifying the event flag number to be used for synchronizing block I/O operations. This event flag number is used by FCS to signal the completion of the specified block I/O operation. The event flag number, which may also be specified in either the FDBK$A or the FDBK$R macro call, initializes FDB offset location F.BKEF; if so specified, this parameter need not be included in the READ$ (or WRITE$) macro call.

If this optional parameter is not specified through any available means, event flag 32(10) is used by default. The function of an event flag is discussed in further detail in Section 2.8.1.

bkst — represents the symbolic address of the I/O status block in the user program (see Section 2.8.2). This parameter, which initializes offset location F.BKST, is optional. The I/O status block is filled in by the system when the requested block I/O transfer is completed, indicating the success/failure of the requested operation.

The address of the I/O status block may also be specified in either the FDBK$A or the FDBK$R macro call. If the address of this 2-word structure is not supplied to FCS through any of the available means, status information is not returned to the I/O status block. However, the event flag specified through the bkef parameter above is set to indicate block I/O completion, but the user program must assume that the operation was successful. An error indication cannot be returned to the user program without an I/O status block address.

bkdn — represents the symbolic entry-point address of an AST service routine (see Section 2.8.3). If this parameter is specified, a trap occurs upon completion of the specified READ$ (or WRITE$) operation. This parameter, which is optional, initializes offset location F.BKDN. This address value may also be made available to FCS through either the FDBK$A or the FDBK$R macro call, and, if so specified, need not be present in the READ$ (or WRITE$) macro call.

If the address of an AST service routine is not specified through any available means, no AST trap occurs at the completion of block I/O operations.

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

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

```
READ$   R0

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

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

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

The first example assumes that R0 contains the address of the associated FDB. Also, all other required FDB initialization has been accomplished through either the FDBK$A or the FDBK$R macro call.

The second example shows an explicit declaration of the associated FDB and includes the symbolic address of a user-coded error-handling routine.

In the third example, R0 again contains the address of the associated FDB. The block-buffer address and the size of the block are specified next in symbolic form. The address of the 2-word block in the user program containing the virtual-block number is not specified, as indicated by the additional comma in the parameter string. The event flag number, the address of the I/O status block, and the address of the AST service routine then follow in order. Finally, the symbolic address of an optional error routine is specified.

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

NOTE

R0 can only be used to pass the FDB address. Any other use of R0 when issuing the READ$ macro call will fail.

3.15.2 FDB Requirements for READ$ Macro Call

The READ$ macro call requires that the associated FDB be initialized with certain values before it can be issued. These values may be specified through either the FDBK$A or the FDBK$R macro call, or they

may be made available to the FDB through the various parameters of the READ$ macro call. In any case, the following values must be present in the FDB to enable READ$ operations to be performed:

1. The block-buffer address (in offset location F.BKDS+2);
2. The block byte count (in offset location F.BKDS); and
3. The virtual-block number (in offset locations F.BKVB+2 and F.BKVB).

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, F.FFBY may be modified before closing the file using the CLOSE$ macro call.

3.16 WRITE$ - WRITE VIRTUAL BLOCK

The WRITE$ macro call is issued to write a virtual block of data to a block-oriented device (e.g., disk, DECtape, or DECtape II). Like the READ$ macro call, if certain optional parameters are specified in the WRITE$ macro call, status information is returned to the I/O status block (see Section 2.8.2), and, at the completion of the I/O transfer, the program traps to an AST service routine that is supplied to coordinate asynchronous block I/O operations (see Section 2.8.3).

Whether or not the address of an AST service routine and/or an event flag number is supplied, the user is responsible for synchronizing all block I/O processing. The WAIT$ macro call can be issued in conjunction with the WRITE$ macro call to suspend program execution until a program-dependent I/O transfer has been completed. When the WAIT$ macro call is used for this purpose, the event flag number and the I/O status block address in both macro calls must be the same. Again, as with READ$ operations, the user should check for an error code immediately following the WRITE$ macro call as well as on return from the WAIT$ macro call.

NOTE

The WRITE$ macro call cannot be used when accessing ANSI magtapes.

3.16.1 Format of WRITE$ Macro Call

The WRITE$ macro call takes the same parameters as the READ$ macro call, as shown below. However, the bkvb parameter in this case represents the number of the virtual block to be written. The virtual-block number is incremented automatically after each WRITE$ operation is performed.

The WRITE$ macro call has the following format:

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

When this macro call is issued, the virtual-block number (i.e., the bkvb parameter) is checked to ensure that it references a block within the file's allocated space; if it does, the block is written. If the specified block is not within the file's allocated space, FCS attempts to extend the file. If this attempt is successful, the block is written; if not, an error code indicating the reason for the failure of the extend operation is returned to the I/O status block and to offset location F.ERR of the associated FDB.

If FCS determines that the file must be extended, the actual extend operation is performed synchronously. After the extend operation has been successfully completed, the WRITE$ operation is queued, and only then is control returned to the instruction immediately following the WRITE$ macro call.

The following examples illustrate WRITE$ macro calls:

```
WRITE$  R0

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

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

The first example specifies only the FDB address and assumes that all other required values are present in the FDB. The second example reflects explicit declarations for the FDB, the block-buffer address, the block-buffer size, the virtual block number address, and the event flag number for signalling block I/O completion. The third example shows null specifications for three parameter fields, then continues with the event flag number, the address of the I/O status block, and the address of the AST service routine. Finally, the address of a user-coded error-handling routine is specified.

NOTE

R0 can only be used to pass the FDB address. Any other use of R0 when issuing the WRITE$ macro call will fail.

3.16.2 FDB Requirements for WRITE$ Macro Call

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

3.17 WAIT$ - WAIT FOR BLOCK I/O COMPLETION

The WAIT$ macro call, which is issued only in connection with READ$ and WRITE$ operations, causes program execution to be suspended until the requested block I/O transfer is completed. This macro call may be used to synchronize a block I/O operation 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 call, the user may 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, the user may issue a WAIT$ macro call (specifying the same event flag number and I/O status block address) following the READ$ (or WRITE$) macro call.

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

The system returns the final status of the I/O operation to the I/O status block (see Section 2.8.2) upon completion of the requested operation. A positive value (+) indicates successful completion, and a negative value (-) indicates unsuccessful completion.

Event flags are discussed in further detail in Section 2.8.1.

3.17.1 Format of WAIT$ Macro Call

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

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

where:

fdb — represents a symbolic value of the address of the associated FDB.

bkef — represents a numeric value specifying the event flag number to be used for synchronizing block I/O operations. The WAIT$ macro causes task execution to be suspended by invoking the WAITFOR system directive. This parameter must agree with the corresponding (bkef) parameter in the associated READ$/WRITE$ macro call.

If this parameter is not specified, either in the WAIT$ macro call or the associated READ$/WRITE macro call, FDB offset location F.BKEF is assumed to contain the desired event flag number, as previously initialized through the bkef parameter of the FDBK$A or the FDBK$R macro call.

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

If this parameter is not specified, either in the WAIT$ macro call or the associated READ$/WRITE$ macro call, FDB offset location F.BKST is assumed to contain the address of the I/O status block, as previously initialized through the bkst parameter of the FDBK$A or the FDBK$R macro call. If F.BKST has not been initialized, no return of information to the I/O status block occurs.

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

The following statements represent WAIT$ macro calls:

```
WAIT$   R0

WAIT$   #INFDB,#25.

WAIT$   R0,#25.,#IOSTAT

WAIT$   R0,,#IOSTAT,ERRLOC
```

The first example assumes that R0 contains the address of the associated FDB; furthermore, since the event flag number (bkef parameter) is not specified, offset location F.BKEF is assumed to contain the desired event flag number. If this cell in the FDB contains 0, event flag number 32(10) is used by default.

The second example shows an explicit specification of the FDB address and also specifies 25(10) as the event flag number. Again, in this example, the FDB is assumed to contain the address of the I/O status block. In contrast, the third example shows an explicit specification for the address of the I/O status block.

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

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

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

When performing several asynchronous transfers in the same general sequence as above, a separate buffer, I/O status block, and event flag must be maintained for each operation. If the user intends to wait for the completion of a given transfer, the appropriate event flag number and I/O status block address must be specified in the associated WAIT$ macro call.

NOTE

R0 can only be used to pass the FDB address. Any other use of R0 when issuing the WAIT$ macro call will fail.

3.18 DELET$ - DELETE SPECIFIED FILE

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

This macro call can be issued for a file that is either open or closed. If issued for an open file, that file is then closed and deleted; if issued for a closed file, that file is deleted only if the filename string specified in the associated dataset descriptor or default filename block contains an explicit file version number.

Thus, if the file is not open, and the file version number is 0 (indicating the latest version), or if the file version number is -1 (indicating the oldest version), then the DELET$ operation fails.

NOTE

If the DELET$ macro call is issued for use with a file containing sensitive information, it is recommended that the user 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 users.)

3.18.1 Format of DELET$ Macro Call

The DELET$ macro call takes the following format:

```
DELET$ fdb,err
```

where:

- fdb — represents a symbolic value of the address of the associated FDB.
- err — represents the symbolic address of an optional user-coded error-handling routine.

The following statements illustrate DELET$ macro calls:

```
DELET$  R0

DELET$  #OUTFDB,ERRLOC

DELET$  R0,ERRLOC
```

CHAPTER 4

FILE CONTROL ROUTINES

File control routines can be invoked in MACRO-11 programs to perform the following functions:

- Read or write default directory-string descriptors in $$FSR2.
- Read or write the default UIC word in $$FSR2.
- Read or write the default file-protection word in $$FSR2.
- Read or write the file-owner word in $$FSR2.
- Convert a directory string from ASCII to binary, or vice versa.
- Fill in all or part of a filename block from a dataset descriptor and/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 op-code/macro is used to invoke file control routines (JSR PC, dst). These routines are included from the system object library ([1,1]SYSLIB.OLB) at task-build time and incorporated into the user task. The file control routines are called as shown below:

```
CALL    .RDFDR

CALL    .EXTND
```

Before the CALL is issued, certain file control routines require that specific registers be preset with requisite information. These requirements are identified in the respective descriptions of the routines. Upon return, all registers are preserved, except those explicitly specified as changed.

As a general rule, if an error is detected by a file control routine, the C-bit (carry condition code) in the Processor Status Word is set, and an error indication is returned to FDB offset location F.ERR. However, certain file control routines do not return error indications because of the specific nature of their functions. The following file control routines are listed according to whether or not they return error indications.

Normal Error Return (C-bit and F.ERR)	No Error Return
.ASCPP	.RDFDR
.PARSE	.WDFDR
.PRSDV	.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 indicators that are placed in FDB offset location F.ERR by the routines identified above.

4.2 DEFAULT DIRECTORY-STRING ROUTINES

The .RDFDR and .WDFDR routines are used to read and write directory-string descriptors.

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

The user calls the .RDFDR routine to read 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 constitutes the default directory that is to be used by FCS when one is not explicitly specified in a dataset descriptor.

If the user has not established default directory string descriptor words in $$FSR2 through the .WDFDR routine described below, the descriptor words in $$FSR2 are null and FCS uses a default directory (when one is not specified in a dataset descriptor) corresponding to the 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 $$FSR2.

R2 Contains the address of the default directory string in $$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

The .WDFDR routine is called to create default directory-string descriptor words in $$FSR2. For example, if a user program is to operate on files in the directory [220,220], regardless of the UIC the program runs under, then the user can establish default directory-string descriptor cells in $$FSR2 to point to the alternate directory string [220,220] created elsewhere in the program. To do this, the desired directory string is first created through an .ASCII directive. Then, by calling the .WDFDR routine, the default directory-string descriptor cells in $$FSR2 are initialized to point to the new directory string.

Assume that the task is currently running under default UIC [200,200]. By issuing a MACRO-11 directive similar to the following:

```
NEWDDS:    .ASCII /[220,220]/
```

a new directory string is defined. Then, by calling the .WDFDR routine, the user can initialize string descriptor cells in $$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.

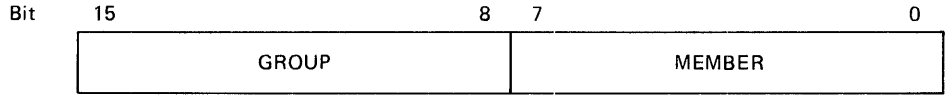
NOTE

Establishing default directory-string descriptor words in $$FSR2 does not change the default UIC in $$FSR2 or the task's privileges.

4.3 DEFAULT UIC ROUTINES

The .RDFUI and .WDFUI routines are used to read and write the default UIC maintained in program section $$FSR2 of the file storage region (FSR). Unlike the default directory string descriptor which describes

an ASCII string, the default UIC is maintained as a binary value with the following format:



The default UIC in $$FSR2 provides directory identification information for a file being accessed. FCS uses it only when all other sources of such information have failed to specify a directory (refer to Section 4.7.1.2). It is never used to establish file ownership or file access privileges.

Unless the user explicitly changes the default UIC through the .WDFUI routine described below, the default UIC in $$FSR2 always corresponds to the UIC under which the task is running.

4.3.1 .RDFUI - Read Default UIC

When called, the .RDFUI routine returns 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

The .WDFUI routine is called to create a new default UIC in $$FSR2.

The following register must be preset before calling the .WDFUI routine:

R1 Must contain the binary representation (as shown above) of a UIC.

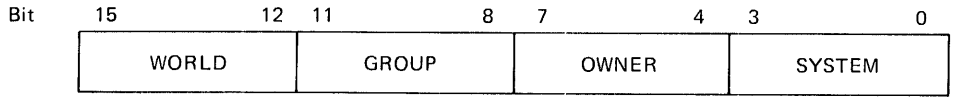
NOTE

The .WDFUI routine overrides any previous .WDFDR routine - created default UIC descriptor present in $$FSR2.

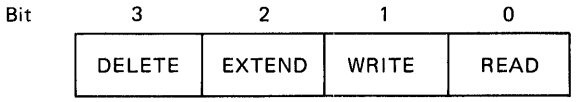
4.4 DEFAULT FILE-PROTECTION WORD ROUTINES

The .RDFFP and .WDFFP routines described below are used to read and write the default file protection word in a location in program section $$FSR2 of the file storage region (FSR). This word is used only at file-creation time (e.g., by the OPEN$W macro call) to establish the default file protection values for the new file. Unless altered, this value constitutes the default file-protection word for that file. If the value is -1, it indicates that the volume default file-protection value is to be used for the new file.

The default file-protection word has the following format:



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



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

4.4.1 .RDFFP - Read $$FSR2 Default File Protection Word

The user calls the .RDFFP routine to read the default file-protection word in program section $$FSR2 of the FSR. No registers need be set before calling this routine.

When called, the .RDFFP routine returns the following information:

R1 Contains the default file-protection word from $$FSR2.

4.4.2 .WDFFP - Write New $$FSR2 Default File-Protection Word

The .WDFFP routine is used to write a new default file-protection word into $$FSR2.

The following register must be preset before calling this routine:

R1 Must contain the new default file-protection word to be written into $$FSR2. If this register is set to -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 above, 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:

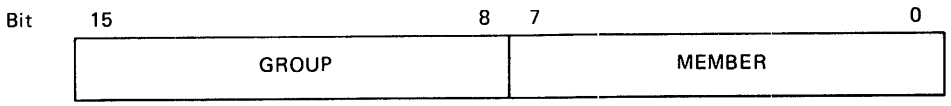
1. If the volume on which the new file is created is private (allocated), the owner's UIC is the same as the UIC of the task creating the file.

2. If the volume on which the new file is created is a system volume, the owner's UIC is the same as the task's login UIC.

Note that for files created by privileged or nonprivileged tasks that are started by a time-scheduled request, the owner's UIC is set to the UIC specified at task build time.

A specific UIC value can be stored in the file owner word via the .WFOWN routine (see section 4.5.2). All new files then created and closed by the user task will contain the specified UIC value.

The format of the file-owner word is shown below:



The routines for reading and writing the file-owner word are described below.

NOTES

1. 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. If this condition prevails, a privilege violation results.

2. When a file is created, the owner's UIC is always set to either the task UIC creating the file or the task's login UIC, as previously described. However, when closing the file, the owner's UIC can be changed using the .WFOWN routine. If the file is not closed properly, the owner's UIC will not change.

4.5.1 .RFOWN - Read $$FSR2 File-Owner Word

The .RFOWN routine is used to read the contents of the file-owner word in $$FSR2. No registers need be preset before calling this routine.

When called, the .RFOWN routine returns the following information:

R1 Contains the file-owner word (UIC). 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

The .WFOWN routine is used to initialize the file-owner word in $$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

The .ASCPP and .PPASC routines are called to convert a directory string from ASCII to binary, or vice versa.

4.6.1 .ASCPP - Convert ASCII Directory String to Equivalent Binary UIC

The .ASCPP routine is called to convert an ASCII directory string to its corresponding binary UIC.

The following registers must be preset before calling this routine:

R2 Must contain the address of the directory-string descriptor in the user program (see Section 2.4.1) for the string to be converted.

R3 Must contain the address of a word location in the user program to which the binary UIC is to be returned. The member number is stored in the low-order byte of the word, and the group number is stored in the high-order byte.

4.6.2 .PPASC - Convert UIC to ASCII Directory String

The .PPASC routine is called to convert a binary UIC to its corresponding ASCII directory string.

The following registers must be preset before calling this routine:

R2 Must contain the address of a storage area within the user program into which the ASCII string is to be placed. The resultant string can be up to 9 bytes in length, e.g., [200,200].

R3 Must contain the binary UIC value to be converted. The low-order byte of the register contains the member number, and the high-order byte of the register contains the group number.

R4 Must contain a control code. Bits 0 and 1 of this register indicate the following:

Bit 0 is set to 0 to suppress leading zeros (e.g., 001 is returned as 1). Bit 0 is set to 1 to indicate that leading zeros are not to be suppressed.

Bit 1 is set to 0 to place separators in the directory string (e.g., [10,20]. Bit 1 is set to 1 to suppress separators (e.g., 1020).

The .PPASC routine increments the contents of R2 to point to the byte immediately following the last byte in the converted directory string.

4.7 FILENAME BLOCK ROUTINES

The .PARSE, .PRSDV, .PRSDI, .PRSFN, and .ASLUN routines are available for performing functions related to a specified filename block. These routines are described in the following sections.

4.7.1 .PARSE - Fill in All Filename Information

When called, the .PARSE routine first zeros the filename block pointed to by R1 and then stores the following information in the filename block:

1. The ASCII device name (N.DVNM);
2. The binary unit number (N.UNIT);
3. The directory ID (N.DID);
4. The Radix-50 filename (N.FNAM);
5. The Radix-50 file type or extension (N.FTYP); and
6. The binary file version number (N.FVER).

In addition, .PARSE calls .ASLUN to assign the LUN associated with the FDB to the device and unit currently specified in the filename block.

The format of a filename block is shown in detail in Appendix B.

Before the .PARSE routine can be called, the FINIT$ macro call (see Section 2.6) must be invoked explicitly in the user program, or it must be invoked implicitly through a prior OPEN$x macro call. Note, however, that the FINIT$ call must be issued only once in the initialization section of the program, i.e., the FINIT$ operation must be performed only once per task execution. Furthermore, FORTRAN programs issue a FINIT$ call at the beginning of task execution; therefore, MACRO-11 routines used with the FORTRAN object time system must not issue a FINIT$ macro call.

The following registers must be preset before calling the .PARSE routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of the filename block to be filled in. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0 (i.e., R0 + F.FNB).

R2 Must contain the address of the desired dataset descriptor if .PARSE is to access a dataset descriptor in building the specified filename block. This structure is usually, but not necessarily, the same as that associated with the FDB specified in R0, i.e., the dataset descriptor pointed to by the address value in F.DSPT.

If R2 contains 0, this value implies that a dataset descriptor has not been defined; therefore, the dataset descriptor logic of .PARSE is bypassed.

R3 Must contain the address of the desired default filename block if .PARSE is to access a default filename block in building the specified filename block. This structure is usually, but not necessarily, the same as that associated with the FDB specified in R0, i.e., the default filename block pointed to by the address value in F.DFNB.

As above, if R3 contains zero (0), this value implies that a default filename block has not been defined; therefore, the default filename block logic of .PARSE is bypassed.

Thus, R0 and R1 each must contain the address of the appropriate data structure, while either R2 or R3 must contain the address of the desired filename information. Both R2 and R3, however, may contain address values if the referenced structures both contain information required in building the specified filename block.

The .PARSE routine fills in the specified filename block in the order described in the following sections.

4.7.1.1 **Device and Unit Information** - The .PARSE routine first attempts to fill in the filename block with device (N.DVNM) and unit (N.UNIT) information. The following operations are performed in sequence until the required information is obtained from the specified data structures:

1. If the address of a dataset descriptor is specified in R2 and this structure contains a device string, the device and unit information therein is moved into the specified filename block.

2. If Step 1 fails, and if the address of a default filename block is specified in R3, and this structure contains a nonzero value in the device-name field, the device and unit information therein is moved into the specified filename block.

3. If Step 2 fails, .PARSE uses the device and unit currently assigned to the logical unit number in offset location F.LUN of the specified FDB in building the filename block.

   This feature allows a program to use preassigned logical units that are assigned through either the device assignment (ASG) option of the Task Builder or one of the following commands: ASSIGN (under IAS) or REASSIGN (under RSX-11). In this case, the user simply avoids specifying the device string in the dataset descriptor and the device name in the default filename block.

4. If the logical unit number in F.LUN is currently unassigned, .PARSE assigns this number to the system device (SY0:).

Once the device and unit are determined and the logical unit number is assigned, .PARSE invokes the GLUN$ directive to obtain necessary device information. Requisite information is returned to the following offsets in the filename block pointed to by R1:

N.DVNM - Device Name Field. Contains the redirected device name.

N.UNIT - Unit Number Field. Contains the redirected unit number.

In addition, requisite information is returned to the following offsets in the FDB pointed to by R0:

F.RCTL - Device Characteristics Byte. This cell contains device-dependent information from the first byte of the third word returned by the GLUN$ directive. The bit definitions pertaining to the device characteristics byte are described in detail in Table A-1. If desired, the user can examine this cell in the FDB to determine the characteristics of the device associated with the assigned LUN.

F.VBSZ - Device Buffer-Size Word. This location contains the information from the sixth word returned by the GLUN$ directive. The value in this cell defines the device buffer size (in bytes) pertaining to the device associated with the assigned LUN.

The GLUN$ directive is described in detail in the Executive Reference Manual of the host operating system.

4.7.1.2 **Directory-Identification Information** - Following the operations described in the preceding section, .PARSE attempts to fill in the filename block with directory-identification information (N.DID). The precedence rules for establishing this information are as follows:

1. If the address of a dataset descriptor is specified in R2 and this structure contains a directory string, that directory string is used to find the associated UFD in the MFD, and the resulting file ID is then moved into the directory-ID field of the specified filename block.

2. If Step 1 fails, and if the address of a default filename block is specified in R3, and this structure contains a nonzero directory ID, it is moved into the specified filename block.

   Since none of the parameters of the NMBLK$ macro call (see Section 2.4.2) initialize the 3 words starting at offset location N.DID in the default filename block, these cells must be initialized manually, or they must be initialized by issuing a call to either the .GTDIR routine (see Section 4.9.1) or the .GTDID routine (see Section 4.9.2). Note that these routines can also be used to initialize a specified filename block directly with required directory information.

3. If neither Step 1 nor Step 2 yields the required directory string, .PARSE examines the default directory string words in $$FSR2. If the user 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 $$FSR2 as the directory identifier. Unless changed by the user through the .WDFUI routine, this word contains the UIC under which the task is running.

NOTE

Wild card UICs are not acceptable to .PARSE. In addition, .PARSE will not set either Filename-Block Status Word (N.STAT) bits NB.SD1 or NB.SD2 (group and owner wild card specifications, respectively).

4.7.1.3 **Filename, File-Type or Extension, and File Version Information** - Following the operations described in the preceding section, .PARSE attempts to obtain filename information (N.FNAM, N.FTYP, and N.FVER), as follows:

1. If the address of a dataset descriptor is specified in R2 and this structure contains a filename string, the filename information therein is moved into the specified filename block.

2. If the address of a default filename block is specified in R3, and one or more of the filename, file-type or extension, and file version number fields of the dataset descriptor specified in R2 are null, then the corresponding fields of the default filename block are used to fill in the specified filename block.

3. If neither Step 1 nor Step 2 yields the requisite filename information, any specific field(s) not available from either source remain(s) null.

NOTE

If a dot (.) appears in the filename string without an accompanying file type designation (e.g., TEST. or TEST.;3), the file type is interpreted as being explicitly null. In this case, the default file type is not used. Similarly, if a semicolon (;) appears in the filename string without an accompanying file version number (e.g., TEST.DAT;), the file version number is likewise interpreted as being explicitly null; again, the default file version number is not used in this case.

4.7.1.4 **Other Filename Block Information** - Finally, after performing all the operations above, the .PARSE routine also fills in the filename block status word (offset location N.STAT) of the filename block specified in R1. The bit definitions for this word are presented in Table B-2. Note in this table that an "explicit" directory, device, filename, file-type, or file version number specification pertains to ASCII data supplied through the dataset descriptor pointed to by R2.

In addition, .PARSE explicitly zeros offset location N.NEXT in the filename block pointed to by R1. This action has implications for wildcard operations, as described in Section 4.8.1 below.

4.7.2 .PRSDV - Fill in Device and Unit Information Only

The .PRSDV routine is identical to the .PARSE routine above, except that it performs only those operations associated with requisite device and unit information (see Section 4.7.1.1). This routine zeros the filename block pointed to by R1, performs a .PARSE operation on the device and unit fields in the specified dataset descriptor and/or default filename block, and assigns the logical unit number contained in offset location F.LUN of the specified FDB.

4.7.3 .PRSDI - Fill in Directory-Identification Information Only

The .PRSDV routine is identical to the .PARSE routine above, except that it performs only those operations associated with requisite directory-identification information (see Section 4.7.1.2). This routine zeros the filename block pointed to by R1 and performs a .PARSE operation on the directory-identification information (N.DID) field in the specified dataset descriptor and/or default filename block.

4.7.4 .PRSFN - Fill in Filename, File-Type or Extension, and File Version Only

The .PRSDV routine is identical to the .PARSE routine above, except that it performs only those operations associated with requisite filename, file-type or extension, and file version information (see Section 4.7.1.3). This routine zeros the filename block pointed to by R1 and performs a .PARSE operation on the filename, file-type or extension, and file version information fields (N.FNAM, N.FTYP, N.FVER) in the specified dataset descriptor and/or default filename block.

4.7.5 .ASLUN - Assign Logical Unit Number

The .ASLUN routine is called to assign a logical unit number to a specified device and unit and to return the device information to a specified FDB and filename block.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of the filename block containing the desired device and unit. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

If the device-name field (offset location N.DVNM) of the filename block pointed to by R1 contains a 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 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 device, the .ASLUN routine assigns it to the system device (SY0:) by default.

The information returned to the specified filename block and to the specified FDB is identical to that returned by the device and unit logic of the .PARSE routine (see Section 4.7.1.1).

4.8 DIRECTORY ENTRY ROUTINES

The .FIND, .ENTER, and .REMOV routines are used to find, insert, and delete directory entries. The term "directory entry" encompasses entries in both the master file directory (MFD) and the user file directory (UFD).

4.8.1 .FIND - Locate Directory Entry

The .FIND routine is called to locate a directory entry by filename and to fill in the file-identification field (N.FID) of a specified filename block.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

When invoked, the .FIND routine searches the directory file specified by the directory-ID field of the filename block. This file is searched for an entry that matches the specified filename, file type, and file version number. In this regard, two special file versions are defined:

Version 0 is matched by the latest (largest) version number encountered in the directory file.

Version -1 is matched by the oldest (smallest) version number encountered in the directory file.

If either of these special versions is specified in the filename block, the matching version number is returned to the filename block. In this way, the actual version number is made available to the program.

Certain wildcard operations require the use of the .FIND routine. Three bits in the filename-block status word (see N.STAT in Table B-2) indicate whether a wildcard (*) was specified for a filename, a file type, or a file version number field. If the wildcard bit in N.STAT is set for a given field, any directory entry matches in that corresponding field. Thus, if the filename and file version number fields contain wildcard specifications (*), and the file-type field is specified as .OBJ (i.e., *.OBJ;*), the first directory entry encountered that contains .OBJ in the file-type field matches, irrespective of the values present in the other two fields.

When a wildcard match is found, the complete filename, file type, and file version number fields of the matching entry are returned to the filename block, along with the file-ID field (N.DID). Thus, the program can determine the actual name of the file just found. Offset location N.NEXT in the filename block is also set to indicate where that directory entry was found in the directory file. This information is used in subsequent .FIND operations to locate the next matching entry in the directory file.

For example, the .FIND routine is often used to open a series of files when wildcard specifications are used. The following operations are typical:

1. Call the .PARSE routine. This routine zeros offset location N.NEXT in the filename block in preparation for the iterative .FIND operations described in Step 3 below.

2. Check for wildcard bits set by the .PARSE routine in the filename block status word (see N.STAT in Table B-2). An instruction sequence such as that shown below may be used to test for the setting of wildcard bits in N.STAT:

```
        BIT     #NB.SVR!NB.STP!NB.SNM,N.STAT(R1)

        BEQ     NOWILD          ;BRANCH IF NOT SET.
```

3. If wildcard specifications are present in the filename block status word, repeat the following sequence until all the desired wildcard files have been processed:

```
        CALL    .FIND

        BCS     DONE            ;ERROR CODE IE.NSF INDICATES
                                ;NORMAL TERMINATION.

        OPEN$   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 search of the directory file for a matching entry.

4. Perform the desired operations on the file.

NOTE

The above procedure applies only for the following types of wildcard file specifications:

```
TEST.DAT;*
TEST.*;*
*.DAT;*
TEST.*;5
*.DAT;3
```

The procedure does not work for the following types of wildcard file specifications:

```
*.DAT
TEST.*
```

In summary, if a wildcard file specification is present in either the filename field or the file-type field, the file version number field must also contain either an explicit wildcard specification (*) or a specific file version number (e.g., 2, 3, etc.). In the latter case, however, the version number cannot be 0, for the latest version of the file, or -1, for the oldest version of the file.

To delete a file whose file-descriptor entry in the FDB contains wildcards, the user must save the values in the fields N.STAT and N.NEXT in the FDB, then zero those 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

The .ENTER routine is used to insert an entry by filename into a directory.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

If the file version number field of the filename block contains 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

The .REMOV routine is called to delete an entry from a directory by filename. This routine only deletes a specified directory entry; it does not delete the associated file.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

Wildcard specifications operate in the same manner as for the .FIND routine described in section 4.8.1 above, except that the special file version numbers 0 and -1 are illegal. The file version number for .REMOV operations must be explicit or wildcard. Each .REMOV operation deletes the next directory entry having the specified filename, file type, and file version number.

4.9 FILENAME BLOCK ROUTINES

The .GTDIR and .GTDID routines are used to insert directory information in a specified filename block.

4.9.1 .GTDIR - Insert Directory Information in Filename Block

The .GTDIR routine is called to insert directory information taken from a directory-string descriptor into a specified filename block.

Before calling this routine, the following registers must be preset:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block in which the directory information is to be placed. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

R2 Must contain the address of the 2-word directory-string descriptor in the user program. This string descriptor defines the size and the address of the desired directory string.

This routine performs a .FIND operation for the specified user file directory (UFD) in the master file directory (MFD) and returns the resulting directory ID to the 3 words of the specified filename block, starting at offset location N.DID. The .GTDIR routine preserves the information in offset locations N.FNAM, N.FYTP, N.FVER, N.DVNM, and N.UNIT of the filename block, but zeros (clears) the rest of the filename block.

The .GTDIR routine can also be used in conjunction with the NMBLK$ macro call (see Section 2.4.2) to insert directory information into a specified default filename block.

4.9.2 .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 above, this routine uses the binary value found in the default UIC word maintained in $$FSR2 as the desired user file directory (UFD).

Before calling this routine, the following registers must be preset:

R0 Must contain the address of the desired FDB.

R1 Must contain the address of a filename block in which the directory information is to be placed. This filename block is usually, but not necessarily, the filename block within the FDB specified in R0.

When called, the .GTDID routine takes the default UIC from its one-word location in $$FSR2 and performs a .FIND operation for the associated user file directory (UFD) in the master file directory (MFD). The resulting directory ID is returned to the 3 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 zeros the rest of the filename block.

The .GTDID routine embodies 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 intended specifically for use in programs that open files via the OFNB$ macro call (see Section 3.6). Such a program does not invoke the .PARSE logic because all required filename information is provided to the program in filename block format.

As is true of the .GTDIR routine described in Section 4.9.1 above, .GTDID can be used in conjunction with the NMBLK$ macro call (see Section 2.4.2) to insert directory information (N.DID) into a specified default filename block. The user also has the option to initialize offset location N.DID manually with required directory information.

4.10 FILE POINTER ROUTINES

The .POINT, .POSRC, .MARK, and .POSIT routines are used to point to a byte or a record within a specified file.

4.10.1 .POINT - Position File to Specified Byte

The .POINT routine is called to position a file to a specified byte in a specified virtual block. If locate mode is in effect for record I/O operations, the .POINT routine also updates the value in offset location F.NRBD+2 in the associated FDB in preparation for a PUT$ operation in locate mode.

The following registers must be preset before calling this routine:

- R0 Must contain the address of the desired FDB.
- R1 Must contain the high-order bits of the virtual-block number.
- R2 Must contain the low-order bits of the virtual-block number.
- R3 Must contain the desired byte number within the specified virtual block.

For a description of virtual-block numbers and how these 2-word values are formed, refer to Item 4 in Section 2.2.2.1.

NOTE

The 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 often used in conjunction with the .MARK routine to achieve a limited degree of random access with variable-length records. The .MARK routine saves the positional context of a file in

anticipation of temporarily closing that file and then reopening it later at the same position. For such purposes, the following procedure applies:

1. Call the .MARK routine (see Section 4.10.3 below) to save the current positional context of the file.

2. Close the file.

3. When desired, reopen the file.

4. Load the information returned by the .MARK routine into R1, R2, and R3, as required above, before calling the .POINT routine.

5. Call the .POINT routine.

   The .POINT routine may be called to rewind a file on disk or ANSI magtape to its start. For this case, R0 and R3 must be 0, and R2 must be 1.

6. Resume processing of the file.

4.10.2 .POSRC - Position File to Specified Record

The .POSRC routine is called to position a file to a specified fixed-length record within a file. If locate mode is in effect for record I/O operations, the .POSRC routine also updates the value in offset location F.NRBD+2 in the associated FDB in preparation for a PUT$ operation in locate mode.

Before calling this routine, the user must set offset locations F.RCNM+2 and F.RCNM in the FDB to the desired record number and ensure that the correct record size is reflected in offset location F.RSIZ of the FDB.

Also, the register below must be preset before calling the .POSRC routine:

R0 Must contain the address of the associated FDB.

The .POSRC routine is used when performing random-access PUT$ operations in locate mode. Normally, PUT$ operations in locate mode are sequential; however, when random-access mode is used, the following procedure must be performed to ensure that the record is built at the desired location:

1. Set offset locations F.RCNM+2 and F.RCNM in the associated FDB to the desired record number.

2. Call the .POSRC routine.

3. Build the new record at the address returned (by the .POSRC call) in offset location F.NRBD+2 of the associated FDB.

4. Perform the PUT$ operation.

4.10.3 .MARK - Save Positional Context of File

The .MARK routine allows the user to record the current positional context of a file for later use. For example, the user may mark the current position of the file, close that file, and later reopen the file and return to the same position within the file. The .MARK routine is also useful in altering records within a file. After determining the record to be altered, the user may .MARK the file and retrieve information elsewhere in the file for use in updating the desired record. Then, by returning to the saved position of the file, the desired record may be altered. This iterative sequence may be repeated any number of times to update desired records in the file.

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 Positional Information for Specified Record

The .POSIT routine calculates the virtual-block number and the byte number pertaining to the beginning of a specified record.

The following register must be preset before calling this routine:

R0 Must contain the address of the associated FDB.

In addition, offset locations F.RCNM and F.RCNM+2 in the associated FDB must contain the desired record number.

Unlike the .POSRC routine above, which positions the file to the specified record, .POSIT simply calculates the positional information for a specified record so that a .POINT operation can be later performed to position to the desired record.

The register values returned by the .POSIT routine are identical to those described above for the .MARK routine.

4.11 QUEUE I/O FUNCTION ROUTINE (.XQIO)

The .XQIO routine is called to execute a specified Queue I/O function and to wait for its completion.

The following registers must be preset before calling this routine:

R0 Must contain the address of the desired FDB.

R1 Must contain the desired Queue I/O function code. Refer to the IAS Device Handlers Reference Manual or the RSX-11M/M-PLUS I/O Drivers Reference Manual for the desired Queue I/O directive function codes.

R2 Must contain the number of optional parameters to be included in the Queue I/O directive, if any.

R3 Must contain the beginning address of the list of optional Queue I/O directive parameters, if R2 contains a nonzero value.

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, the user must specify the address of an FDB containing filename information, a LUN, and an event flag number to be used in connection with renaming the file.

If the file to be renamed is open when the call to .RENAM is issued, that file is closed 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 originally named file.

R1 Must contain the address of the FDB containing the desired filename information, LUN assignment, and event flag to be associated with renaming the file.

If the renaming operation is successful, a new directory entry is created, and the original entry is deleted. If the operation is not successful, the file is closed under its original name, and the associated directory is not affected.

The .RENAM routine uses the absence of a value in location F.FNB+N.FID as an indication that .PARSE must be called to parse a file specification. If neither a dataset descriptor nor a default filename block is present, .PARSE returns a null filename. The rename operation then results in a new filename of ".;1".

NOTE

The renaming process is merely a directory operation that replaces an old entry with a new entry. The filename stored in the file-header block is not altered.

4.13 FILE EXTENSION ROUTINE (.EXTND)

The .EXTND routine is called to extend either contiguous or noncontiguous files. The file to be extended can be either open or closed.

The following registers must be preset before calling this routine:

R0 Must contain the address of the associated FDB.

R1 Must contain a numeric value specifying the number of blocks to be added to the file.

R2 Must contain the extension control bits, as appropriate. The possible bit configurations for controlling file extend operations are detailed in Table 4-1. This table defines the bits in the low-order byte of R2. The high-order 8 bits of R2 (Bits 8 through 15) are used in conjunction with the 16 bits of R1 to define the number of blocks to be added to the file (see Note below).

NOTES

1. The contents of R1 and the high-order byte of R2 (Bits 8 through 15) are used by FCS in accomplishing the specified .EXTND operation. Thus, 24 bits of magnitude are available for specifying the number of blocks by which the file is to be extended.

2. If a file previously had space allocated to it, a contiguous .EXTND (Bit EX.ACl=1) cannot be done.

3. When writing a new file using Queue I/O directives, the task must explicitly issue .EXTND calls as necessary to reserve enough blocks for the file, or the file must be initially created with enough blocks allocated for the file. 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.

4.14 FILE TRUNCATION ROUTINE (.TRNCL)

The .TRNCL routine truncates a file to its logical end-of-file point, 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.

4.15 FILE DELETION ROUTINES

The .MRKDL and .DLFNB routines are provided for deleting files.

NOTE

If the .MRKDL or .DLFNB routine is used to delete a file containing sensitive information, it is recommended that the user zero 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

The .MRKDL routine is used in conjunction with a temporary file, i.e., a file created through the OPNT$W macro call (see Section 3.3). Such a file has no associated directory entry.

A call to the .MRKDL routine is issued prior to closing a temporary file. The file so marked is then deleted automatically when the file is closed.

Table 4-1
R2 Control Bits for .EXTND Routine

BIT SETTINGS - Low-Order Byte of R2								BIT DEFINITIONS AND MEANING[1]
7	6	5	4	3	2	1	0	
								EX.ENA - Bit 7 = 0
0	x	x	x	x	x	x	0	EX.ACl - BIT 0 = 0; indicates that extend is to be noncontiguous.
0	x	x	x	x	x	x	1	EX.ACl - BIT 0 = 1; indicates that extend is to be contiguous and that file is to be contiguous.
								EX.ENA - Bit 7 = 1
1	x	x	x	x	x	x	0	EX.ACl - Bit 0 = 0; indicates that noncontiguous area is to be added to the file.
1	x	x	x	x	x	x	1	EX.ACl - Bit 0 = 1; indicates that contiguous area is to be added to the file.

[1] Bit settings must be defined by the user program.

(continued on next page)

Table 4-1 (Cont.)
R2 Control Bits for .EXTND Routine

BIT SETTINGS - Low-Order Byte of R2								BIT DEFINITIONS AND MEANING[1]
7	6	5	4	3	2	1	0	
1	x	x	x	x	x	1	x	EX.AC2 - Bit 1 = 1; indicates that the largest available contiguous area is to be added to the file if desired extend space is not available. This bit is set only if bit 0 in EX.AC1 is set to 1.
1	x	x	x	x	0	x	x	EX.FCO - Bit 2 = 0; indicates that the file is to be noncontiguous.
1	x	x	x	x	1	x	x	EX.FCO - Bit 2 = 1; indicates that the file is to be contiguous.
1	x	x	x	0	x	x	x	EX.ADF - Bit 3 = 0; indicates that the user intends to allocate the number of blocks specified by R1 and the high-order bits of R2 (see Note above).
1	x	x	x	1	x	x	x	EX.ADF - Bit 3 = 1; indicates that file extension is to occur according to the volume default extend value, as established by the INITIALIZE, INITVOLUME, or MOUNT command.

[1] Bit settings must be defined by the user program.

Before calling the .MRKDL routine, the following register must be preset:

R0 Must contain the address of the associated FDB. This FDB is assumed to contain the file identification, device name, and unit information pertaining to the file to be deleted.

If the .MRKDL routine is invoked while the temporary file is open, as is normally done, then the file is deleted unconditionally when it is closed, even if the calling task terminates abnormally without closing the file.

4.15.2 .DLFNB - Delete File by Filename Block

This routine is used to delete a file by filename block. The .DLFNB routine assumes that the filename block is completely filled in; when called, it closes the file, if necessary, and then deletes the file.

Before calling this routine, the following register must be preset:

R0 Must contain the address of the associated FDB.

The .DLFNB routine operates in the same manner as the routine invoked by the DELET$ macro call (see Section 3.18), but .DLFNB does not require any of the .PARSE logic and is thus considerably smaller (in

terms of memory requirements) than the normal DELET$ function. Like the DELET$ operation, however, if the file to be deleted is not currently open, and if an explicit file version number is not present in offset location N.FVER of the associated filename block, then the .DLFNB operation fails.

4.16 DEVICE CONTROL ROUTINE (.CTRL)

The .CTRL routine is called to perform device-specific control functions. The following are examples of .CTRL device-specific functions:

1. Rewind a magnetic tape volume set.
2. Position to the logical end of a magnetic tape volume set.
3. Close the current magnetic tape volume and continue file operations on the next volume.
4. Space forward or backward n records.
5. Rewind a file.

The following registers must be preset before calling this routine for items 1 to 3 above:

R0 Must contain the address of the associated FDB.

R1 Must contain one of the following function codes:

FF.RWD to rewind a magnetic tape volume set;

FF.POE to position to the logical end of a magnetic tape volume set;

FF.NV to close the current volume and continue file operations on the next volume of a magnetic tape volume set.

R2 and R3 must each contain 0.

When using .CTRL to space forward or backward, registers R0, R1, R2, and R3 must 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 records 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, register R1 must contain the value FF.RWF and registers R2 and R3 must contain 0.

See Chapter 5 for an explanation of the use of .CTRL to accomplish magnetic tape device-specific functions.

CHAPTER 5

FILE STRUCTURES

IAS and RSX-11 support an identical file structure on disk, DECtape, and DECtape II. They also support ANSI file structure on magnetic tape.

The disk, DECtape and DECtape II file structure is called FILES-11; the magnetic tape file structure is ANSI standard.

5.1 DISK AND DECTAPE FILE STRUCTURE (FILES-11)

Volumes contain both user files and system files. Disks and DECtapes initialized through the INITIALIZE (IAS) or INITVOLUME (RSX) command have the standard FILES-11 structure built for them automatically. The standard system files created through these commands include the following:

1. Index file
2. Storage-allocation file
3. Bad-block file
4. Master file directory (MFD)
5 Checkpoint file

Each FILES-11 volume has a file of each type. A volume may have more than one directory file; such files, created by the CREATE/DIRECTORY command in IAS, and the UFD command in RSX-11 systems, are used by the system to locate user files on the volume.

5.1.1 User File Structure

User data files on disk and DECtape consist of ordered sets of virtual blocks that constitute the virtual structure of the files as they appear to the user. Virtual blocks can be read and written directly by issuing READ$ and WRITE$ macro calls (see Sections 3.15 and 3.16, respectively). Virtual blocks are numbered in ascending sequence relative to the first block in the file (which is virtual block 1).

The virtual blocks of a file are stored on the volume as logical blocks. The logical-block size of all volumes is 256 words; thus, each virtual block is also 256 words. When access to a virtual block is requested, the virtual-block number is mapped into a logical-block number. The logical-block number is then mapped to the physical address on the associated volume.

5.1.2 Directory Files

A directory file contains directory entries. Each entry consists of a filename and its associated file number and file-sequence number. The number of 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, a master file directory (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). A user file directory is created by the UFD command in RSX-11 systems, and by the CREATE/DIRECTORY command in IAS. These commands are described in detail in the RSX-11M/M-PLUS MCR Operations Manual and the IAS System Management Guide.

Figures 5-1 and 5-2 illustrate the directory structure for single-user and multiuser volumes, respectively.

5.1.3 Index File

The index file contains volume information and user file-header blocks, which are used by the file control primitives (FCP). Because the file-header blocks (see below) are stored in the index file, they can be located quickly. Furthermore, since a file-header block is 256 words in length, it can be read into memory with a single access.

The index file is created when a volume is initialized for use by the host operating system. During initialization, the information required by the system is placed in the index file. Appendix E contains a detailed description of the format and content of an index file.

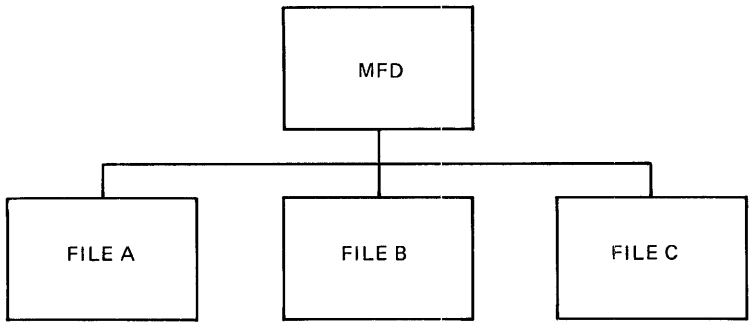


Figure 5-1 Directory Structure for Single-User Volumes

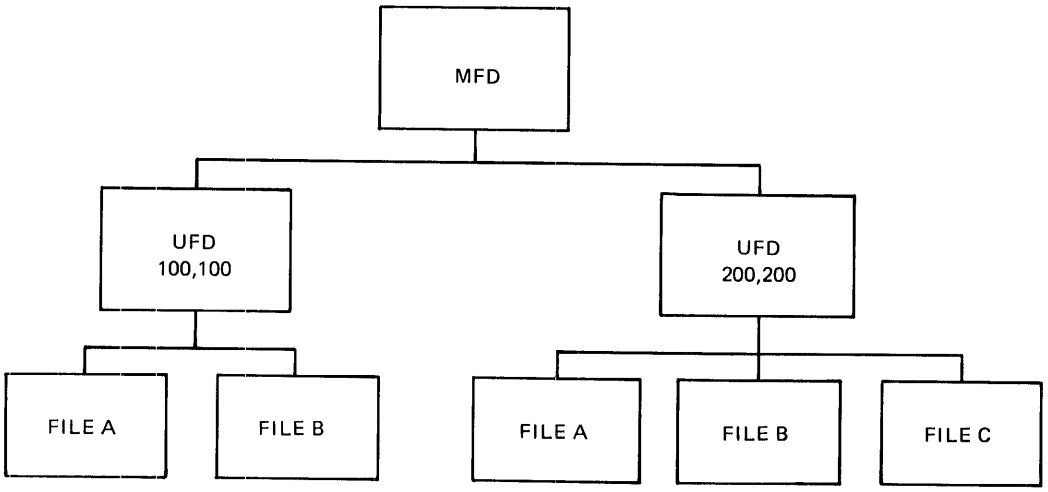


Figure 5-2 Directory Structure for Multiuser Volumes

5.1.4 File-Header Block

Associated with each file is a file-header block that contains information describing the file. File-header blocks are stored in the index file. Each file-header block is 256 words in length and contains three areas: the header area, the identification area, and the map area.

The header area identifies the block as a file-header block. Each file is uniquely identified by a file ID consisting of 2 words. The first word of the file ID, i.e., the file number, is used to calculate the virtual-block number (VBN) of that file's header block in the index file. (This calculation is done as follows: VBN = the file number + 2 + the number of index-file bit-map blocks.) The second word, i.e., the file sequence number, is used to verify that the header block is in fact the header for the desired file.

When a request to access a file is issued, both the file number and the file sequence number are specified. The access request is denied if the file sequence number does not match the corresponding field in the file-header block associated with the specified file number.

When a file is deleted, its file-header block is made available for the subsequent creation of a new file, and when the new file is created, a different file sequence number is stored in the file-header block. If a user attmpts to access a file that has been deleted (e.g., by referencing an obsolete directory entry), this updated file sequence number ensures the failure of the access request, even if the same file-header block is reused for a different file.

The identification area specifies the creation name of the file and identifies the file owner's UIC. This area also specifies the creation date and time, the revision number, the date and time of the last revision (i.e., the time and date on which the last modification to the file occurred), and the expiration date.

The map area provides the information needed by the system to map virtual-block numbers to logical-block numbers.

A checksum value is computed each time the file-header block is read from or written to the volume, thus ensuring that the file-header block 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

IAS and RSX-11 support the standard ANSI magnetic tape structure as described in the June 19, 1974 proposed revision to "Magnetic Tape Labels and File Structure for Information Interchange," ANSI X3.27-1969. Any of the following file/volume combinations can be used:

1. Single file on a single volume,
2. Single file on more than one volume,
3. Multiple files on a single volume,
4. Multiple files on more than one volume.

Items 2 and 4 above constitute a volume set.

The record format on magtape is different from that on disk. When a file that contains variable-length records or fixed-length records that cross block boundaries is copied to magtape, it occupies more blocks on the magtape than it did on the disk. This is so because on magtape the record counts are larger than on disk, and there is unused space at the end of the blocks. In addition, the cannot-cross-block-boundaries bit is set in the file's FDB.

The sequence in which volume and file labels are used and the format of each label type is defined in Appendix G.

NOTE

There is no place for the creation time or the length of the file in an ANSI file-header label. Consequently, the creation time of a file on ANSI magtape is listed as 0. If a contiguous file is copied to ANSI magtape and then transferred back to disk, the resulting disk file is not marked contiguous even if the /CO switch is used because the system cannot know how much space to allocate for the output file when it reads from magtape.

5.2.1 Access to Magnetic Tape Volumes

Magnetic tape is a sequential-access, single-directory storage medium. Only one user can have access to a given volume set at a time. No more than one file in a volume set can be open at a time. Access protection is performed on a volume-set basis. On volumes produced by DIGITAL systems, user access rights are determined by the contents of the owner identification field as described in Section G.1.1.1. Volumes produced by nonDIGITAL systems are restricted to read-only access unless explicitly overridden at MOUNT time.

5.2.2 Rewinding Volume Sets

A magnetic tape volume set can be rewound either by using the FDOP$R macro call before an OPEN$ or CLOSE$ or by using the .CTRL file control subroutine. Regardless of the method used to rewind the volume set, the following procedures are performed by the file control system.

1. All mounted volumes are rewound to BOT;
2. If the first volume in the set is not mounted, the unit to be used is placed offline;
3. If the volume is not already mounted and if the rewind was requested by an OPEN$ macro call or by a .CTRL call, a request to mount the first volume is printed on the operator's console;
4. If the rewind was requested on a CLOSE$ macro call, no mount message is issued until the next volume is needed.

5.2.3 Positioning to the Next File Position

The normal procedure for writing a new file onto a magnetic tape is to begin writing the file following the end of the last file currently in the volume set. However, the FDOP$R macro call can be used to indicate that the new file is to be written immediately after the end-of-file labels of the most recently closed file. This 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 at the logical end of the volume set to create the file.

When the default is used, no check is made for the existence of a file with the same name in the volume set. Therefore, a program written to use magnetic tape normally should specify FA.POS.

The next file-position option is ignored by directory-device file processors. However, programs written mainly for directory devices can specify the next file-position option in open commands for output and, therefore, cause the position to end process to be used automatically when used with ANSI magtape.

5.2.4 Single-File Operations

Single-file operations are performed by specifying the rewind option before the open and before the close. Using this approach, scratch tape operations can be performed as follows:

1. Open the first file with rewind specified.
2. Write the data records and close the file with rewind.

3. Open the first file again for input (rewind is optional).
4. Read and process the data.
5. Close the file with rewind.
6. Open the second file with rewind specified.
7. Write the data records.
8. Close the file with rewind and perform any additional processing.

5.2.5 Multiple-File Operations

A multiple-file volume is created by opening, writing, and then closing a series of files without specifying a rewind. The sequential processing of files on the volume can be accomplished by closing without rewind and then opening the next file without rewind.

Opening a file for extend (OPEN$A) is legal only for the last file on the volume set.

The following tape operations are performed to create a multiple-file tape volume:

1. Open a file for output with rewind.
2. Write data records and close the file.
3. Open the next file with no rewind.
4. Write the data records and close the file.
5. Repeat for as many files as desired.

Files on tape can be opened in a nonsequential order, but increased processing and tape-positioning time is required. Nonsequential access of files in a multivolume set is not recommended.

5.2.6 Using .CTRL

The .CTRL file control routine can be called to override normal FCS defaults for magnetic tape. Examples of its uses are:

1. Continue processing a file on the next volume of a volume set before the end of the current volume is reached.
2. Position to the logical end-of-volume set.
3. Rewind a volume at other than file open or close.
4. Space forward or backward n records.
5. Rewind a file.

When .CTRL is used to continue processing a file on the next volume, the first file section on the next volume is opened. File sections occur when a file is written on more than one volume. The portion of the file on each of the volumes constitutes a file section. For input files, the following .CTRL processing occurs.

1. If the current volume is the last volume in the set, i.e., there is no next volume, end-of-file is reported to the user.

2. If another file section exists, the current volume is rewound and the next volume is mounted. A request to the operator is printed if necessary.

3. The header label (HDR1) of the 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 processing occurs.

1. The current file section is closed with EOV1 and EOV2 labels and the volume is rewound.

2. The next volume is mounted.

3. A file with the same name and the next higher section number is opened for write. The file-set identifier is identical with the volume identifier of the first volume in the volume set.

NOTE

I/O buffers that are currently in memory are written on the next file section.

When .CTRL is used to position to the logical end-of-volume set, the file system positions between the two tape marks at the logical end of the last volume in the set.

When .CTRL is used to space forward or backward across blocks on magnetic tape, spacing crosses volumes for multivolume files.

5.2.7 Examples of Magnetic Tape Processing

The following pages contain examples of FCS statements used to process magnetic tape. Macro parameters not related to magnetic tape handling have been omitted from the statements so that the user need consider only those matters directly related to magnetic tape.

5.2.7.1 Examples of OPEN$W to Create a New File - All routines expect R0 to contain the FDB address.

```
OPRWDO:
;
; OPEN WITH REWIND
;
        FDOP$R  R0,,,,#FA.ENB!FA.RWD    ;SET REWIND AND ENABLE USE
        BR      OPNOUT                  ;OF F.ACTL
OPNXTO:
;
; OPEN FOR NEXT FILE POSITION
```

```
;
        FDOP$R  R0,,,,#FA.ENB!FA.POS    ;SET POSITION TO NEXT
        BR      OPNOUT                  ;AND ENABLE USE OF F.ACTL
OPROYK:
;
; OPEN FILE AT END OF VOLUME KEEPING CURRENT USER
; ACCESS CONTROL BITS
;
        BIC     #FA.ENB,F.ACTL(R0)      ;DISABLE USE OF F.ACTL
        BR      OPNOUT
OPROVO:
;
; OPEN FILE AT END OF VOLUME - SELECT SYSTEM DEFAULT FOR
; USER ACCESS CONTROL BITS
        FDOP$R  R0,,,,#0                ;DISABLE USE OF AND RESET
        BR      OPNOUT                  ;F.ACTL TO ZERO
;
; OPEN FILE WITH CURRENT USER ACCESS CONTROL
;
OPOURO:
        BIS     #FA.ENB,F.ACTL(R0)      ;ENABLE USE OF F.ACTL
OPNOUT: FDBF$R  R0,,#2048.              ;OVERRIDE BLOCK SIZE FOR TAPE
        OPEN$W  R0
        RETURN
```

5.2.7.2 **Examples of OPEN$R 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$** - All routines expect R0 to contain the FDB address.

```
CLSCUR:
;
; CLOSE LEAVING TAPE AT CURRENT POSITION AND KEEPING
; USER ACCESS CONTROL BITS
;
        BIC     #FA.ENB,F.ACTL(R0)      ;DISABLE USE OF F.ACTL
        BR      CLOSE                   ;DEFAULT IS LEAVING AT CURRENT
                                        ;POSITION
```

```
CLSRWD:
;
; CLOSE REWINDING THE VOLUME
;
        FDOP$R  R0,,,,#FA.ENB!FA.RWD    ;SET REWIND AND ENABLE USE OF
        BR      CLOSE                   ;F.ACTL
;
; CLOSE WITH USER ACCESS CONTROL BITS
;
CLSDFL: BIS     #FA.ENB,F.ACTL(R0)      ;ENABLE USE OF F.ACTL
CLOSE:  CLOSE$  R0
        RETURN
```

5.2.7.4 **Combined Examples of OPEN$ and CLOSE$ for Magnetic Tape** - The following examples call routines in previous examples. By combining various magnetic tape operations, the user can process tape volumes in the following ways.

```
;
; SCRATCH TAPE OPERATIONS--SINGLE FILE VOLUME--
;
SCROUT: MOV     #FDBOUT,R0              ;SELECT FDB AND OPEN
        CALL    OPRWDO                  ;OUTPUT FILE WITH REWIND
        RETURN
SCRIN:  MOV     #FDBIN,R0               ;SELECT FDB AND OPEN FOR
        CALL    OPRWDI                  ;INPUT WITH REWIND
        RETURN
CLSCRO: MOV     #FDBOUT,R0              ;CLOSE SCRATCH FILE
        BR      CLSVOL                  ;REWINDING VOLUME
CLSCRI: MOV     FDBIN,R0
CLSVOL: CALL    CLSRWD
        RETURN
;
; MULTI-FILE VOLUME OPERATIONS
;
OPNXTI:
;
; OPEN FILE FOR READING WHEN FILE IS NEXT OR FURTHER UP THE VOLUME
;
        MOV     #FDBIN,R0               ;SELECT FDB
        CALL    OPCURI                  ;OPEN FILE
        RETURN
OPENIN:
;
; OPEN FILE FOR READING WHEN POSITIONED PAST IT
;
        MOV     #FDBIN,R0               ;SELECT FDB
        CALL    OPRWDI
        RETURN
;
; MULTI-FILE OUTPUT OPERATIONS
;
OPNINT:
;
; START NEW VOLUME DESTROYING ALL PAST FILES ON IT
;
        MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        CALL    OPRWDO                  ;OPEN WITH REWIND
        RETURN
```

```
OPNEXT:
;
; OPEN OUTPUT FILE AT NEXT FILE POSITION DESTROYING ANY FILE
; THAT MAY BE AT OR PAST THAT POSITION
;
        MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        CALL    OPNXTO
        RETURN

OPENDT:
;
; OPEN OUTPUT FILE AT CURRENT END OF VOLUME SET KEEPING USER
; ACCESS CONTROL BITS
;
        MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        CALL    OPROVK
        RETURN

OPNEOV:
;
; OPEN OUTPUT FILE AT CURRENT END OF VOLUME AND MAKE THAT THE USER
; ACCESS CONTROL
;
        MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        CALL    OPROVO
        RETURN
;
; NOT LAST FILE IN FILE SET CLOSE ROUTINE
;
CLSFLO: MOV     #FDBOUT,R0              ;SELECT OUTPUT FDB
        BR      CLSXX
CLSFLI: MOV     #FDBIN,R0               ;SELECT INPUT FDB
CLSXX:  CALL    CLSCUR
        RETURN
;
; TO APPEND TO LAST FILE
;
        OPEN$A  #FDBOUT
```

CHAPTER 6

COMMAND-LINE PROCESSING

As noted in Section 2.4.3, a collection of routines available from the system object library ([1,1]SYSLIB.OLB) may be linked with the user program to provide all the logical capabilites required to process command lines dynamically. These system facilities include the following routines:

1. Get Command Line (GCML). This routine accomplishes all the logical functions associated with the entry of command lines from a terminal, an indirect command file, or an on-line storage medium. Using GCML relieves the user of the burden of manually coding command-line-input operations.

2. Command String Interpreter (CSI). Normally, this routine takes command lines from the GCML command-line-input buffer and parses them into the appropriate dataset descriptors required by FCS for opening files.

This body of routines is linked with the user program at task-build time. GCML and CSI are often jointly incorporated in system or application programs as a standardized interface for obtaining and interpreting dynamic command-line input. The flow of data during command-line processing is shown in Figure 6-1.

Although these routines are presented in the context of being used together for processing command-line input, each may be used independently of the other. Doing so, however, means that the user must manually code the functions otherwise performed by the missing component. The joint use of these routines is assumed throughout this chapter to be the "normal" situation.

The invocation of GCML and CSI functions requires that certain initialization operations be accomplished at assembly time. This initialization sets up the GCML command-line-input buffer, defines and initializes control blocks for both GCML and CSI, and establishes the necessary working storage and communication areas for these routines. Also, the appropriate macro calls that invoke GCML and CSI execution-time functions must be included in the source coding at desired logical points to effect the dynamic processing of command lines.

GCML and CSI macro calls observe the same register conventions applicable to FCS. All registers, except R0, are preserved exactly as in all FCS macro calls. R0 is used to contain the address of the GCML control block or the CSI control block, as appropriate.

As with all FCS macro calls, the GCML and CSI macro calls must also be listed as an argument in an .MCALL directive (see Section 2.1) before being issued in the user program.

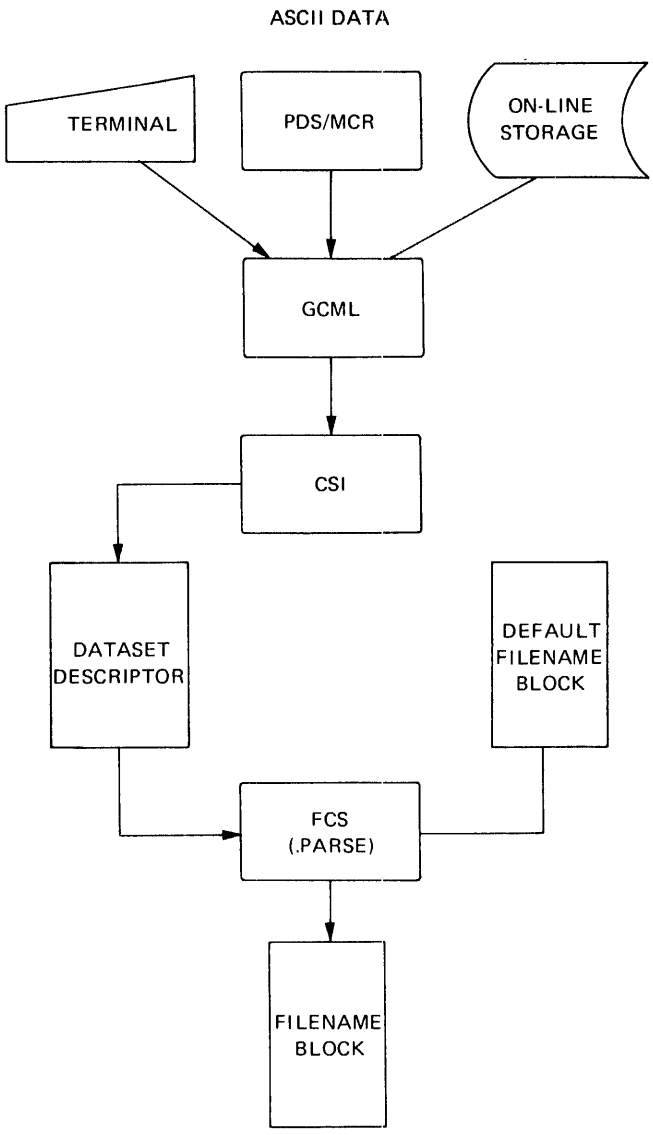


Figure 6-1 Data Flow During Command Line Processing

6.1 GET COMMAND LINE (GCML)

The Get Command Line (GCML) routine embodies all the logical capabilities required 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 the user 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 continuation of commands from one line to the next is effected when a hyphen appears as the last printing character of a command line.

All GCML functions require the creation and initialization of a GCML control block. The macro call that accomplishes this function is described in detail in the following section. The GCML run-time macro calls that may be issued dynamically are described in Section 6.1.3.

6.1.1 GCMLB$ - Allocate and Initialize GCML Control Block

Issuing the GCMLB$ macro call accomplishes the following assembly-time functions:

1. Reserves storage for, and initializes a GCML control block within, the user program.

2. Creates and initializes an FDB in the first part of the GCML control block. This FDB is used to open a command file. Such a file, which may employ a terminal or a file-structured device such as a disk, is opened and read by the user program in the same manner as any other file. The initialization and maintenance of this FDB, however, is under GCML and FCS control and need not be of concern to the user.

3. Creates and initializes a default filename block within the GCML control block. This default filename block pertains to an indirect command file. If an explicit filename string is not specified by the user for an indirect command file, the values CMI for the filename and .CMD for the file type are assumed by default. There is no default designation for the device name.

4. Defines the symbolic offsets for the GCML control block and initializes certain offsets to required values. These offsets are described in detail in Section 6.1.2.

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

```
label:  GCMLB$  maxd,prmpt,ubuf,lun,pdl,size
```

where:

label — represents a symbol that names the GCML control block and defines its address. This label permits the GCML control block to be referenced directly by all the GCML run-time routines that require access to this structure (see Section 6.1.3).

maxd — represents a numeric value that specifies the maximum nesting depth permitted for indirect command files. This parameter determines the number of nested indirect command files that GCML will be allowed to access in obtaining command-line input.

An indirect command file, which often resides on disk, contains well-defined, nonvarying command sequences, which may be read directly by GCML to control operations that are highly repetitive (such as Task Builder activities). Significant advantages in terms of convenience and faster execution result from the use of an indirect command file.

If this parameter is not specified, a nesting level depth of 0 is defined by default, effectively eliminating an indirect command file as a source of command-line input.

prmpt — represents a user-specified, 3-character ASCII prompting sequence. This parameter constitutes a default prompt string that is typed out by GCML to the user terminal to solicit command line input.

The ASCII prompting sequence is formulated into the following 6-byte string:

1. A carriage return (<CR>) and a line-feed (<LF>);

2. The 3 ASCII characters specified by the user; and

3. A right angle bracket (>).

The above string initializes GCML control block offset location G.DPRM (see Section 6.1.2).

If this parameter is not specified, the right angle bracket (>), preceded by three blanks, is defined by default for use by GCML as the default prompting sequence.

ubuf — represents the address of a buffer to be used by GCML for temporary storage of command-line input. If this parameter is not specified, a buffer, whose length is determined by the size parameter, is reserved in the GCML control block for command-line input. If neither this parameter nor the size parameter is specified, a 41-word buffer is reserved by default in the GCML control block.

lun — represents a logical unit number. The device assigned to this logical unit number is used by GCML as the command-input device. If this parameter is not specified, a logical unit number of 1 is used by default.

pdl — represents the address of an area reserved in the user program for use as a push-down list. This area is reserved as working storage for use in connection with indirect command files.

Normally, the pdl parameter is not specified; in this case, sufficient storage for the push-down list is added to the control block by default in accordance with the algorithm described below.

The push-down list is created through statements logically equivalent to the following:

```
        .EVEN
label:  .BLKB   G.LPDL
```

The user-supplied label 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. The value of G.LPDL is calculated according to the following algorithm:

1. Add 1 to the maximum nesting level depth declared with the maxd parameter (see above).

2. Multiply the sum of Step 1 by 16(10), the appropriate number of bytes that must be reserved for the push-down list.

For example, if the maxd parameter is specified as 4, the length of the push-down list is determined as follows:

(4+1)*16. = 80. bytes

From the above, note that 16(10) bytes of storage are required for each indirect command file, plus a total of 16(10) bytes for use as general overhead.

size — represents the size, in bytes, of the buffer reserved for command-line input. The specified size must always include 2 extra bytes that are used internally by GCML. The default value for size is 82 (that is, 80 bytes for command-line input and 2 bytes GCML overhead).

If the user wants GCML to accept continuation lines, the specified value for the size parameter must be greater than 82. When 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.

The following examples represent a GCMLB$ macro call as it might appear in a user program:

```
GCLBLK: GCMLB$  4.,GCM,BUFADR,1.
GCLBLK: GCMLB$  ,,BUFADR
GCLBLK: GCMLB$  DEPTH,GCM,BUFADR,CMILUN,PDLIST,BUFSIZ
```

6.1.2 GCMLD$ - Define GCML Control Block Offsets and Bit Values

THE GCMLD$ macro, which is invoked automatically by the GCMLB$ macro call, locally defines the GCML control block offsets and bit values within the current module. These offsets and associated bit values are listed and described below.

OFFSET NAME — FUNCTIONAL SIGNIFICANCE

G.ERR — Error Return Code Byte. This field initially contains 0. If any one of the error conditions recognized by GCML occurs during the processing of a command line, an appropriate error code is returned to offset location G.ERR in the control block. These error codes are described below:

GE.IOR[1] - Indicates that an I/O error occurred during the input of a command line.

---

[1] 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.

GE.OPR[1] - Indicates that GCML was unable to open or reopen the specified command file.

GE.BIF - Indicates that a syntax error was detected in the name of the indirect command file.

GE.MDE - Indicates that an attempt was made to exceed the maximum permissible nesting-level depth for an indirect command file (see the maxd parameter in the GCMLB$ macro call above).

GE.RBG - Indicates that the 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 - Indicates that the end-of-file (EOF) on the first (unnested) command file was detected.

This code is set in connection with command-file input. When the first call is issued for input, GCML attempts to retrieve an MCR/PDS command line. The first line obtained, whether it is an MCR/PDS command or a terminal command, is accomplished at command level 0. If the name of an indirect command file is then entered, the command-input level is incremented to 1. Each indirect filename entry thereafter increments the command-input level. When the end-of-file (EOF) is encountered on any given indirect file, the command input level is decremented by 1, restoring the count to the previous level and re-opening the associated command file. The next command line from that file is then read.

If an MCR/PDS command has already been read at level 0, entering another MCR/PDS command when level 0 is again reached causes the error code GE.EOF to be returned to offset location F.ERR of the GCML control block. Hence, only one MCR/PDS command line can be read at level 0. If input thus fails at MCR/PDS level 0, then GCML continues to prompt for input until CTRL/Z is typed by the user to indicate terminal end-of-file (EOF).

In summary, the first line of input is always read at level 0. This initial input may be an MCR/PDS command; if the MCR/PDS command fails or is null, the command-input file (normally a terminal) is then opened at level 0. Multiple inputs at level 0 are permissible only in the latter case, i.e., from the command-input file.

G.MODE — Status and Mode Control Byte. This field is initialized at assembly-time with bit definitions to specify certain default actions for GCML during the retrieval of a command line. At runtime, the

[1] 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.

user can reset default status and mode control bits, if desired, by issuing a Bit Clear Byte (BICB) instruction that takes as the source operand the symbolic name of the bit to be cleared. In the case of the GE.LC value (see below), the BISB instruction can be used to override the default action.

The symbolic names of the bits defined in the status and mode control byte are as follows:

GE.IND - (Default) Indicates that a command line containing a leading at sign (@) is to be treated as an explicit indirect-command-file specifier. If, for any reason, the user resets this bit to zero (0), a command line containing a leading at sign (@) is returned to the calling program.

GE.CLO - (Default) Indicates that the command file currently being read is to be closed after each issuance of the GCML$ macro call. If the user resets this bit to 0 for any reason, GCML keeps the current command file open between calls for input. In this case, the FSR (see Section 2.6.1) must include one additional 512(10)-byte buffer for command-line input. This requirement 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 effectively renders 512(10) bytes of FSR-block-buffer space unavailable for other purposes, since the command file remains open between calls for command-line input.

GE.COM - (Default) Indicates that a command line having a leading semicolon (;) is to be treated as a comment. Such lines are not returned to the calling program. If, for any reason, the user resets this bit to 0, a command line containing a leading semicolon (;) is returned to the calling program.

GE.CON - Indicates that the continuation mechanism is in effect. This is the default if the value of the size parameter of the GCMLB$ macro is greater than 82. The user must not attempt to set this value in the mode byte without providing a buffer larger than 82 bytes.

GE.LC - Indicates that lower-case characters in the command line are to be passed to the user program without mapping. Unless the user explicitly sets this value in the GCML control block at runtime, the default action will be to map lower-upper case characters to upper-case before transmission to the user program.

G.PSDS — Prompt-String Descriptor. This 2-word field is initialized to 0 at assembly-time through the GCMLB$ macro call (see Section 6.1.1).

When the GCML$ macro call is issued to request command-line input (see Section 6.1.3.1), the address and the length of a prompting sequence is usually not specified. In this case, the prompt-string descriptor words in the GCML control block are cleared, causing GCML to type out the default prompt string contained in offset location G.DPRM (see below) to solicit command-line input.

The user who wishes to define an alternate prompt string elsewhere in the program may do so through the .ASCII directive. The address and length of this alternate prompt string may then be specified as the adpr and lnpr parameters in subsequent GCML$ macro calls. These parameters cause offset locations G.PSDS+2 and G.PSDS to be initialized with the address and the length, respectively, of the alternate prompt string. The alternate prompt string is then typed out by GCML to solicit command-line input, thereby overriding the default prompt string previously established through the GCMLB$ macro call (see G.DPRM below).

If the adpr and lnpr parameters are not specified in a subsequent GCML$ macro call, offset location G.PSDS in the control block is automatically reset to 0, causing GCML to revert to the use of the default prompt string contained in offset location G.DPRM.

G.CMLD — Command-Line Descriptor. This 2-word field is initialized by GCML after retrieving a command line. The address of the line just obtained is returned to offset location G.CMLD+2, and the length (in bytes) of the command line is returned to offset location G.CMLD.

The contents of these word locations in the GCML control block may be passed to CSI as the "buff" and "len" parameters in the CSI$1 macro call (see Section 6.2.3.1). The combination of these parameters constitutes the command-line descriptors that enable CSI to retrieve file specifiers from the GCML command-line input buffer.

G.ISIZ — Impure Area Size Indicator. This symbol is defined at assembly-time, indicating the size of an impure area within the GCML control block to be used as working storage for pointers, flags, counters, etc., in connection with input from an indirect command file. In normal usage, this symbol need not be of concern to the user.

The space between the FDB and the default prompt string (see G.DPRM below) constitutes the impure area of the GCML control block. The size of the FDB is defined by the value of the symbol S.FDB. Thus, the size of the impure area is equal to G.DPRM-S.FDB.

G.DPRM — Default Prompt String. This 6-byte field is initialized at assembly-time with the default prompt string created through the prmpt parameter

of the GCMLB$ macro call (see Section 6.1.1). In the absence of the adpr and lnpr parameters in the GCML$ macro call (see Section 6.1.3.1), this default prompt string is typed out by GCML to solicit terminal input.

The user who wants to reference the GCML control-block offsets and bit vaues in another module may establish the appropriate symbolic definitions within that module through one of the following statements, as desired:

```
GCMLD$                  ;DEFAULT LOCAL DEFINITION.

GCMLD$  DEF$L           ;LOCAL DEFINITION.

GCMLD$  DEF$G           ;GLOBAL DEFINITION.
```

6.1.3 GCML Run-Time Macro Calls

Three run-time macro calls are provided in GCML to perform specific functions, as described below:

- GCML$ - To retrieve a command line.
- RCML$ - To reset the indirect-command-file scan to the first (unnested) level.
- CCML$ - To close the current command file.

These routines are described separately in the following sections.

6.1.3.1 **GCML$ - Get Command Line** - The GCML$ macro call serves as the user program interface for retrieving command lines from a terminal or an indirect command file. This macro call can be issued at any logical point in the program to solicit command-line input.

This macro call takes the following format:

```
GCML$   gclblk,adpr,inpr
```

where:

- gclblk — represents the address of the GCML control block. This symbol must be the same as that specified at assembly-time in the label field of the GCMLB$ macro call (see Section 6.1.1). If this parameter is not specified, R0 is assumed to contain the address of the GCML control block.
- adpr — represents the address of the user program location containing an alternate prompt string. When this optional parameter and the inpr parameter below are present in the GCML$ macro call, the alternate prompt string is typed out on the user terminal to solicit command-line input. The normal default prompt string, as contained in offset location G.DPRM of the GCML control block (see Section 6.1.2), is thereby overridden.
- lnpr — represents the length (in bytes) of the alternate prompt string. This parameter is also optional; if not specified, offset location G.PSDS in the GCML control block (see Section 6.1.2) is cleared.

If this parameter is specified, but the adpr parameter above is not, an .ERROR directive is generated during assembly 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 this parameter is not given but the adpr parameter above is given, the default prompt string is used.

If the adpr and lnpr parameters are not specified in a subsequent GCML$ macro call, offset location G.PSDS in the GCML control block is automatically reset to 0, causing GCML to revert to the use of the default prompt string contained in offset location G.DPRM (see Section 6.1.2 above).

When the GCML$ macro call is issued, the following actions occur:

1. R0 is loaded with the address of the GCML control block. If the gclblk parameter is not specified, as described above, R0 is assumed to contain the address of the GCML control block. If it does not, R0 must first be initialized manually with the address of the control block before the GCML$ macro call is issued.

2. The address and the length of the alternate prompt string, if specified, are stored in control-block offset locations G.PSDS+2 and G.PSDS, respectively. These 2 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 will automatically transfer commands that run to more than one line. The continuation lines obtained are concatenated in the input buffer with the continuation hyphen(s) removed.

At the initial issuance of the GCML$ macro call, an attempt is made to retrieve an MCR/PDS command line. If this attempt fails, or if the MCR/PDS command line is null, the FDB within the GCML control block is used to open a file for command-line input. If the command-input device is a terminal, a prompt string is typed out to solicit input. Any desired command input may then be entered. If the continuation mechanism is being used, the prompt string is similarly typed to solicit subsequent portions of a continued command line.

If appropriate, the user may enter an at sign (@) as the first character in the command line, followed by the name of an indirect command file. This filename identifies an explicit indirect command file from which input is to be read. GCML then opens this file and retrieves the first command line therein. On successive GCML calls, this file is read until one of the following occurs:

1. The end-of-file (EOF) is detected on the current indirect file. In this case, the current indirect file is closed, the command-input-level count is decremented by 1, and the previous command file is re-opened. 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).

2. An indirect-file specifier is encountered in a command line. In this case, the current indirect command file is closed (if not already closed), and the new indirect file is opened. The first command line therein is then read.

3. An RCML$ macro call is issued in the program (see Section 6.1.3.2 below). In this case, the current indirect command file is closed, and the command-input count reverts to level 0, i.e., the top level command file is again used for input.

The user may also enter a semicolon (;) as the first character in the command line. 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 thus obtained are returned to GCML control-block offset locations G.CMLD+2 and G.CMLD, respectively. Together, these 2 words constitute the command-line descriptors. These descriptors may be specified as the "buff" and "len" parameters in the CSI$1 macro call (see Section 6.2.3.1).

Successful retrieval of a command line causes the C-bit in the Processor Status Word to be cleared. Any error condition that occurs during the retrieval of a command line, however, causes the C-bit to be set. In addition, a negative error code is returned to offset location G.ERR of the GCML control block. These error codes are described in detail in Section 6.1.2 above.

Examples of the GCML$ macro call follow:

```
GCML$   #GCLBLK

GCML$

GCML$   #GCLBLK,#ADPR,#LNPR
```

The first example specifies the symbolic address of the GCML control block. The second example assumes that R0 contains the address of the GCML control block. Both these forms of the GCML$ macro call employ the default prompt string contained in offset location G.DPRM of the control block to solicit command-line input. The last example specifies the address and the length of an alternate prompt string that the user has defined within the program. This alternate prompt string is used by GCML to prompt for terminal input, rather than the default prompt string contained in the GCML control block.

6.1.3.2 **RCML$ - Reset Indirect Command File Scan** - If the user must close the current indirect command file and return to the top-level file, i.e., to the first (unnested) file, he or she may do so by issuing the RCML$ macro call.

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

```
RCML$   gclblk
```

where: gclblk represents the address of the GCML control block. If this parameter is not specified, R0 is assumed to contain the address of the GCML control block.

When this macro call is issued, the current indirect command file is closed, returning control to the top-level (unnested) file. A subsequent GCML$ macro call then retrieves the next command line from the 0-level command file. Note, however, that a second MCR/PDS command at level 0 cannot be read (see GE.EOF error code in offset location G.ERR of GCML control block, Section 6.1.2).

Examples of the RCML$ macro call follow:

```
RCML$   #GCLBLK

RCML$   R0
```

This macro call requires only the address of the GCML control block.

6.1.3.3 **CCML$ - Close Current Command File** - It is often desirable to close the current command file between calls for input in order to free FSR-block-buffer space for some other use. The command file is closed automatically after the retrieval of a command line, provided that the GE.CLO bit in the status and mode control byte remains appropriately initialized (see Section 6.1.2). This bit is set to 1 at assembly-time. If the user resets 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, since keeping the file open between calls for input reduces total file-access time. However, should it be desirable to close such a file to free FSR-block-buffer space, the user may do so by issuing the CCML$ macro call.

The CCML$ macro call takes the following format:

```
CCML$   gclblk
```

where: gclblk represents the address of the GCML control block. If this parameter is not specified, R0 is assumed to contain the address of the GCML control block.

Issuing this statement closes the current command file, effectively releasing 512(10) bytes of FSR-block-buffer space for some other use between calls for input. If the command file is already closed when the CCML$ macro call is issued, control is merely returned to the calling program. A subsequent GCML$ macro call then causes the command file to be reopened and the next command line in the file to be returned to the calling program.

Examples of this macro call are shown below:

```
CCML$   #GCLBLK

CCML$   R0
```

As in the RCML$ macro call above, this macro call takes a single parameter, viz., the address of the GCML control block.

6.1.4 GCML Usage Considerations

As noted in Section 6.1.1, the GCMLB$ macro call creates an FDB in the first part of the GCML control block. Although this FDB ordinarily need not be manipulated by the user (since it is under GCML and FCS control), the following operations may be performed on this FDB:

1. In an irrecoverable error situation, the user may issue a CLOSE$ macro call (see Section 3.8) in connection with this FDB before issuing the system EXIT$ macro call.

2. The user may test the FD.TTY bit in the device-characteristics byte (offset location F.RCTL) of the FDB to determine if the command line just obtained was retrieved from a terminal.

3. In the event that error code GE.IOR 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), the user may test offset location F.ERR of the associated FDB for more complete error analysis. This cell in the FDB also contains an error code that may be helpful in determining the nature of the error condition.

At task-build time, the Task Builder device-assignment (ASG) directive should be issued to assign the appropriate physical device-unit to the desired logical unit number. For example, to assign the logical unit number (lun parameter) in the GCMLB$ macro call (see Section 6.1.1) to a terminal, the following Task Builder directive should be issued:

```
ASG = TI:1
```

The designation TI: is a pseudo-device name that is redirected to the command-input device. Note that the numeric value following the colon (:) must agree with the numeric value specified as the lun parameter in the GCMLB$ macro call.

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

As discussed in Section 2.6.1 on FSRSZ$, at any given time there must be an FSR block buffer available for each file currently open for record I/O operations. The block-buffer requirements of the command file must be considered when issuing the FSRSZ$ macro (FSRSZ$ must be issued with a nonzero first parameter).

6.2 COMMAND STRING INTERPRETER (CSI)

The Command String Interpreter (CSI) analyzes command lines and parses them into their component device name, directory, and filename strings. The user should be aware that CSI processes command lines in the following formats only:

1. dev:[g,m]outputfilename.type;version/switch

   More than one such file specification can be specified by separating them with commas.

2. dev:[g,m]outputfilename.type;version/switch,...= dev:[g,m] input filename.type;version/switch,...

In addition, CSI maintains a dataset descriptor within the CSI control block (see next section) which may be used by FCS in opening files. The run-time routines that analyze and parse command lines for a calling user program are described in Section 6.2.3.

The use of CSI requires that the CSI control-block offsets and bit values be defined and that a control block be allocated within the program. The macro described in the following section accomplishes these requisite actions.

6.2.1 CSI$ - Define CSI Control-Block Offsets and Bit Values

The only initialization coding required for CSI at assembly-time is that shown below:

```
        CSI$                    ;DEFINES CSI CONTROL BLOCK OFFSETS
                                ;AND BIT VALUES LOCALLY.
        .EVEN                   ;WORD ALIGNS CSI CONTROL BLOCK.
CSIBLK: .BLKB   C.SIZE          ;NAMES CSI CONTROL BLOCK AND
          .                     ;ALLOCATES REQUIRED STORAGE.
          .
          .
```

The CSI$ macro is strictly definitional in nature and does not generate any executable code. The CSI control block resulting from the .BLKB directive serves as a means of communication between CSI and the calling program. The length of the control block is specified by the symbol C.SIZE, which is defined during the expansion of the CSI$ macro. The expansion of this macro also results in the local definition of the symbolic offsets and bit values within the CSI control block.

If desired, the user may cause the control-block offsets to be defined globally within the current module. This is done by specifying DEF$G as an argument in the CSI$ initialization macro call, as shown below:

```
        CSI$    DEF$G
```

6.2.2 CSI Control-Block Offset and Bit-Value Definitions

The CSI$ macro call causes the following symbolic offsets and bit values within the CSI control block to be defined locally:

OFFSET NAME	FUNCTIONAL SIGNIFICANCE
C.TYPR	Command String Request Type. This byte field indicates the type of file specifier being requested. Depending on whether an input or output file specifier is being requested (see the io parameter in the CSI$2 macro call, Section 6.2.3.2), the corresponding bit in this byte is set. The bit definitions for this byte are as follows: CS.INP - Indicates that an input file specifier is being requested. CS.OUT - Indicates that an output file specifier is being requested.

C.STAT — Command-String Request Status. This byte field reflects the status of the current command-line request. The bits in this field are initialized in accordance with the bit definitions listed below.

CS.EQU - Indicates that an equal sign (=) has been detected in the current command line, signifying that the command line contains both output and input file specifiers. Once set, the value of CS.EQU is preserved during processing by both CSI1 and CSI2.

CS.NMF - Indicates that the current file specifier contains a filename string. Accordingly, control-block offset locations C.FILD+2 and C.FILD (see below) are initialized with the address and the length (in bytes), respectively, of the command-line segment containing the filename string. If no filename string is present, this bit is not set, and the filename string descriptors in the control block are cleared.

CS.DIF - Indicates that the current file specifier contains a directory string. Thus, control-block offset locations C.DIRD+2 and C.DIRD (see below) are initialized with the address and the length (in bytes), respectively, of the command-line segment containing the directory string. If no directory string is present, this bit is not set. In this case, any residual nonzero values in the directory- string descriptor cells that pertain to a previous command-string request of like type (see C.TYPR above) are used by default. Thus, the last directory string encountered in a file specifier is used.

CS.DVF - Indicates that the current file specifier contains a device-name string. Similarly, control-block offset locations C.DEVD+2 and C.DEVD (see below) are initialized with the address and the length (in bytes), respectively, of the device-name string. If no device name string is present, this bit is not set. Again, similar to CS.DIF above, any residual nonzero values in the device-name descriptor cells that pertain to a previous command-string request of like type are used by default. Thus, the last device-name string encountered in a file specifier is used.

CS.WLD - Indicates that the current file specifier contains an asterisk (*), signalling the presence of a wildcard specification.

CS.MOR - Indicates that the current file specifier is terminated by a comma (,). The comma indicates that more file specifiers are to follow. If this bit is not set, it signifies that the end of the input or output file specifiers has been reached.

C.CMLD — Command-Line Descriptor. This 2-word field is initialized with the address and the length (in bytes), respectively, of the compressed command line. In other words, the values returned to

these cells constitute the output of CSI after scanning a file specifier and removing all nonsignificant characters from the string (i.e., nulls, blanks, tabs, and RUBOUT's).

The values contained in these cells are used by CSI as the descriptors of the compressed command line to be parsed (see CSI$2 macro call in Section 6.2.3.2).

C.DSDS — Dataset Descriptor Pointer. This offset defines the address of the 6-word dataset descriptor in the CSI control block. This structure is functionally identical to the manually created dataset descriptor detailed in Section 2.4.1.

This symbol may be used to initialize offset location F.DSPT in the FDB associated with the file to be processed. Thus, FCS is able to retrieve requisite ASCII information from this structure for use in opening files.

Assembly-time initialization of F.DSPT in the associated FDB may be accomplished as follows:

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

where CSIBLK is the address of the CSI control block, and C.DSDS represents the beginning address of the descriptor strings in the CSI control block (see C.DEVD, C.DIRD, and C.FILD below) identifying the requisite ASCII filename information.

Run-time initialization of F.DSPT in the associated FDB may also be accomplished through the dspt parameter of the FDOP$R macro call (see Section 2.2.2) or the generalized OPEN$x macro call (see Section 3.1).

C.DEVD — Device-Name String Descriptor. This 2-word field contains the address (C.DEVD+2) and the length in bytes (C.DEVD) of the most recent device-name string (of like request type) encountered in a file specifier.

C.DIRD — Directory String Descriptor. This 2-word field contains the address (C.DIRD+2) and the length in bytes (C.DIRD) of the most recent directory string (of like request type) encountered in a file specifier.

C.FILD — Filename String Descriptor. This 2-word field contains the address (C.FILD+2) and the length in bytes (C.FILD) of the filename string in the current file specifier.

If an error condition is detected by the command-syntax analyzer during the syntactical analysis of a command line (see Section 6.2.3.1 below), a segment descriptor is returned to this field, defining the address and the length of the command-line segment in error.

C.SWAD — Current Switch-Table Address. This word location contains the address of the switch descriptor table specified in the current CSI$2 macro call (see Section 6.2.3.2).

C.MKW1 — CSI Mask Word 1. This word indicates the particular switch(es) present in the current file specifier after each invocation of the CSI$2 macro call. The switch mask for each of the defined switches encountered in a file specifier between delimiting commas is OR'ed into this location.

The mask for a switch is specified in the CSI$SW macro call (see Section 6.2.4.1). When a switch is encountered in a file specifier for which a defined mask exists, the corresponding bits in C.MKW1 are set. By testing C.MKW1, the user can determine the particular combination of defined switches present in the current file specifier.

C.MKW2 — CSI Mask Word 2. This word provides the user an indication of switch polarity.

When a switch is present in a file specifer and that switch is not negated, the defined mask for that switch is OR'ed into C.MKW2 in the same manner as described above for C.MKW1. Conversely, when a switch is present in a file specifer and that switch is negated, the corresponding bits in C.MKW2 are cleared. Thus, for each switch indicated as being present by C.MKW1, the user can check the polarity of that switch by examining the corresponding bits in C.MKW2.

C.SIZE — Control-Block Size Indicator. This symbol, which is defined during the expansion of the CSI$ macro, represents the size in bytes of the CSI control block.

6.2.3 CSI Run-Time Macro Calls

Two run-time macro calls are provided in CSI to invoke routines 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 call also results in the initialization of certain cells in the CSI control block with the address and the length, respectively, of the validated and compressed command line.

- CSI$2 - Parses a file specifier in the validated and compressed command line into its component device name, directory, and filename strings, and processes any associated switches and accompanying switch values. Also, certain cells in the CSI control block are initialized with the appropriate string descriptors for subsequent use by FCS in opening the specified file.

6.2.3.1 **CSI$1 - Command Syntax Analyzer** - The CSI$1 macro call results in the invocation of a routine called the command syntax analyzer. This routine analyzes a command line (which is normally read into the GCML command-line input buffer) and checks it for syntactic validity. In addition, it compresses the file specifiers in the command line by removing all nonsignificant characters (i.e., nulls, tabs, blanks, and RUBOUTs). Finally, the command syntax analyzer initializes offset locations C.CMLD+2 and C.CMLD in the CSI control block (see Section 6.2.2) with the address and the length (in bytes), respectively, of the validated and compressed command line. Each file specifier in the command line is then parsed into its component device name, directory, and filename strings during successive issuances of the CSI$2 macro call (see next section).

The CSI$1 macro call is issued in the following format:

```
CSI$1   csiblk,buff,len
```

where:

csiblk — represents the address of the CSI control block. If this parameter is not specified, R0 is assumed to contain the address of the CSI control block.

buff — represents the address of a command-line input buffer. This parameter initializes CSI control-block offset location C.CMLD+2, enabling CSI to retrieve the current command line from a command-line input buffer.

If this parameter is not specified, the user must manually initialize CSI control-block offset location C.CMLD+2 with the address of a command-line input buffer before issuing the CSI$1 macro call. This may be accomplished through a statement similar to the following:

```
MOV     GCLBLK+G.CMLD+2,CSIBLK+C.CMLD+2
```

len — represents the length of the command-line input buffer. Similarly, this parameter initializes CSI control-block offset location C.CMLD, thus completing the 2-word descriptor that enables CSI to retrieve the current command line from the input buffer.

As with the "buff" parameter above, if this parameter is not specified, the user must manually initialize CSI control-block offset location C.CMLD with the length of the command-line input buffer before issuing the CSI$1 macro call. This may be accomplished as follows:

```
MOV     GCLBLK+G.CMLD,CSIBLK+C.CMLD
```

The combination of the buff and len parameters above enables CSI to analyze the current command line. Following the analysis of the command line, CSI updates offset location C.CMLD with the length of the validated and compressed command line.

If a syntactic error is detected during the validation of the command line, the C-bit in the Processor Status Word is set, and offset locations C.FILD+2 and C.FILD in the CSI control block (see Section 6.2.2) are set to values that define the address and the length, respectively, of the command-line segment in error.

Examples of the CSI$1 macro call follow:

```
CSI$1     #CSIBLK,#BUFF,#LEN

CSI$1     R0,GCLBLK+G.CMLD+2,GCLBLK+G.CMLD

CSI$1     #CSIBLK
```

The first example shows symbols 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** - The CSI$2 macro call results in the invocation of the command semantic parser. This routine uses the values in CSI control-block offset locations C.CMLD+2 and C.CMLD as the address and the length, respectively, of the command line to be parsed. The referenced line is then parsed into its component device name, directory, and filename strings. The equal sign (=) in the command line indicates that the following string 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 the user program (see Section 2.4.1).

The command semantic parser also decodes any switches and associated switch values present in a file specifier. If the user expects to encounter switches in the current file specifier, the command semantic parser decodes them, provided that the address of the appropriate switch descriptor table has been specified in the CSI$2 macro call (see below). The CSI switch definition macro calls are described in detail in Section 6.2.4.

The CSI$2 macro call is specified in the following format:

```
CSI$2   csiblk,io,swtab
```

where:

csiblk represents the address of the CSI control block. If this parameter is not specified, R0 is assumed to contain the address of the CSI control block.

io represents a symbol that explicitly identifies the type of file specifier to be parsed. Either of two symbolic arguments may be specified in this parameter field, as follows:

INPUT - Indicates that the next input file specifier in the command line is to be parsed.

OUTPUT - Indicates that the next output file specifier in the command line is to be parsed.

Offset location C.TYPR in the CSI control block (see Section 6.2.2) must be initialized, either manually or through the CSI$2 macro call, with the type of file specifier being requested. If other than the symbolic arguments defined above are specified in the CSI$2 macro call, an .ERROR directive is generated during assembly that causes the error message INCORRECT REQUEST TO .CSI2 to be printed in the assembly listing. This diagnostic message alerts the user to the presence of an invalid io parameter in the CSI$2 macro call.

swtab represents the address of the associated switch descriptor table for this particular issuance of the CSI$2 macro call. This optional parameter need be specified only when the user anticipates the presence of a switch in the file specifier that is to be decoded. Specifying this parameter presumes that the user previously created a corresponding switch descriptor table in the program through the CSI$SW macro call (see Section 6.2.4.1). In addition, if the switch to be decoded has any associated switch values, the user must also have created an associated switch value descriptor table in the program through the CSI$SV macro call (see Section 6.2.4.2).

This parameter initializes offset location C.SWAD in the CSI control block (see Section 6.2.2); if not specified, any residual nonzero value in this cell is used by default as the address of the switch descriptor table.

Offset location C.SWAD may also be initialized manually prior to issuing the CSI$2 macro call, as shown in the following statement:

```
MOV     #SWTAB,CSIBLK+C.SWAD
```

where SWTAB is the symbolic address of the associated switch descriptor table.

If an error condition occurs during the parsing of the file specifier, the C-bit in the Processor Status Word is set, and control is returned to the calling program. The possible error conditions that may occur during command-line parsing include the following:

1. The request type was invalid, i.e., offset location C.TYPR in the CSI control block (see Section 6.2.2) was incorrectly initialized.

2. A switch was present in a file specifier, but the address of the switch descriptor table was not specified in the CSI$2 macro call, or the switch descriptor table did not contain a corresponding entry for the switch.

3. An invalid switch value was present in the file specifier.

4. More values accompanied a given switch in the file specifier than there were corresponding entries in the switch value descriptor table for decoding those values.

5. A negative switch was present in the file specifier, but the corresponding entry in the switch descriptor table did not allow the switch to be negated (see the nflag parameter of the CSI$SW macro call in the next section).

Examples of the CSI$2 macro call are shown below:

```
CSI$2   #CSIBLK,INPUT,#SWTBL

CSI$2   R0,OUTPUT,#SWTBL

CSI$2   #CSIBLK,INPUT
```

The first example shows a request to parse an input file specifier, which may include an associated switch. The second example, which assumes that R0 presently contains the address of the CSI control block, parses an output file specifier that likewise may include a switch. The last example is a request to parse an input file specifier and to disallow any accompanying switch(es).

6.2.4 CSI Switch Definition Macro Calls

The following macro calls must be issued at assembly-time to create the requisite switch descriptor tables in the program for processing switches that appear in a file specifier:

CSI$SW - Creates an entry in the switch descriptor table for a particular switch that the user expects to encounter in a file specifier.

CSI$SV - Creates a matching entry in the switch-value descriptor table for the switch defined through the CSI$SW macro call above.

CSI$ND - Terminates a switch descriptor table or a switch-value descriptor table created through the CSI$SW or the CSI$SV macro call, respectively.

These macro calls are described separately in the following sections.

6.2.4.1 **CSI$SW - Create Switch Descriptor Table Entry** - To process each switch that the user expects to encounter in a file specifier, a matching entry in the switch descriptor table must be defined. If no switch descriptor table is specified, the presence of any switch in the command line causes an error to occur. When the address of a switch descriptor table is specified in any particular issuance of the CSI$2 macro call (see Section 6.2.3.2), the following processing occurs:

1. For each switch encountered in a file specifier, CSI searches the switch descriptor table for a matching entry. If the switch descriptor table address is not specified, or a matching entry is not found in the table for the switch, that switch is considered to be invalid. As a result, the C-bit in the Processor Status Word is set, any remaining switches in the file specifier are bypassed, and control is returned to the calling program.

2. If a matching entry is found in the switch descriptor table, mask word 1 in the CSI control block is set according to the defined mask for that switch (see C.MKW1, Section 6.2.2).

3. The negation status of the switch is determined. If the switch is not negated, the corresponding bits in mask word 2 (C.MKW2) in the CSI control block are set according to the defined mask for that switch. If the switch is negated, and negation is not allowed, then the switch is considered to be invalid. In this case, the error sequence described in Step 1 above applies. However, if the switch is negated, and negation is allowed, then the corresponding bits in C.MKW2 are cleared.

   The negation flag for a switch is established through the nflag parameter of the CSI$SW macro call (see below).

4. If the address of the optional user mask word is not present in the corresponding switch descriptor table entry, i.e., if the mkw parameter has not been specified in the associated CSI$SW macro call (see below), switch processing continues with Step 7. If, however, the address of the optional mask word is specified, switch processing continues with Step 5.

5. If SET has been specified as the clear/set flag in the corresponding switch descriptor table entry, and the switch is not negated, then the corresponding bits in the optional mask word are set according to the defined mask for that switch. If, however, the switch is negated, the corresponding bits in the optional mask word are cleared.

   The clear/set flag is specified as the csflg parameter in the CSI$SW macro call (see below).

6. If CLEAR has been specified as the clear/set flag in the corresponding switch descriptor table entry, and the switch is not negated, the corresponding bits in the optional mask word are cleared. Conversely, if the switch is negated, the corresponding bits in the optional mask word are set.

7. If a switch value accompanies a switch in a file specifier, the associated switch-value descriptor table created through the CSI$SV macro call (see next section) is used to decode the value. There must be at least as many entries in the switch-value descriptor table as there are such values accompanying the switch in the file specifier. If the switch-value descriptor table is incomplete, if an invalid switch value is encountered, or if the address of the switch-value descriptor table is not present in the associated switch descriptor table, then the switch is considered to be invalid, and the error sequence described in Step 1 again applies.

   The address of the switch-value descriptor table is specified as the vtab parameter in the CSI$SW macro call (see below).

The CSI$SW macro call is specified in the following format:

```
label:  CSI$SW  sw,mk,mkw,csflg,nflg,vtab,compflg
```

where: label represents an optional symbol that names the resulting switch descriptor table entry and defines its address. In order to establish the address of a switch descriptor table, the first

CSI$SW macro call issued in the program must include a label. This label allows the table to be referenced by other instructions in the program.

sw — represents the alphabetic switch name to be stored in the switch descriptor table. This name may comprise any number of characters. CSI compares the name entered on the command line with this switch name, as entered in the switch descriptor table. The method CSI uses to compare their names is described below. This parameter is required. If omitted, an .ERROR directive is generated during assembly that causes the error message MISSING SWITCH NAME to be printed in the assembly listing.

mk — represents a user-defined mask for the switch specified through the sw parameter above. To enable CSI to indicate the presence of a given switch in a file specifier, a mask value for the switch must be defined, as shown below:

```
ASMSK = 1
NUMSK = 2
   .
   .
   .
VWMSK = 40000
XYMSK = 100000
```

where the (octal) value assigned by the user to each symbol defines a unique bit configuration that is to be set in CSI mask word 1 (C.MKW1) of the control block when a defined switch is encountered in a file specifier.

By specifying the appropriate symbol as the mk parameter in the CSI$SW macro call, the corresponding mask value is stored in the resulting switch descriptor table entry. Thus, a mechanism is established through which the user can determine the particular combination of switches present in a file specifier. For every matching entry found in the switch descriptor table, the corresponding bits are set in C.MKW1.

mkw — represents the address in user 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 the user's program. The SET and CLEAR functions are set using the csflg parameter described below.

Such an optional word may be reserved through a statement logically equivalent to that shown below:

```
MASKX:  .WORD   0
```

csflg — represents a symbolic argument that specifies the clear/set flag for a given switch. This parameter is optional; if not specified, SET is assumed (see below). Either one of two symbolic arguments may be specified for this parameter, as follows:

CLEAR - Indicates that the bits in the optional user mask word corresponding to the switch mask, are to be cleared provided that the switch is not negated. (If the switch is negated, the bits are set.)

SET - Indicates, conversely, that the bits in the optional user mask word corresponding to the switch mask are to be set provided that the switch is not negated. (If the switch is negated, the bits are cleared.)

If other than one of the above arguments is specified, an .ERROR directive is generated during assembly which causes the error message INVALID SET/CLEAR SPEC to be printed in the assembly listing.

nflg — represents a symbolic argument that specifies an optional negation flag for the switch. If this parameter is specified, it indicates that the switch is allowed to be negated, e.g., /-LI or /NOLI.

If this parameter is specified as other than NEG, an .ERROR directive is generated during assembly that causes the error message INVALID NEGATE SPEC to be printed in the assembly listing. If this parameter is not specified, the default assumption is that switch negation is not allowed.

vtab — represents the address of the switch descriptor table associated with this switch. This optional parameter, if specified, allows CSI to decode any switch values accompanying the switch, provided that an associated switch descriptor table entry has been defined for that switch. The switch-value descriptor table is defined through the CSI$SV macro call, as described in the next section. (If the vtab parameter is specified in the CSI$SV macro call, there is no need to specify it in the CSI$SW macro call.)

compflg — specifies the method CSI uses to compare the switch name entered on the command line with the value entered in the switch descriptor table via the sw parameter. Either LONG or EXACT may be specified. The default value is entered if a value is not coded.

Default - If the parameter is not coded, only the first 2 characters of the switch name (specified via sw) are entered into the switch descriptor table and only these 2 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 are treated as an error.

The format of the switch descriptor table entry that results from a call to the CSI$SW macro is shown in Figure 6-2 below. One such switch entry must be defined for each switch appearing in the file specifier that the user intends to recognize. Entries in the switch descriptor table consist of control information and switch-name characters stored 2 characters per word.

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. A sign bit set to 0 indicates 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 is ignored by CSI.

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. Otherwise, they are ignored.

The switch-name characters are followed by entry control information consisting of the CSI mask word, the address in user program storage 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 the user 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.

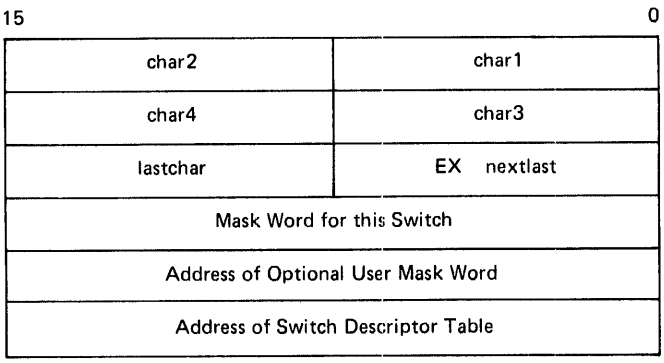


Figure 6-2 Format of Switch Descriptor Table Entry

The following example shows a 2-entry switch descriptor table created through successive CSI$SW macro calls:

```
ASSWT:  CSI$SW  AS,ASMSK,MASKX,SET,,ASVTBL

        CSI$SW  NU,NUMSK,MASKX,CLEAR,NEG,NUVTBL

        CSI$ND                  ;END OF SWITCH DESCRIPTOR TABLE.
```

The first statement results in the creation of an entry in the switch descriptor table for the switch /AS. The second parameter is an equated symbol that defines the switch mask, and the third parameter (MASKX) is the address of an optional user mask word (see the mkw parameter above). The fourth parameter indicates that the bits in MASKX 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 results in the creation of a switch descriptor table entry for the switch /NU. In contrast to the first statement, the fourth parameter (CLEAR) indicates that the bits in the optional user mask word (MASKX) 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 value table associated with this switch.

Note that the switch descriptor macros are terminated with the CSI$ND macro call (see Section 6.2.4.3).

6.2.4.2 **CSI$SV - Create Switch-Value Descriptor Table Entry** - For every switch value that the user expects to encounter in connection with a given switch in a file specifier, a corresponding switch-value descriptor table entry must be defined in the user program in order to allow the switch value(s) to be decoded. The CSI$SV macro call is provided for this purpose. When issued, this macro call results in the creation of a 2-word entry in the switch-value descriptor table. The format of this table is shown in Figure 6-3 below.

The CSI$SV macro call is specified in the following format:

```
CSI$SV  type,adr,len,vtab
```

where:

type represents a symbolic argument that specifies the conversion type for the switch value. Any one of four symbolic values may be specified in this parameter field to indicate the conversion type for the accompanying switch value. The possible conversion-type arguments include the following:

ASCII - Indicates that the switch value is to be treated as an ASCII string.

NUMERIC - Indicates that a numeric switch value is to be converted to binary using octal as a default conversion radix.

OCTAL - Indicates that a numeric switch value is to be converted to binary using octal as a default conversion radix.

DECIMAL - Indicates that a numeric switch value is to be converted to binary using decimal as a default conversion radix.

If any value other than those defined above is specified, an .ERROR directive is generated during assembly that causes the error message INVALID CONVERSION TYPE to be printed in the assembly listing. If none of the above parameters is specified, ASCII is assumed by default.

adr represents the address of the user program location 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 represents a numeric value that defines the length (in bytes) of the area that is to receive the switch value resulting 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.

NOTE

The len parameter is ignored for all types except ASCII.

vtab represents a symbol that names the switch-value descriptor table and defines its address. This parameter is optional. The vtab parameter may also be specified in the CSI$SW macro call (see Section 6.2.4.1) when the user anticipates the presence of a switch value in a file specifier that is to be decoded. (If the vtab parameter is specified in the CSI$SW macro call, there is no need to specifv it in the CSI$SV macro call.)

The format of a switch-value descriptor table entry that results from the CSI$SV macro call is shown in Figure 6-3 below.

The low-order byte of the first word in the switch-value descriptor table indicates whether the conversion type is ASCII or numeric. 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; it 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 simply ignored. If, however, the actual number of ASCII characters present in the switch value falls short of the specified length, the remaining portion of the area receiving the resultant value is null-padded.

If the conversion type is NUMERIC, the resultant binary value is assumed to be 2 bytes in length, and the area receiving the value is assumed to be word-aligned.

On numeric conversions, the default conversion type specified for a switch value can be overridden by means of a pound sign (#) or a dot (.). A numeric value preceded by a pound sign (e.g., #10) forces the conversion type to octal; a numeric value followed by a dot (e.g., 10.) forces the conversion type to decimal. Note also that a numeric switch value may be preceded by a plus sign (+) or a minus sign (-). The plus sign is the default assumption. If an explicit octal switch value is specified using the pound sign (#), as described above, the arithmetic sign indicator (+ or -), if included, must precede the pound sign (e.g., -#10).

If the conversion type is decimal, the switch value is evaluated as a single number; an overflow into the high order bit (bit 15) results in an error condition. However, if the conversion type is octal, a full 16-bit value may be specified.

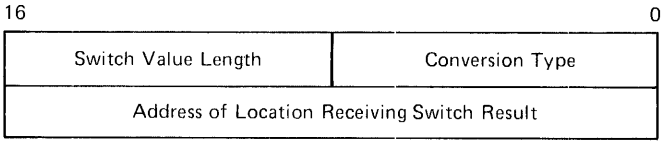


Figure 6-3 Format of Switch-Value Descriptor Table Entry

Representative CSI$SV macro calls are shown below:

```
ASVTBL: CSI$SV  ASCII,ASVAL,3
        CSI$SV  ASCII,ASVAL+4,3
        CSI$ND                  ;END OF SWITCH VALUE TABLE.
NUVTBL: CSI$SV  OCTAL,NUVAL,2
        CSI$SV  DECIMAL,NUVAL+2,2
        CSI$ND                  ;END OF SWITCH VALUE TABLE.
```

In all cases above, the first parameter in the CSI$SV macro call defines the conversion type. The next two parameters, in all cases, define the address and the length of the user program location to receive the resultant switch value.

The required storage for the first switch value table above may be reserved as follows:

```
ASVAL   .BLKW   4               ;ASCII VALUE STORAGE.
```

The required storage for the second switch-value table may be similarly reserved through the following statement:

```
NUVAL:  .BLKW   2               ;NUMERIC VALUE STORAGE.
```

Note again that switch value tables are terminated with the CSI$ND macro call.

6.2.4.3 **CSI$ND - Define End of Descriptor Table** - Switch descriptor tables and switch-value descriptor tables must be terminated with a 1-word end-of-table entry. This word, which contains 0, may be created through the CSI$ND macro call.

This macro call takes no arguments, as shown below:

```
        CSI$ND
```

The examples near the end of the preceding section illustrate the use of this macro call.

CHAPTER 7

THE TABLE-DRIVEN PARSER (TPARS)

TPARS is a table-driven parser designed to parse command lines. TPARS provides the means to define a unique syntax and, using TPARS-supplied macros, built-in variables, and the user's own code, recognize a command line written in that syntax.

For TPARS, parsing is breaking down a command line into syntax elements and resolving the form, function, and interrelationship of these elements. TPARS parses command lines at two levels, a syntactical level and a semantic level.

At the syntactical level, TPARS evaluates each syntax element on the command line based on a predefined arrangement of command line elements. This arrangement of command line elements is defined by TPARS macros in a state table that consists of states and transitions. Also at the syntactic level, TPARS provides for resolving complex syntax types by means of subexpressions.

On the semantic level, TPARS resolves the meaning of each element based on definitions supplied in action routines. Action routines make use of TPARS built-in variables and user-supplied code to fit the needs of user parsing routine.

TPARS parses command lines using a user-defined table consisting of states and transitions. This table is referred to as a state table and is built 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 provides direction for further parsing at another state.

The user-written parser routine is included in user programs that parse command lines. TPARS is invoked from within an executing program by means of a CALL instruction. The CALL invokes the user-defined parser routine as well as the TPARS processor itself. For further information on the interrelationships between 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 syntax. The ISTAT$ macro initializes a state table, the STATE$ macro defines a state in the user's state table, and the TRAN$ macro defines the conditions for transition to another.

7.1.1.1 **Initializing the State Table: the ISTAT$ Macro** - The ISTAT$ macro initializes the state table. The state table is built using two macros, STATE$ and TRAN$, which are described below. This state table is built into a program section. User-defined keyword strings, for use in parsing command lines, are also accumulated in a program section. A program section is also provided for a keyword pointer table to index into the list of keyword strings. The ISTAT$ macro initializes these program sections. The format for coding the ISTAT$ macro is:

```
ISTAT$ statetable,keytable,$DEBUG
```

where:

statetable is the label the user assigns to the state table. TPARS equates this label to the start of the state table.

keytable is the label the user assigns to the keyword table. TPARS equates this label to 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 user-supplied debug routine (see Section 7.1.3.4). When $DEBUG is not included, 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; the keyword pointer table is built in a program section named $KTAB.

If the user defines the symbol $RONLY, each of these program sections is generated as read-only. A read-only state table is generated by specifying the symbol $RONLY preceding the ISTAT$ macro in the form:

```
$RONLY = 1
ISTAT$ statetable,keytable,$DEBUG
     .
     .
     .
```

7.1.1.2 **Defining a Syntax Element: the STATE$ Macro** - The STATE$ macro declares the beginning of a state. Syntactically, this macro delimits one command line element from another. The STATE$ macro is coded in the following form:

```
STATE$ [label]
```

where label is an alphameric symbol that is equated to the address of the state.

Each state is comprised of any number of transitions, which define the conditions under which control can be passed to another state.

7.1.1.3 **Defining a Transition: the $TRAN Macro** - The TRAN$ macro provides:

- The means for matching a given type of syntax element.
- Direction to the next state to be processed.
- The address of the action routine to process the current syntax element.
- A maskword and maskword address.

The TRAN$ macro is coded in the following form:

```
TRAN$ type[,label][,action][,mask][,maskaddr]
          [,$EXIT]
```

where:

type — specifies the syntactical type of the command line element being parsed. The type parameter is coded using one of the types of command line elements described in the section "Types of Command Line Syntax Elements," below.

[label] [$EXIT] — specifies the label associated with the state to which control is directed after the code for this transition is executed. If label is omitted, control drops through to the next sequential STATE$ macro. A null label parameter is allowed only for the last transition in a state; the statement following a TRAN$ macro with a null label field must be a STATE$ macro.

$EXIT specified in the label field terminates TPARS execution and returns control to the calling program. $EXIT is also used to terminate a TPARS subexpression.

action — specifies the label of an action routine the user includes in the parser routine. This routine can include TPARS built-in variables, described in Section 7.1.3, below.

mask — specifies a maskword to be stored into a maskword address whenever the transition is executed. If mask is specified, maskaddr, below, must be specified. This maskword is ORed into maskaddr (described below) when the transition is taken (after the action routine is called).

maskaddr — specifies the label for an address into which TPARS stores the value specified by the mask parameter. The maskaddr parameter must be specified if mask is specified.

The mask and maskaddr parameters provide a convenient means for flagging the execution of a particular transition.

7.1.2 Types of Command Line Syntax Elements

The type parameter of the TRAN$ macro requires the entry of one of the following types of syntax elements:

$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 a number. A number consists of a string of digits, followed optionally by a period. If number is not followed by a period, it is 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 nonnumeric 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.
$STRNG	Matches any alphameric character string (0-9,A-Z). The string will not be null.
$RAD50	Matches any legal RADIX-50 string, that is, any string containing alphameric characters and/or the period (.) and dollar sign ($) characters. Conversion to RADIX-50 format is the responsibility of the action routine.
$BLANK	Matches a string of blank and/or tab characters.
$EOS	Matches the end of the input string. Once TPARS has reached the end of the input string, $EOS matches as many times as it is encountered in the state table.
char	Matches a single character whose ASCII code corresponds to the value of the expression char. The value of the expression must be a 7-bit ASCII code, that is, the value must be in the range 0-177 (octal). Constructions such as 'X are valid and, in fact, recommended.
"keyword"	Matches a specified keyword. Keywords may be any length, may contain only alphameric characters, and are terminated by the first nonalphameric character encountered. The maximum number of keywords allowed in a state table is 64.
!label	Matches the string processed by executing the state table section that starts with the state specified as label. This syntax type is used to pass control to a subexpression. For information on TPARS subexpressions, refer to Section 7.1.4.

7.1.3 Action Routines and Built-in Variables

Action routines provide the means for processing command line elements at the semantic level. That is, a given syntax element can have more than one meaning. Action routines provide a mechanism for determining and validating the meaning of the syntax elements.

Write action routines to perform functions related to the unique requirements of the user's parsing program.

7.1.3.1 **TPARS Built-in Variables** - TPARS provides the following built-in variables for use in 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 via register 1 (R1). Action routines can modify this word to change options dynamically.
.TPDEB	contains the entry address of the optional (user-supplied) debug routine.
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.

7.1.3.2 **Calling Action Routines** - Action routines are called via a JSR PC instruction. Action routines may modify registers R0, R1, and R2; all other registers must be preserved.

7.1.3.3 **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 allows extension of 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.

7.1.3.4 **Optional Debug Routine for RSX-11 Users** - A user-supplied debug routine 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 user-supplied debug routine is to be called by TPARS, the user task must first specify the entry point address for the debug routine in TPARS location .TPDEB, as follows:

```
MOV     #DENTER,.TPDEB
```

Then, invoke TPARS via the .TPARD entry point (rather than via .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.

In order for addresses displayed by the debug routine to be useful, it is necessary to 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).

7.1.4 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. This macro is followed 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 via 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 describing how to arrange syntax types in a state table, rules for entering special characters (commas and angle brackets), and how to direct TPARS to ignore blanks and tab characters in a command line.

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 the types are entered 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 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
```

7.2.3 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 one 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 zero, 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
```

7.2.4 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 (non-alphanumeric) 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 will scan and attempt 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. The following example can be used to achieve the desired operation:

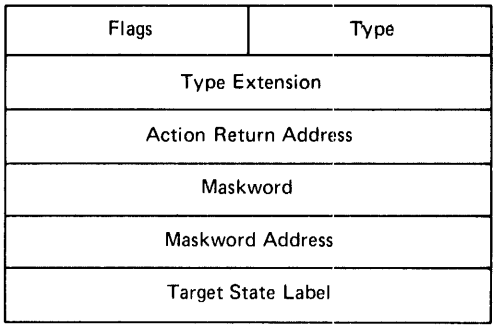
```
STATE$
TRAN$    !NONO,STATE1,SETNEG
TRAN$    $LAMDA,,SETPOS

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 PSECTs GENERATED BY TPARS MACROS

TPARS macros generate three PSECT's. Data associated with the STATE$ macro is stored in the PSECT $STATE. Data associated with the TRAN$ macro is stored in PSECTs $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 they are specified. 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:



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. The keyword strings are stored in the PSECT $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

The user controls execution of TPARS using the calling conventions and options described in this section. TPARS is invoked from within an executing program on an IAS or RSX-11M system by means of the instruction:

```
CALL    .TPARS
```

When a user-supplied debug routine is used for tracing TPARS operation (see Section 7.1.3.4), a special entry point is called, as follows:

```
CALL    .TPARD
```

When TPARS is called in this manner, TPARS calls the debug routine at each state transition. If TPARS is invoked via 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 zero 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 zero, 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

There are three processes involved in generating a parser program using TPARS, as shown in Figure 7-1.

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.

Assembly of the source module produces an object module comprised of three PSECT's, $STATE, $KTAB, and $KSTR. When the Task Builder executes, the task image produced is placed on an appropriate volume for future use.

The assembly listing produced by the state table macros is not straightforward. The source language macros are designed for clarity and convenience in writing state tables; the object code is designed for compactness and processing convenience. As a result, the mechanism used to translate the source code to object code is not a simple one.

The binary output of the macros is delayed by one statement. Thus, if the listing of macro-generated binary code is enabled, 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.

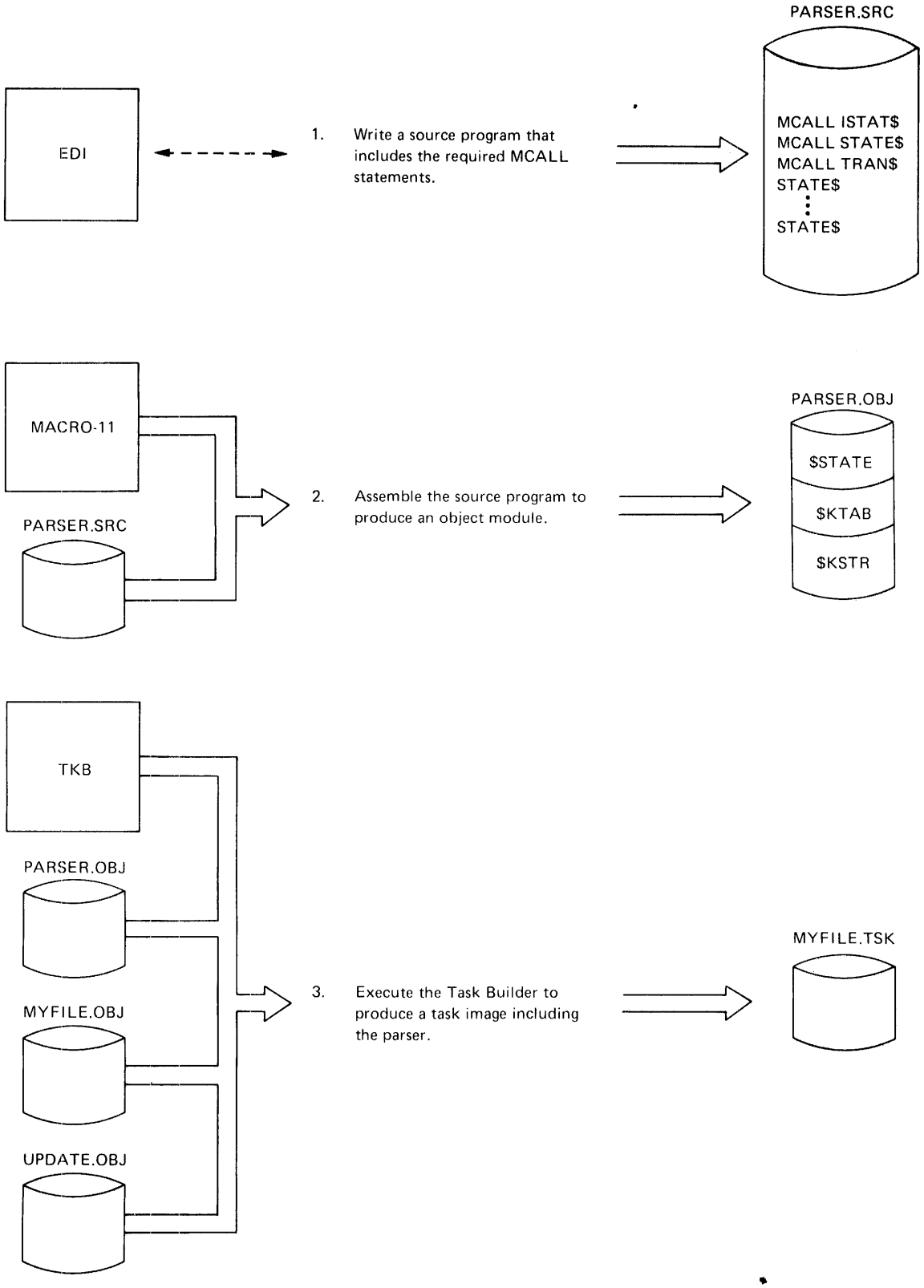


Figure 7-1 Processing Steps Required to Generate a Parser Program Using TPARS

When the parser program is linked and is in task image form, it can be invoked from within an executing user program, as shown in Figure 7-2.

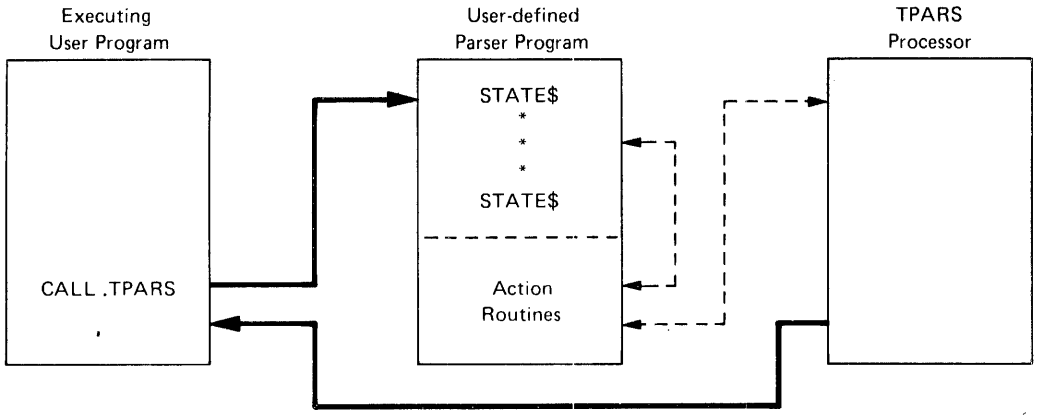


Figure 7-2 Flow of Control When TPARS Is Called from an Executing User Program

Figure 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.

7.6 PROGRAMMING EXAMPLES

This section includes three programmed examples of how to use TPARS. 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 indeterminate grammars.

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 the user-written parser routine:

```
        .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,$UUIC
TSTPPN:           TST     .PNUMH          ; CHECK FOR ZERO HIGH ORDER
                  BNE     10$
                  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 How to Use Subexpressions and Reject Transitions

This example is an excerpt of a state table that parses a string quoted by an arbitrary character. That is, 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

; 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 Grammars

This example is an excerpt from a state table that shows how subexpressions are used to parse complex grammars.

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

; 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 via $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 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.

8.1 PRINT$ MACRO CALL

A task issues the PRINT$ macro call to queue a file for printing on a specified device. The specified device must be a unit-record, carriage-controlled device such as a line printer or terminal. If the device is not specified, LP: is used.

The file to be spooled must be open when the PRINT$ macro is issued. PRINT$ closes the file. Error returns differ from normal FCS conventions and are described in Section 8.3.

The PRINT$ macro call has the following format:

```
PRINT$ fdb,err,,dev,unit,pri,forms,copies,presrv
```

fdb — represents the address of the associated FDB.

err — represents the address of an optional user-coded error handling routine. See Section 8.3.

The following parameters are not applicable to RSX-11M/M-PLUS.

A blank parameter is present between err and dev, thus requiring an additional comma.

dev — represents the 2-character device mnemonic of the device on which the file is to be printed. If dev is not specified, LP: is used by default.

unit — represents the unit number of the device on which the file is to be printed. If unit is not specified, unit 0 is used by default.

pri — represents a number in the range 1 through 250 to indicate the priority of the request. The priority is used to determine the order in which spooled files are dequeued for printing. If pri is omitted, the task's priority is used by default.

forms — represents the specific form-type on which the file is to be printed as indicated by a number in the range 0 through 6. This parameter must be specified as a single integer without a preceding number sign (#). The numbers 0 through 6 are associated with the various forms for an installation by the system manager. If forms is omitted, form-type 0 is used by default.

copies — represents a number in the range 1 through 32(10) to indicate the number of copies of the file to be produced. The number of copies must be specified as a 1- or 2-digit integer without a preceding number sign (#). If copies is omitted, one copy is printed. If copies is specified as a decimal number, a trailing decimal point should be included in the parameter.

presrv — should be specified if the file is not to be deleted after it is printed. Any parameter value results in the file's being preserved after printing.

8.2 .PRINT SUBROUTINE

The .PRINT subroutine is called to queue a file for printing. The file must be open when .PRINT is called. The .PRINT routine closes the file. R0 must contain the address of the associated FDB. One copy of the file is printed on LP:.

Section 8.3 describes error handling for the .PRINT file control routine.

8.3 ERROR HANDLING

The error returns provided in conjunction with PRINT$ and .PRINT differ from the standard FCS error returns in that error codes are placed in F.ERR or in the directive status word depending on when the failure occurred.

If the failure is FCS related, e.g., the PRINT$ macro cannot close the file, the C-bit is set and F.ERR contains the error code. If the failure is related to the SEND/REQUEST directive that queues the file, the C-bit is set and the directive status word contains an error code. Directive status word error codes are provided in the Executive Reference Manual of the host operating system.

Normally, user-coded error routines, upon determining that the C-bit is set, should test F.ERR first and then test the directive status word.

APPENDIX A

FILE DESCRIPTOR BLOCK

A file descriptor block (FDB) contains file information that is used by 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 referring to FDB locations, it is essential to use the symbolic offset names, rather than the actual address of such locations. The position of information within the FDB may be subject to change from release to release, whereas the offset names remain constant.

FILE DESCRIPTOR BLOCK

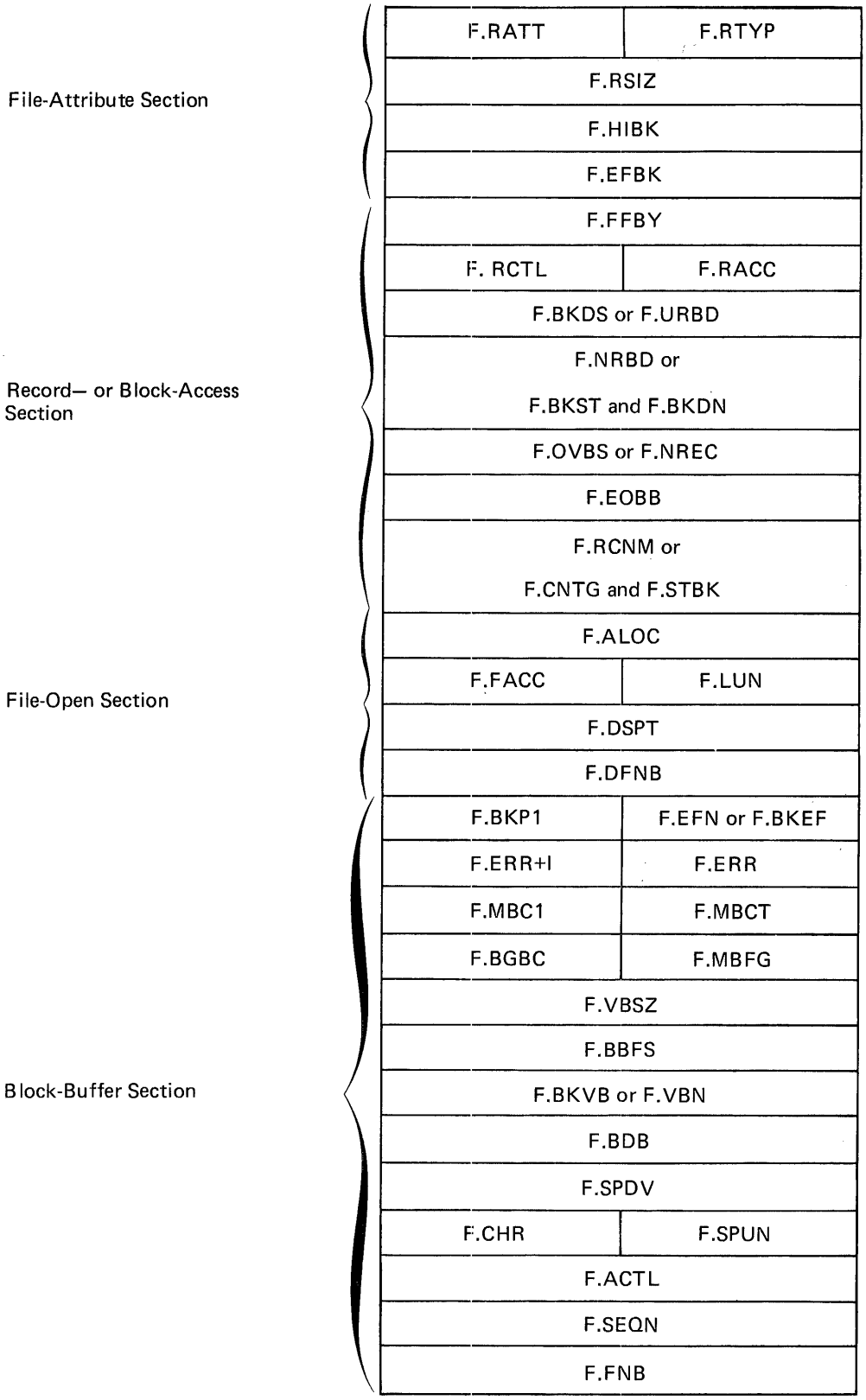


Figure A-1 File Descriptor Block Format

Table A-1
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.RTYP	1	Record-type byte. This byte is set, as follows, to indicate the type of records for the file: 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.RATT	1	Record-attribute byte. Bits 0 through 3 are set to indicate record attributes, as follows: Bit 0 = 1 to indicate that the first byte of a record is to contain a FORTRAN carriage-control character (FD.FTN); otherwise, it is 0. Bit 1 = 1 to indicate for a carriage-control device that a line feed is to be performed before the line is printed and a carriage return is to be performed after the line is printed (FD.CR); otherwise, it is 0. Bit 2 is not used. Bit 3 = 1 to indicate that records cannot cross block boundaries (FD.BLK); otherwise, it is 0.
F.RSIZ	2	Record-size word. This location contains the size of fixed-length records or indicates the size of the largest record that currently exists in a file of variable-length records.
F.HIBK	4	Indicates the highest virtual-block number allocated.
F.EFBK	4	Contains the end-of-file block number.
F.FFBY	2	Indicates the first free byte in the last block, or the maximum block size for magnetic tape.

(continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.RACC	1	Record access byte. Bits 0 through 3 of this byte define the record-access modes, as follows: Bit 0 = 1 to indicate READ$/WRITE$ mode (FD.RWM); otherwise, it is 0 to indicate GET$/PUT$ mode. Bit 1 = 1 to indicate random-access mode (FD.RAN) for GET$/PUT$ record I/O; otherwise, it is 0 to indicate sequential access mode.
F.RACC	1	Bit 2 = 1 to indicate locate mode (FD.PLC) for GET$/PUT$ record I/O; otherwise, it is 0 to indicate move mode. Bit 3 = 1 to indicate that PUT$ operation in sequential mode does not truncate the file (FD.INS); otherwise, it is 0 to indicate that PUT$ operation in sequential mode truncates the file.
F.RCTL	1	Device-characteristics byte. Bits 0 through 5 define the characteristics of the device associated with the file, as follows: Bit 0 = 1 to indicate a record-oriented device (FD.REC), e.g., a Teletype or line printer; a value of 0 indicates a block-oriented device, e.g., a disk or DECtape. Bit 1 = 1 to indicate a carriage-control device (FD.CCL); otherwise, it is 0. Bit 2 = 1 to indicate a teleprinter device (FD.TTY); otherwise, it is 0. Bit 3 = 1 to indicate a directory device (FD.DIR); otherwise, it is 0. Bit 4 = 1 to indicate a single-directory device (FD.SDI). An MFD is used, but no UFD's are present.

(continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.RCTL (Cont.)		Bit 5 = 1 to indicate a block-oriented device that is inherently sequential in nature (FD.SQD). A record-oriented device is assumed to be sequential in nature; therefore, this bit is not set for such devices.
F.BKDS or F.URBD	4	Contains the block-I/O-buffer descriptor. Contains the user-record-buffer descriptor.
F.NRBD or F.BKST and	4 2	Contains the next record-buffer descriptor. 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.
F.CNTG	2	Contains a numeric value defining the number of blocks to be allocated in creating a new file. This cell has meaning only before the file is opened. A value of 0 means leave the file empty; a positive value means allocate the specified number of blocks as a contiguous area and make the file contiguous; a negative value means allocate the specified number of blocks as a noncontiguous area and make the file noncontiguous.
F.STBK	2	Contains the address of the statistics block in the user program.
F.ALOC	2	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.

(continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.LUN	1	Contains the logical unit number associated with the FDB.
F.FACC	1	File-access byte. This byte indicates the access privileges for a file, as summarized below: Bit 0 = 1 if the file is accessed for read only (FA.RD). Bit 1 = 1 if the file is accessed for writing (FA.WRT). Bit 2 = 1 if the file is accessed for extending (FA.EXT).
F.ACC	1	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 one (1): Bit 6 = 1 if not superseding an existing file at file-create time (FA.NSP).
F.DSPT	2	Contains the dataset-descriptor pointer.
F.DFNB	2	Contains the default filename block pointer.
F.BKEF or F.EFN	1	Contains the block I/O event flag. Contains the record I/O event flag.
F.BKP1	1	Contains bookkeeping bits for FCS internal control.

(continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.ERR	1	Error return code byte. A negative value indicates an error condition.
F.ERR+1	1	Used in conjunction with F.ERR above. If F.ERR is negative, the following applies: F.ERR+1 = 0 to indicate that error code is an I/O error code (see IOERR$ error codes in Appendix I). F.ERR+1 = negative value to indicate that error code is a Directive Status Word error code (see DRERR$ error codes in Appendix I).
F.MBCT	1	Indicates the number of buffers to be used for multiple buffering.
F.MBC1	1	Indicates the actual number of buffers currently in use.
F.MBFG	1	Multiple-buffering flag word. Contains either one of the multiple buffering flags, as follows: Bit 0 = 1 to indicate read-ahead (FD.RAH). Bit 1 = 1 to indicate write-behind (FD.WBH).
F.BGBC	1	Big-buffer-block count in number of blocks (not implemented).
F.VBSZ	2	Device-buffer-size word. Contains the virtual-block size (in bytes).
F.BBFS	2	Indicates the block-buffer size.
F.BKVB or F.VBN	4	Contains the virtual-block number in the user program for block I/O. Contains the virtual-block number.
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.

(continued on next page)

Table A-1 (Cont.)
FDB Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
F.SPDV	2	Spooler output device designation (IAS only).
F.SPUN	1	Spooler output unit designation (IAS only).
F.CHR	1	Reserved for system use.
F.ACTL	2	The low-order byte of this word indicates the number of retrieval pointers to be used for the file.
F.ACTL	2	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	-	Defines the beginning address of the filename-block portion of the FDB.

APPENDIX B

FILENAME BLOCK

The format of a filename block is illustrated in Figure B-1. The offsets within the filename block are described in Table B-1.

The offset names in a filename block may be defined either locally or globally, as shown below:

```
NBOF$L                  ;DEFINE OFFSETS LOCALLY.

NBOFF$  DEF$L           ;DEFINE OFFSETS LOCALLY.

NBOFF$  DEF$G           ;DEFINE OFFSETS GLOBALLY.
```

NOTE

When referring to filename-block locations, it is essential to use the symbolic offset names, rather than the actual addresses of such locations. The position of information within the filename block may be subject to change from release to release, whereas the offset names remain constant.

Table B-1
Filename-Block Offset Definitions

OFFSET	SIZE (in bytes)	CONTENTS
N.FID	6	File-identification field.
N.FNAM	6	Filename field; specified as 9 characters that are stored in Radix-50 format.
N.FTYP	2	File-type field; specified as 3 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).
N.NEXT	2	Context for next .FIND operation.
N.DID	6	Directory-identification field.
N.DVNM	2	ASCII device-name field.
N.UNIT	2	Unit-number field (binary).

The bit definitions of the filename-block status word (N.STAT) in the FDB and their significance are described in Table B-2.

Symbols marked with an asterisk (*) in Table B-2 indicate bits that are set if the associated information is supplied through an ASCII dataset descriptor.

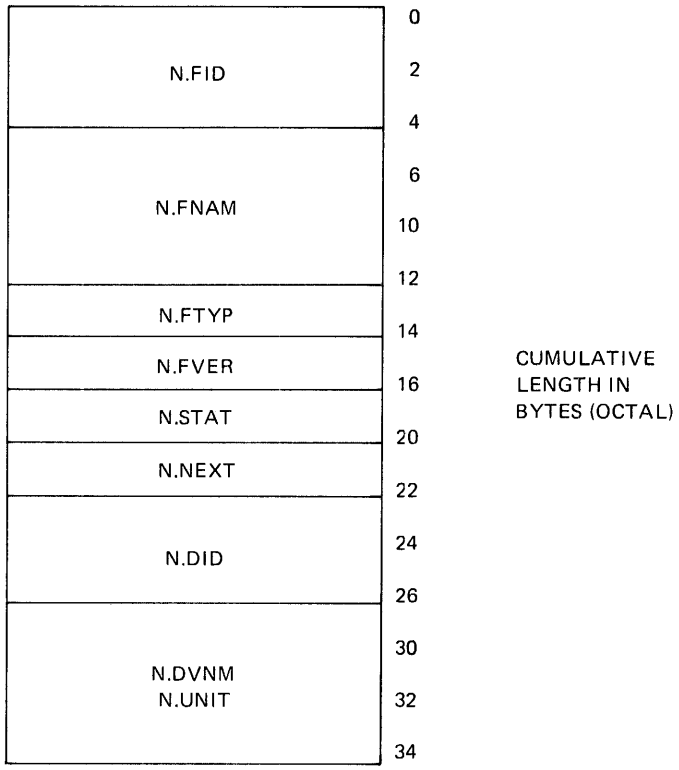


Figure B-1 Filename-Block Format

Table B-2
Filename-Block Status Word (N.STAT)

SYMBOL	VALUE (in octal)	MEANING
NB.VER*	1	Set if explicit file version number is specified.
NB.TYP*	2	Set if explicit file type is specified.
NB.NAM*	4	Set if explicit filename is specified.

(continued on next page)

Table B-2 (Cont.)
Filename-Block Status Word (N.STAT)

SYMBOL	VALUE (in octal)	MEANING
NB.SVR	10	Set if wildcard file version number is specified.
NB.STP	20	Set if wildcard file type is specified.
NB.SNM	40	Set if wildcard filename is specified.
NB.DIR*	100	Set if explicit directory string (UIC) is specified.
NB.DEV*	200	Set if explicit device-name string is specified.
NB.SD1	400	Set if group portion of UIC contains wildcard specification.[1]
NB.SD2	1000	Set if owner portion of UIC contains wildcard specification.[1]

[1] Although NB.SD1 and NB.SD2 are defined, they are not set or supported by FCS.

APPENDIX C

SUMMARY OF I/O-RELATED SYSTEM DIRECTIVES

Table C-1 contains a summary of the I/O-related system directives in alphabetical order for ready reference. The parameters that may be specified with a directive are also described in the order of their appearance in the directive. These directives are described in detail in the Executive Reference Manual of the host operating system.

Table C-1
Summary of I/O-Related System Directives

DIRECTIVE	FUNCTION AND PARAMETERS
ALUN$	Assigns a logical unit number to a physical device: lun = Logical unit number. dev = Physical device name (2 ASCII characters). unt = Physical device-unit number.
GLUN$	Fills a 6-word buffer with information about a physical unit: lun = Logical unit number. buf = Address of a 6-word buffer in which the LUN information is to be stored.
GMCR$	Transfers an 80-byte MCR/PDS command line to the issuing task. No parameters are required in this directive.
QIO$	Places an I/O request in the device queue associated with the specified logical unit number: fnc = I/O function code. lun = Logical unit number. efn = Event flag number. pri = Priority of the request (IAS only).

(continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

DIRECTIVE	FUNCTION AND PARAMETERS
QIO$ (Cont.)	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>.
RCVD$	Receives a 13-word data block that has been queued (FIFO) by a send-data directive (see SDAT$ and SDRQ$ below). tsk = Name of the sending task. This field is ignored by RSX-11. The tsk parameter is specified as a null value (,) in RSX-11 for compatibility with IAS (see the description of the RCVD$ directive in the RSX-11M/M-PLUS Executive Reference Manual). buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block).
RCVS$	Receives a 13-word data block, if queued by a send-data directive (see SDAT$ AND SDRQ$ below), or suspends task if no data is queued: tsk = Name of the sending task. buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block). This directive is not supported in RSX-11.
RCVX$	Receives a 13-word data block, if queued by a send data directive (see SDAT$ and SDRQ$ below), or exits if data is not queued for the task: tsk = Name of the sending task. This field is ignored by RSX-11. The tsk parameter is specified as a null value (,) in RSX-11 for compatibility with IAS (see the description of the RCVX$ directive in the RSX-11M/M-PLUS Executive Reference Manual). buf = Address of the 15-word data buffer (2-word sending task name and 13-word data block). SDAT$ Queues (FIFO) a 13-word block of data for a task to receive: tsk = Name of the receiving task. buf = Address of the 13-word data buffer. efn = Event flag number.

(continued on next page)

Table C-1 (Cont.)
Summary of I/O-Related System Directives

DIRECTIVE	FUNCTION AND PARAMETERS
SDRQ$	Queues (FIFO) a 13-word block of data for a task to receive; also requests or resumes the execution of the receiving task: tsk = Name of the receiving task. par = Partition name of the receiving task. pri = Priority of the request. ugc = UIC group code. upc = UIC owner code. buf = Address of the 13-word data buffer. efn = Event flag number. This directive is not supported in RSX-11.

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. In IAS, the core image is pointed to by a file header block in the index file.

E.2 HOME BLOCK

The home block contains volume identification information that is formatted as shown in Table E-1. This block is located either in Logical Block 1 or at any even multiple of 256 blocks.

The offset names in the home block may be defined either locally or globally, as shown below:

```
HMBOF$  DEF$L          ;DEFINES OFFSETS LOCALLY.

HMBOF$  DEF$G          ;DEFINES OFFSETS GLOBALLY.
```

E.3 INDEX-FILE BIT MAP

The index-file bit map controls the use of file-header blocks in the index file. The bit map contains a bit for each file-header block contained in the index file. The bit for a file-header block is located by means of the file number of the file with which it is associated. The values of the bit map are as follows:

0 - Indicates that the file-header block is available. The file-control primitives can use this block to create a file.

1 - Indicates that the file-header block is in use. This block has already been used to create a file.

E.4 PREDEFINED FILE-HEADER BLOCKS

The first five file-header blocks are described below.

FILE-HEADER BLOCK	SIGNIFICANCE
Index-File Header	This is the standard header associated with the index file.
Storage-Map-File Header	The storage map is a file that is used to control the assignment of disk blocks to files.
Bad-Block-File Header	The bad-block file is a file that consists of unusable blocks (bad sectors) on the disk.
Master-File-Directory Header	This header block is associated with the master file directory for the disk. This directory contains entries for the index file, the storage-map file, the bad-block file, the master file directory (MFD), the checkpoint file, and all user file directories (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 illustration at the beginning of this section.

Table E-1
Home-Block Format

SIZE (in bytes)	CONTENT	OFFSET
2	Index-bit-map size.	H.IBSZ
4	Location of index bit map.	H.IBLB
2	Maximum files allowed.	H.FMAX
2	Storage-bit-map cluster factor.	H.SBCL
2	Disk-device type.	H.DVTY
2	Structure level.	H.VLEV
12.	Volume name (12 ASCII characters).	H.VNAM
4	Reserved.	
2	Volume owner's UIC.	H.VOWN
2	Volume-protection code.	H.VPRO
2	Volume characteristics.	H.VCHA
2	Default-file protection word.	H.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

(continued on next page)

Table E-1 (Cont.)
Home-Block Format

SIZE (in bytes)	CONTENT	OFFSET
100.	Volume-header label (not used).	--
82.	System-specific information (not used).	--
254.	Relative volume table (not used).	--
2	Checksum of home block (Words 0 through 255).	H.CHK2

APPENDIX F

FILE-HEADER BLOCK FORMAT

Table F-1 shows the format of the file-header block. The various areas within the file-header block are described in detail in the following sections. The offset names in the file-header block may be defined either locally or globally, as shown in the following statements:

```
FHDOF$  DEF$L          ;DEFINE OFFSETS LOCALLY.
FHDOF$  DEF$G          ;DEFINE OFFSETS GLOBALLY.
```

Table F-1
File Header Block

AREA	SIZE (in bytes)	CONTENT	OFFSET
HEADER AREA	1	Identification-area offset in words.	H.IDOF
	1	Map-area offset in words.	H.MPOF
	2	File number.	H.FNUM
	2	File sequence number.	H.FSEQ
	2	Structure level and system number.	H.FLEV
	-	Offset to file owner information, consisting of member number and group number.	H.FOWN
	1	Member number.	H.PROG
	1	Group number.	H.PROJ
	2	File-protection code.	H.FPRO
	1	User-controlled file characteristics.	H.UCHA

(continued on next page)

Table F-1 (Cont.)
File Header Block

AREA	SIZE (in bytes)	CONTENT	OFFSET
HEADER AREA (Cont.)	1	System-controlled file characteristics.	H.SCHA
	32.	User file attributes.	H.UFAT
	-	Size in bytes of header area of file-header block.	S.HDHD
IDENTIFICATION AREA	6	Filename (Radix-50).	I.FNAM
	2	File type (Radix-50).	I.FTYP
	2	File version number (binary).	I.FVER
	2	Revision number.	I.RVNO
	7	Revision date.	I.RVDT
	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

(continued on next page)

Table F-1 (Cont.)
File Header Block

AREA	SIZE (in bytes)	CONTENT	OFFSET
MAP AREA (Cont.)	1	Size (in bytes) of the logical-block-number field of a retrieval pointer (2, 3, or 4); only 3 is used.	M.LBSZ
	1	Words of retrieval pointers in use in the map area.	M.USE
	1	Maximum number of words of retrieval pointers available in the map area.	M.MAX
	-	Start of retrieval pointers.	M.RTRV
	-	Size in bytes of map area of file-header block.	S.MPHD
CHECKSUM WORD	2	Checksum of words 0 through 255.	H.CKSM

NOTE

The checksum word is the last word of the file-header block. Retrieval pointers occupy the space from the end of the map area to the checksum word.

F.1 HEADER AREA

The information in the header area of the file-header block consists of the following:

IDENTIFICATION AREA OFFSET - Word 0, Bits 0-7. This byte locates the start of the identification area relative to the start of the file-header block. This offset contains the number of words from the start of the header to the identification area.

MAP AREA OFFSET - Word 0, Bits 8-15. This byte locates the start of the map area relative to the start of the file-header block. This offset contains the number of words from the start of the header area to the map area.

FILE NUMBER - The file number defines the position this file-header block occupies in the index file, e.g., the index file is number 1, the storage bit map is file number 2, etc.

FILE SEQUENCE NUMBER - The file number and the file sequence number constitute the file identification number used by the system. This number is different each time a header is 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 UIC's are within the range [1,1] to [377,377]. UIC [1,1] is reserved for the system.

FILE PROTECTION CODE - This word specifies the manner in which the file can be used and who can use it. When creating the file, the user specifies the extent of protection desired for the file.

FILE CHARACTERISTICS - This word, consisting of 2 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), 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 7 words of this area are a direct image of the first 7 words of the FDB when the file is opened. The other 9 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:

FILENAME - The file's creator specifies a filename of up to 9 Radix-50 characters in length. This name is placed in the name field. The unused portion of the field (if any) is 0-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 (2 bytes, 3 bytes, and 2 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 (2 bytes each).

CREATION DATE - The date on which the file was created is kept in a 7-byte field having the same format as the revision date (see above).

CREATION TIME - The time of the file's creation is maintained in a 6-byte field having the same format as the revision time (see above).

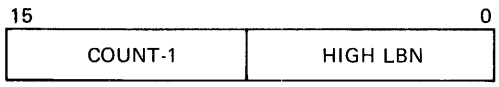
EXPIRATION DATE - The date on which the file becomes eligible to be deleted is kept in a 7-byte field having the same format as the revision date (see above). Use of expiration is not implemented.

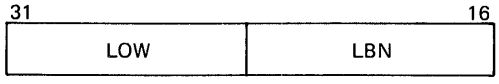
F.3 MAP AREA

The map area contains the information necessary to map virtual-block numbers to logical-block numbers. This is done by means of pointers, each of which points to an area of contiguous blocks. A pointer consists of a count field and a number field. The count field defines the number of blocks contained in the contiguous area pointed to, and the logical block number (LBN) field defines the block number of the first logical block in the area.

A value of n in the count field (see below) means that n+1 blocks are allocated, starting at the specified block number.

The retrieval pointer format used in the FILES-11 file structure is shown below:





NOTE

The remaining paragraphs in this appendix apply to IAS, and RSX-11 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 is used to number 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 the user.

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 June 19, 1974 Proposed Revision to the ANSI standard Magnetic Tape Labels and File Structure for Information Interchange (X3.27-1969). The only exception is that ANSI does not support spanned records.

G.1 VOLUME AND FILE LABELS

Tables G-1, G-2, and G-3 present the format of volume labels and file-header labels.

G.1.1 Volume Label Format

Table G-1
Volume Label Format

CHARACTER POSITION	FIELD NAME	LENGTH IN BYTES	CONTENTS
1-3	Label identifier	3	VOL
4	Label number	1	1
5-10	Volume identifier	6	Volume label. Any alphanumeric or special character in the center four columns of the ASCII code table.
11	Accessibility	1	Any alphanumeric or special character in the center four columns of the ASCII code table. A space indicates no restriction. All volumes produced by IAS or RSX-11 have a space in this position.
12-37	Reserved	26	Spaces

(continued on next page)

Table G-1 (Cont.)
Volume Label Format

CHARACTER POSITION	FIELD NAME	LENGTH IN BYTES	CONTENTS
38-51	Owner identification	14	The contents of this field are system-dependent and are used for volume protection purposes. See Section G.1.1.1 below.
52-79	Reserved	28	Spaces.
80	Label standard version	1	1

G.1.1.1 **Contents of Owner Identification Field** - The owner identification field is divided into the following three subfields and a single pad character:

1. System identification (positions 38 through 40),
2. Volume protection code (positions 41 through 44),
3. UIC (positions 45 through 50),
4. Pad character of one space (position 51).

The system identification consists of the following character sequence.

D%x

x is the machine code and can be one of the following.

8 - PDP-8
A - DECsystem-10
B - PDP-11
F - PDP-15

The D%x characters provide an identification method so that the remaining data in the owner identification field can be interpreted. In the case of tapes produced on PDP-11 systems, the system identification is D%B and the volume protection code and UIC are interpreted as described below.

The volume protection code in positions 41 through 44 defines access protection for the volume for four classes of users. Each class of user has access privileges specified in one of the four columns, as follows.

Position	Class
41	System (UIC no greater than [7,255])
42	Owner (group and member numbers match)
43	Group (group number matches)
44	World (any user not in one of the above)

One of the following access codes can be specified for each character position.

Code	Privilege
0	No access
1	Read only
2	Extend (append) access
3	Read/extend access
4	Total access

The UIC is specified in character positions 45 through 50. The first 3 characters are the group code in decimal. The next 3 are the user code in decimal.

The last character in the owner identification field is a space.

The following is an example of the owner identification field.

Owner identifier - D%B1410051102_ (_ indicates space)

1. The file was created on a PDP-11.

2. System and group have read access.
   Owner has total access.
   All others are denied access.

3. The UIC is [051,102].

G.1.2 User Volume Labels

User volume labels are never written or passed back to the user. If present, they are skipped.

G.1.3 File-Header Labels

The following information should be kept in mind when creating file-header labels.

- The Files-11 naming convention uses a subset (Radix-50) of the available ANSI character set for file identifiers.
- One character in the file identifier, the period (.), is fixed by Files-11.
- A maximum of 13 of the 17 bytes in the file identifier are processed by Files-11.
- It is strongly recommended that all file identifiers be limited to the Radix-50 PDP-11 character set and that no character other than the period (.) be used in the file type delimiter position for data interchange between PDP-11 and DECsystem-10 systems.
- For data interchange between DIGITAL and nonDIGITAL systems, the conventions listed above should be followed. If they are not, refer to Section G.1.3.1.

Tables G-2 and G-3 describe the HDR1 and HDR2 labels respectively.

Table G-2
File-Header Label (HDR1)

CHARACTER POSITION	FIELD NAME	LENGTH IN BYTES	CONTENT
1-3	Label identifier	3	HDR
4	Label number	1	1
5-21	File identifier	17	Any alphanumeric or special character in the center four columns of the ASCII code table.
22-27	File set identifier	6	Volume identifier of the first volume in the set of volumes.
28-31	File section number	4	Numeric characters. This field starts at 0001 and is increased by 1 for each additional volume used by the file.
32-35	File sequence number	4	File number within the volume set for this file. This number starts at 0001.
36-39	Generation number	4	Numeric characters.
40-41	Generation version	2	Numeric characters.
42-47	Creation date	6	_yyddd (_ indicates space) or l _00000 if no date.
48-53	Expiration date	6	Same format as creation date.
54	Accessibility	1	Space.
55-60	Block count	6	000000
61-73	System code	13	The three letters DEC followed by name of the system that produced the volume. See Section G.1.1.1. Examples: DECFILE11A DECSYSTEM10 Pad name with spaces.
74	Reserved	7	Spaces.

Table G-3
File Header Format (HDR2)

CHARACTER POSITION	FIELD NAME	LENGTH IN BYTES	CONTENT
1-3	Label identifier	3	HDR
4	Label number	1	2
5	Record format	1	F - fixed length D - variable length S - spanned U - undefined
6-10	Block length	5	Numeric characters.
11-15	Record length	5	Numeric characters.
16-50	System-dependent information	35	Positions 16 through 36 are spaces. Position 37 defines carriage control and can contain one of the following: A - first byte of record contains FORTRAN control characters, space - line feed/carriage return is to be inserted between records, M - the record contains all form-control information. If DEC appears in positions 61 through 63 of HDR1, position 37 must be as specified above. Positions 38 through 50 contain spaces.
51-52	Buffer offset	2	Numeric characters. 00 on tapes produced by Files-11. Not supported on input to Files-11.
53-80	Reserved	28	Spaces.

G.1.3.1 **File Identifier Processing by Files-11** - The following steps describe the processing of a file identifier by Files-11.

1. The first 9 characters at a maximum are processed by an ASCII to Radix-50 converter. The conversion continues until one of the following occurs:

   A conversion failure,
   9 characters are converted,
   A period (.) is encountered.

2. If the period is encountered, the next 3 characters after the period are converted and treated as the file type. If a failure occurs or all 9 characters are converted, the next character is examined for a period. If it is a period, it is skipped and the next 3 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 filename is placed in the first positions in the file identifier field. It can occupy up to 9 positions. It is followed by a period.

2. The file type of up to 3 characters is placed after the period. The remaining spaces are padded with spaces.

3. The version number is then placed in the generation and generation version number fields as described in the following formulas.

   a. $\text{generation number} = \left(\frac{\text{version \# - 1 + 1}}{100}\right)$

   b. $\text{generation version \#} = \left(\frac{\text{version \# - 1}}{\text{Modulo 100}}\right)$

NOTE

In both calculations, remainders are ignored.

The following are examples.

FILES-11 VERSION #	GENERATION #	GENERATION VER #
1	1	0
50	1	49
100	1	99
101	2	0
1010	11	9

G.1.4 End-of-Volume Labels

End-of-volume labels are identical to the file-header labels with the following exceptions:

1. Character positions 1 through 4 contain EOV1 and EOV2 instead of HDR1 and HDR2, respectively.
2. The block-count field contains the number of records in the last file section on the volume.

G.1.5 File-Trailer Labels

End-of-file labels (file-trailer labels) are identical with file-header labels, with the following exceptions:

1. Columns 1 through 4 contain EOF1 and EOF2 instead of HDR1 and HDR2, respectively.
2. The block count contains the number of data blocks in the file.

G.1.6 User File Labels

User file labels are never written or passed back to the user. If present, they are skipped.

G.2 FILE STRUCTURES

The file structures illustrated below are the types of file and volume combinations that the file processor produces. Additional sequences can be read and processed by the file processor.

The minimum block size and fixed length record size is 18 bytes. The maximum block size is 8192 bytes.

If HDR2 is not present, the data type is assumed to be fixed (F) and the block size and record size are assumed to be the default value for the file processor. 512 decimal bytes is the default for both block and record size.

The meaning of graphics used in the file structure illustrations is as follows.

1. * indicates a tape mark,
2. BOT indicates beginning of tape,
3. EOT indicates end of tape,
4. , indicates the physical record delimiter.

G.2.1 Single File Single Volume

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**

G.2.2 Single File Multivolume

BOT,VOL1,HDR1,HDR2*---DATA---*EOV1,EOV2**

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2**

G.2.3 Multifile Single Volume

BOT,VOL1,HDR1,HDR2*---DATA---*EOF1,EOF2*HDR1,HDR2*---DATA--*EOF1,EOF2**

G.2.4 Multifile Multivolume

BOT,VOL1,HDR1,HDR2*--DATA--*EOF1,EOF2*HDR1,HDR2*--DATA--*EOV1,EOV2**

BOT,VOL1,HDR1,HDR2*--DATA--*EOF1,EOF2*HDR1,HDR2*--DATA--*EOF1,EOF2**

G.3 END-OF-TAPE HANDLING

End-of-tape is handled automatically by the magnetic tape file processor. Files are continued on the next volume 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.

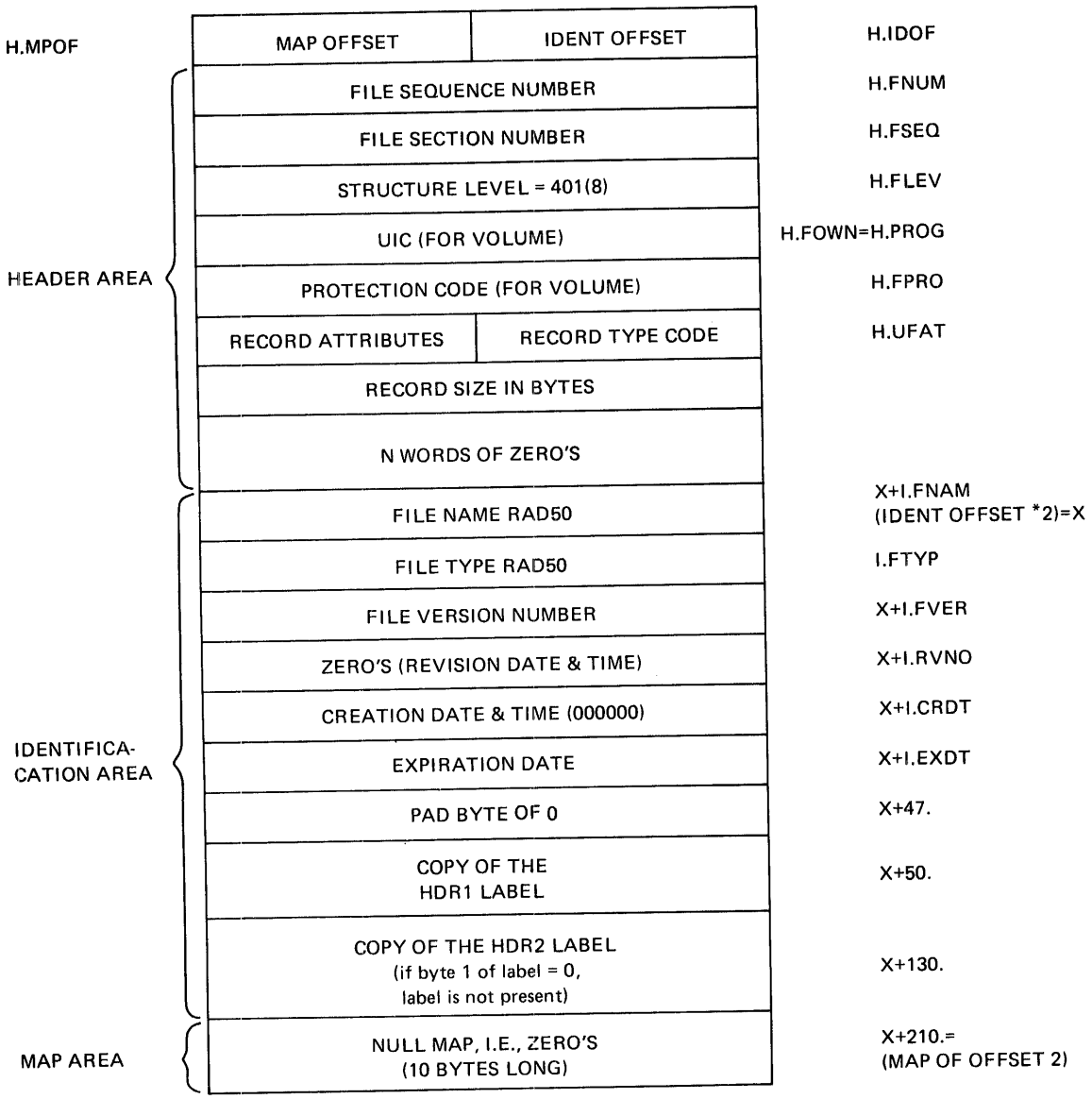


Figure G-1 ANSI Magnetic Tape File-Header Block (FCS Compatible)

APPENDIX H

STATISTICS BLOCK

The format of the statistics block is shown in Figure H-1 below. The statistics block is allocated manually in the user program as described in Item 3.d of Section 3.1.2.

Word 0	HIGH LOGICAL BLOCK NUMBER (0 if file is noncontiguous)
Word 1	LOW LOGICAL BLOCK NUMBER (0 if file is noncontiguous)
Word 2	SIZE (high)
Word 3	SIZE (low)
Word 4	LOCK COUNT ACCESS COUNT

Figure H-1 Statistics Block Format

APPENDIX I

ERROR CODES

This appendix lists the Directive Status Word error codes and the I/O error codes returned by the system.

```
        .TITLE  QIOMAC - QIOSYM MACRO DEFINITION
;
; DATE OF LAST MODIFICATION:
;
;       C.A. D'ELIA     07-MAR-79
;
;
; ***** ALWAYS UPDATE THE FOLLOWING TWO LINES TOGETHER
        .IDENT  /0335/
        QI.VER=0335

;
; COPYRIGHT (C) 1973,1979
; 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.
;
        .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.,<SHARABLE RESOURCE IN USE>
        .IOER.  IE.OVR,-18.,<ILLEGAL OVERLAY REQUEST>
        .IOER.  IE.BYT,-19.,<ODD BYTE COUNT (OR VIRTUAL ADDRESS)>
        .IOER.  IE.BLK,-20.,<LOGICAL BLOCK NUMBER TOO LARGE>
        .IOER.  IE.MOD,-21.,<INVALID UDC MODULE #>
        .IOER.  IE.CON,-22.,<UDC CONNECT ERROR>
        .IOER.  IE.BBE,-56.,<BAD BLOCK ON DEVICE>
        .IOER.  IE.STK,-58.,<NOT ENOUGH STACK SPACE (FCS OR FCP)>
        .IOER.  IE.FHE,-59.,<FATAL HARDWARE ERROR ON DEVICE>
        .IOER.  IE.EOT,-62.,<END OF TAPE DETECTED>
        .IOER.  IE.OFL,-65.,<DEVICE OFF LINE>
        .IOER.  IE.BCC,-66.,<BLOCK CHECK, CRC, OR FRAMING ERROR>
```

```
;
; FILE PRIMITIVE CODES
;

        .IOER.  IE.NOD,-23.,<CALLER'S NODES EXHAUSTED>
        .IOER.  IE.DFU,-24.,<DEVICE FULL>
        .IOER.  IE.IFU,-25.,<INDEX FILE FULL>
        .IOER.  IE.NSF,-26.,<NO SUCH FILE>
        .IOER.  IE.LCK,-27.,<LOCKED FROM 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>

;
; 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 CODES
;

        .IOER.  IE.AST,-80.,<NO AST SPECIFIED IN CONNECT>
        .IOER.  IE.NNN,-68.,<NO SUCH NODE>
        .IOER.  IE.NFW,-69.,<PATH LOST TO PARTNER> ;THIS CODE MUST BE ODD
        .IOER.  IE.BLB,-70.,<BAD LOGICAL BUFFER>
        .IOER.  IE.TMM,-71.,<TOO MANY OUTSTANDING MESSAGES>
        .IOER.  IE.NDR,-72.,<NO DYNAMIC SPACE AVAILABLE>
        .IOER.  IE.CNR,-73.,<CONNECTION REJECTED>
        .IOER.  IE.TMO,-74.,<TIMEOUT ON REQUEST>
        .IOER.  IE.NNL,-78.,<NOT A NETWORK LUN>

;
; 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>

;
; 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.     ;(RX11) FLOPPY DISK SUCCESSFUL COMPLETION
                                ;OF A READ PHYSICAL, AND DELETED
                                ;DATA MARK WAS SEEN IN SECTOR HEADER
                                ; LAST SECTOR.
        DEFIN$  IS.TNC,+02.     ;(PCL) SUCCESSFUL TRANSFER BUT MESSAGE
                                ;TRUNCATED (RECEIVE BUFFER TOO SMALL).
        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.

;
; 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 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

; ******
;
; THE NEXT AVAILABLE ERROR NUMBER IS: -90.
; ALL LOWER NUMBERS ARE IN USE !!
;
; *****
        .IF     EQ,$$MSG
        .MACRO  IOERR$  A
        .ENDM   IOERR$
        .ENDC
        .ENDM   IOERR$
```

```
;
; DEFINE THE DIRECTIVE ERROR CODES RETURNED IN THE DIRECTIVE STATUS WORD
;
;       FILE CONTROL SERVICES (FCS) RETURNS THESE CODES IN THE BYTE F.ERR
; OF THE FILE DESCRIPTOR BLOCK (FDB).  TO DISTINGUISH THEM FROM THE
; OVERLAPPING CODES FROM HANDLER AND FILE PRIMITIVES, THE BYTE
; F.ERR+1 IN THE FDB WILL BE NEGATIVE FOR A DIRECTIVE ERROR CODE.
;
        .MACRO  DRERR$  $$$GBL
        .MCALL  .QIOE.,DEFIN$
        .IF     IDN,<$$$GBL>,<DEF$G>
        ...GBL=1
        .IFF
        ...GBL=0
        .ENDC
        .IIF    NDF,$$MSG,$$MSG=0
;
; STANDARD ERROR CODES RETURNED BY DIRECTIVES IN THE DIRECTIVE STATUS WORD
;
        .QIOE.  IE.UPN,-01.,<INSUFFICIENT DYNAMIC STORAGE>
        .QIOE.  IE.INS,-02.,<SPECIFIED TASK NOT INSTALLED>
        .QIOE.  IE.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.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
;
;
        .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>
        ...GBL=1
        .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
;
; 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 HANDLER CODES
;
        .WORD.  IO.WLB,000,001  ;WRITE LOGICAL BLOCK
        .WORD.  IO.RLB,000,002  ;READ LOGICAL BLOCK
        .WORD.  IO.LOV,010,002  ;LOAD OVERLAY (DISK DRIVER)
        .WORD.  IO.SCF,001,000  ;SHADOW COPY FUNCTION (DISKS ONLY)
        .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  ;(RX11 DISK) WRITE PHYSICAL W/ DELETED DATA
        .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 LOWER CASE CONVERT
        .WORD   IO.RTM,200,002  ;(TTY) READ WITH TIME OUT
        .WORD   IO.RDB,200,002  ;(CARD READER) READ BINARY MODE
        .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.RLC,020,002  ;(DISK,MAGTAPE) READ LOGICAL W/ READCHECK
        .WORD   IO.ATA,010,003  ;(TTY) ATTACH WITH AST'S
        .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.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.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.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
.WORD.   IO.STA,000,033  ;(LPA11) START DATA TRANSFER
.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
.WORD.   IO.ITI,000,036  ;(UDC) TIMER - INITIALIZE


;
; COMMUNICATIONS FUNCTIONS
;

        .WORD.   IO.CPR,010,033  ;CONNECT NO TIMEOUTS
        .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
```

```
;
; 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 DIAGNOSITCS.  ALL DIAGNOSTIC
; FUNCTION 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 (NON-TRANSFER)
        .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

;
; 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,$$MSG
        .MCALL  .IOMG.
        .IOMG.  SYM,LO,<MSG>
        .ENDC
        .ENDM   .IOER.
;
; I/O ERROR CODES ARE DEFINED THOUGH THIS MACRO WHICH THEN INVOKES THE
; ERROR MESSAGE GENERATING MACRO, ERROR CODES -129 THROUGH -256
; ARE USED IN THE QIOSYM.MSG FILE
;
        .MACRO  .QIOE.  SYM,LO,MSG
        DEFIN$  SYM,LO
        .IF     GT,$$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>
        .ASCIZ  "MSG"
        .EVEN
        .IIF    LT,^O<$$$MAX+<LO>>,$$$MAX=-^O<LO>
        .ENDM   .IOMG.
;
; DEFINE THE SYMBOL SYM WHERE LO IS IS THE LOW ORDER BYTE, HI IS THE HIGH BYTE
;
        .MACRO  .WORD.  SYM,LO,HI
        DEFIN$  SYM,<HI*400+LO>
        .ENDM   .WORD.
```

APPENDIX J

FIELD SIZE SYMBOLS

Definitions for these symbols are contained in the System Library.

S.BFHD - size of FSR block-buffer header in bytes.

S.FATT - size of FDB file-attribute area in bytes.

S.FDB - size of FDB in bytes (including name block).

S.FNAM - size of filename in bytes (stored in RAD-50).

S.FNB - size of filename block in bytes.

S.FNBW - size of filename block in words.

S.FNTY - size of filename and file type in words (stored in RAD-50).

S.FSR2 - size of FSR2 (basic impure area).

S.FTYP - size of file type in bytes (in RAD-50).

S.NFEN - size of a complete filename in bytes -- file ID, name, type, and version.

APPENDIX K

RSX-11M/M-PLUS FCS LIBRARY SYSGEN OPTIONS

The system manager has the option of selecting one of several FCS libraries as the default FCS library. The following list contains the FCS libraries that are available with each RSX-11M or RSX-11M-PLUS system, and a brief description of each:

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 magtape support and "big buffer" (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.
[1,1]FCSMBF.OBJ	Provides multiple buffering support, "big buffer" support, and ANSI magtape support in addition to the standard FCS routines

INDEX

IAS/RSX-11
I/O Operations
Reference Manual
AA-2515D-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 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

Please cut along this line.

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here

Cut Along Dotted Line